



ULTIMATE

# Monorepo and Bazel for Building Apps at Scale

Level Up Your Large-Scale Application  
Development with Monorepo and Bazel for  
Enhanced Productivity, Scalability,  
and Integration

Javier Antoniucci



ULTIMATE

# Monorepo and Bazel for Building Apps at Scale

Level Up Your Large-Scale Application  
Development with Monorepo and Bazel for  
Enhanced Productivity, Scalability,  
and Integration

Javier Antoniucci

# **Ultimate Monorepo and Bazel for Building Apps at Scale**

---

Level up Your Large-Scale  
Application  
Development With Monorepo and  
Bazel  
for Enhanced Productivity,  
Scalability,  
and Integration

---

**Javier Antoniucci**



[www.orangeava.com](http://www.orangeava.com)

Copyright © 2024 Orange Education Pvt Ltd, AVA™

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor **Orange Education Pvt Ltd** or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

**Orange Education Pvt Ltd** has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

**First published:** May 2024

**Published by:** Orange Education Pvt Ltd, AVA™

**Address:** 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,  
N1 7AA, United Kingdom

**ISBN:** 978-81-97223-91-4

[www.orangeava.com](http://www.orangeava.com)

# Dedicated To

*My Beloved Wife Julieta, My Son Franco, and My Daughter  
Sofia,  
Whose Unwavering Support and Love Have Been My  
Bedrock*

To the architects of innovation, the engineers of progress, and the curators of efficiency,

This book, *Ultimate Scalable Monorepo Apps with Bazel*, is dedicated to you, the technical pioneers who continually push the boundaries of what's possible in the realm of software development. In an era where the complexity of applications grows exponentially and the demand for scalability is non-negotiable, you rise to the challenge with unwavering determination and ingenuity.

To the developers who juggle myriad libraries and modules, striving for coherence in a sea of code, this work acknowledges your struggles and offers guidance. To the architects who design systems that stand robust against the test of scalability and change, this book aims to be a beacon, illuminating the path toward more streamlined and efficient workflows.

We dedicate this volume to the unsung heroes in the server rooms, the thinkers in the quiet corners of bustling tech hubs, and the educators shaping the minds of the next wave of software savants. Your commitment to embracing and mastering new paradigms, like Monorepo and Bazel, inspires a future where large-scale applications are not just feasible but flourish with unprecedented efficiency and reliability.

In these pages, may you find the knowledge to craft your monolithic repositories with precision, to harness the power of Bazel in orchestrating builds and tests, and to elevate

your projects to new heights of performance and manageability.

Together, let's build not just applications, but legacies — robust frameworks and systems that will empower generations of developers to come. Here's to the builders of the digital age — may this book serve as both a tribute and a tool in your journey toward creating software that defines the future.

With admiration and respect,

*Javier Antonucci*

# About the Author

**Javier Antoniucci** is a seasoned software engineer and architect with a profound passion for engineering processes and efficient team dynamics in software development. Beginning his programming journey at the tender age of 11, he has accumulated over 25 years of industry experience, with a significant portion dedicated to large financial and insurance corporations.

His expertise extends to leading digital transformation projects and defining organizational models for software governance, demonstrating a keen ability to navigate complex technical and cultural landscapes. Notable achievements include the deployment of corporate models for Global Open API Governance across several tier-1 and tier-2 banks and spearheading pivotal digital transformation initiatives.

As a head architect, he has been instrumental in developing online banking platforms, branch banking systems, and DevOps architectures, showcasing a comprehensive understanding of the financial sector's technological needs. His contributions to Big Data, including smart alerts and forecasting, highlight his proficiency in leveraging data to enhance business intelligence.

Before focusing on his current role as Chief Technology Officer at GFT Group, Javier honed his skills at prestigious firms such as Deloitte and Altran Technology, managing elite multidisciplinary teams and driving technical and organizational transformation. His academic credentials, including a degree in Software Engineering, an executive-level Master's degree in Business Administration, and another Master's degree in Big Data and Business

Intelligence, complement his hands-on experience in the industry.

In his current role, he continues to influence the tech landscape, leading Thought Machine Vault and Open API practices globally, and driving innovation in technology at GFT Technologies. His commitment to excellence is evident in his oversight of high-performance teams delivering critical projects for top-tier banks and financial institutions, showcasing his prowess in integrating technology innovation at scale.

Javier's comprehensive background, spanning technical leadership, strategic planning, and educational achievements, positions him as a trusted voice in the field of software development, making *Ultimate Scalable Monorepo Apps with Bazel* an essential read for developers looking to scale their expertise in today's dynamic technological environment.



# About the Technical Reviewer

**Abhay Joshi** is currently working as a Principal Software Engineer at tiket.com with over 10 years of experience in the software development industry. His career showcased exceptional technical prowess, leadership abilities, and a passion for driving innovation. With a strong foundation in computer science and software engineering principles, Abhay has successfully led numerous high-impact projects from conception to delivery, earning a reputation for excellence and reliability among peers and stakeholders alike.

As a Principal Software Engineer, he provides technical leadership, architectural guidance, and hands-on expertise to cross-functional teams. He excels at designing and implementing scalable, robust, and maintainable software solutions that meet the complex requirements of modern applications. Leveraging his deep understanding of software design patterns, development methodologies, and emerging technologies, he collaborates closely with product managers, designers, and engineers to deliver cutting-edge solutions that drive business success and exceed customer expectations.

His expertise spans a wide range of technologies, including Golang, Java, Python, JavaScript, Kotlin, Swift, Kubernetes, CI/CD tools, and cloud platforms such as AWS and Google Cloud. He excels in building distributed systems, microservices architectures, and highly available applications that are resilient to failures and adaptable to changing business needs. With a keen eye for optimization and performance tuning, he continuously strives to enhance the efficiency and reliability of software systems, ensuring optimal performance, cost, and scalability at all levels.

Beyond his technical skills, he is known for his leadership, mentorship, and commitment to fostering a culture of collaboration, innovation, and continuous learning within organizations. He thrives in dynamic and fast-paced environments, where he can leverage his expertise to solve complex problems, drive strategic initiatives, and inspire teams to achieve excellence, empowering teams to reach their full potential and deliver impactful results.

With his blend of technical expertise, leadership acumen, and passion for innovation, he continues to make significant contributions to the software development community and shape the future of technology.

# Acknowledgements

This book is not just a compilation of insights and methodologies; it's a tapestry woven from the collective wisdom of a vibrant community, the dedication of exceptional individuals, and the intellectual curiosity of you, the reader.

## **The Community at Large**

At the heart of this book's genesis are the dynamic and ever-evolving Bazel, Aspect Build, and Monorepo tools communities. The Bazel community, with its commitment to building reliable and efficient software at scale, has been a beacon of inspiration and knowledge. Aspect Build's enhancements and extensions to Bazel have provided critical insights into optimizing development workflows. Similarly, the diverse array of Monorepo tools and their respective communities offer strategies to manage codebases effectively, enabling a holistic view of large-scale application development.

## **Exceptional DevOps and Contributors**

Special acknowledgment to DevOps experts like Son Luong Ngoc, Sergio Fernandez, Salim Boudriya, Gonzalo Ruiz de Villa, Daniel Garcia, for sharing their experience and uplifting the community. Their contributions inspire countless professionals in the field.

## **Emilio Guillot**

A distinct and heartfelt acknowledgment is reserved for Emilio Guillot, a beacon of knowledge and inspiration in the realm of global architecture. His wisdom has influenced many core concepts and strategies discussed in this book.

Emilio's experience and vision have been a guiding light, much like a lighthouse, illuminating the path forward in digital transformation and application development.

### **You, the Reader**

Lastly, but most importantly, this acknowledgment extends to you, the reader. Your curiosity, desire for skill enhancement, and dedication to navigating large-scale application development challenges give purpose to this work.

As you embark on or continue your journey in building robust and scalable applications, may the contents of this book serve as a reliable compass and a source of inspiration. Remember, the path of learning is perpetual, and each challenge surmounted is a step toward mastery. Thank you for allowing this book to be a part of your professional voyage.

# Foreword

**-by Emilio Guillot  
(Former BBVA CTO)**

In the ever-evolving landscape of software development, organizations are continuously challenged to adapt, innovate, and streamline their processes to stay competitive and efficient. This book explores how the monorepository (monorepo) approach, coupled with the Bazel build tool, is not just a technical shift but a transformative strategy for organizations, particularly for sectors resistant to change, such as banking.

The transition from legacy systems to modern processes is challenging, particularly in conservative organizations with deeply ingrained practices. Monorepo offers a streamlined platform that bridges the gap between old and new, simplifying the transition and minimizing disruption. In banking, for example, monorepo can integrate cutting-edge technologies while maintaining the rigor and reliability that the sector demands.

Another challenge is the disconnect between management and technical teams. Monorepo's centralized structure provides a clear view of development, fostering better understanding and alignment.

Finally, managing numerous tools and processes can be chaotic. Bazel, integrated with monorepo, automates and streamlines builds and tests, bringing order to the development process. This book will guide you to leverage monorepo and Bazel to create a robust, scalable, and maintainable software ecosystem.

As we embark on this journey together, our goal is to not only understand the mechanics of monorepo and Bazel but

to appreciate their potential to transform the very DNA of organizations, ushering them into a new era of efficiency, collaboration, and innovation.

# Preface

Managing large-scale applications development has become increasingly complex, necessitating robust methodologies and tools to streamline the process. *Ultimate Scalable Monorepo Apps with Bazel* serves as a comprehensive guide, offering in-depth insights into leveraging Monorepo and Bazel to enhance productivity and scalability. This book is structured to provide a logical progression, starting with foundational concepts and advancing to intricate techniques, catering to both newcomers and seasoned practitioners.

**Chapters 1-5**: The initial chapters lay the groundwork, introducing the Monorepo approach and Bazel's pivotal role in this ecosystem. Readers will gain a clear understanding of the synergy between Monorepo and Bazel, appreciating how this combination simplifies dependency management, improves build times, and fosters collaboration.

**Chapters 6-10**: Subsequent chapters delve into practical applications, guiding users through setting up Bazel, configuring build rules, and implementing effective testing strategies within a Monorepo setup. These sections are designed to equip readers with the skills necessary to leverage Bazel's full potential in real-world scenarios.

**Chapters 11-14**: Advanced topics are explored in the latter chapters, addressing the challenges of scaling, dependency management, and continuous integration in the context of Monorepos. The book also looks ahead, discussing future trends and the evolving landscape of Monorepo tooling, preparing readers to adapt to forthcoming changes in the industry. Through a blend of theoretical concepts, practical examples, and real-world case studies, this book aims to provide a holistic view of building and managing large-scale

applications with Monorepo and Bazel, enabling developers and teams to harness these powerful tools to their fullest potential.



# Downloading the code bundles and colored images

Please follow the link or scan the QR code to download the *Code Bundles and Images* of the book:

<https://github.com/OrangeAVA/Ultimate-Monorepo-and-Bazel-for-Building-Apps-at-Scale>



The code bundles and images of the book are also hosted on

<https://rebrand.ly/7tl9f8u>



In case there's an update to the code, it will be updated on the existing GitHub repository.

# Errata

We take immense pride in our work at **Orange Education Pvt Ltd** and follow best practices to ensure the accuracy of our content to provide an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

[errata@orangeava.com](mailto:errata@orangeava.com)

Your support, suggestions, and feedback are highly appreciated.

## **DID YOU KNOW**

Did you know that Orange Education Pvt Ltd offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.orangeava.com](http://www.orangeava.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: [info@orangeava.com](mailto:info@orangeava.com) for more details.

At [www.orangeava.com](http://www.orangeava.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on AVA™ Books and eBooks.

## **PIRACY**

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [info@orangeava.com](mailto:info@orangeava.com) with a link to the material.

## **ARE YOU INTERESTED IN AUTHORING WITH US?**

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please write to us at [business@orangeava.com](mailto:business@orangeava.com). We are on a journey to help developers and tech professionals to gain insights on the present technological advancements and innovations happening across the globe and build a community that believes Knowledge is best acquired by sharing and learning with others. Please reach out to us to learn what our audience demands and how you can be part of this educational reform. We also welcome ideas

from tech experts and help them build learning and development content for their domains.

## **REVIEWS**

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at Orange Education would love to know what you think about our products, and our authors can learn from your feedback. Thank you!

For more information about Orange Education, please visit [www.orangeava.com](http://www.orangeava.com).

# Table of Contents

## **1. Introduction**

Introduction

Structure

Understanding the Monorepo Approach

*Welcome to the True Continuous Integration*

*Drawbacks of Polyrepo*

*Benefits of Monorepo*

*A Bit of History*

*Typical Fears about Monorepos*

*Typical Challenges in Implementing a Monorepo*

The Power of Bazel in Monorepo Development

*A Bit of History*

*Bazel Features*

Understanding the Logic Behind Bazel's Design

*When to Use a Monorepo*

*When Not to Use a Monorepo*

*When to Use Bazel*

*When Not to Use Bazel*

Conclusion

Recommended Readings

## **2. Getting Started with Bazel**

Introduction

Structure

Installing and Setting Up Bazel

*Installing Bazelisk*

Building Your First Bazel Project (Java)

Bazel Basics

*WORKSPACE File*

*BUILD Files*

*Build Rules*

[Targets](#)

[Labels](#)

[Packages](#)

[Queries](#)

[Dependencies](#)

[Bazel Sandboxing](#)

[Conclusion](#)

[Recommended Readings](#)

### **3. Bazel Build Rules and Configuration**

[Introduction](#)

[Structure](#)

[Exploring Bazel's Rule-based Build System](#)

[Your First Bazel Rule](#)

[Dissecting a Core Bazel Rule](#)

[Customizing Build and Compilation Rules](#)

[Key Components in Rule Creation](#)

[Solving any Custom Needs not Served by Default Rules](#)

[Writing and Executing a Genrule](#)

[Bazel Configuration](#)

[WORKSPACE File](#)

[BUILD File](#)

[Bazel Flags](#)

[.bazelrc File](#)

[Location](#)

[Syntax](#)

[Best Practices](#)

[Commonly Used Options](#)

[Conclusion](#)

[Recommended Readings](#)

### **4. Testing Strategies in a Monorepo**

[Introduction](#)

[Structure](#)

[Testing Strategies in Bazel](#)

[\*Efficient Testing Strategies\*](#)  
[\*Scalable Testing Strategies\*](#)  
[\*Implementing Testing Strategies\*](#)  
[Writing and Running Unit Tests with Bazel](#)  
[\*Managing Multiple Unit Tests\*](#)  
[\*Reporting Unit Test Coverage\*](#)  
[\*Performance Testing\*](#)  
[\*User Acceptance Tests \(End-to-end\)\*](#)  
[Achieving Test Isolation and Parallelism](#)  
[\*Test Isolation\*](#)  
[\*Test Parallelism\*](#)  
[Conclusion](#)  
[Recommended Readings](#)

## **5. Dependency Management and Versioning**

[Introduction](#)  
[Structure](#)  
[Managing Internal and External Dependencies](#)  
[\*Internal Dependencies\*](#)  
[\*Best Practices\*](#)  
[\*External Dependencies\*](#)  
[\*Conflict Resolution\*](#)  
[Bazel MODULES: A Modern Way for Handling External Dependencies](#)  
[\*Declaring Dependencies with MODULES\*](#)  
[\*Configuring an Air-Gapped Bazel Build\*](#)  
[Enforcing Versioning and Compatibility in a Monorepo](#)  
[\*Querying Dependencies and Getting Graphs\*](#)  
[Integrating Bazel Within an IDE](#)  
[Conclusion](#)  
[Further Readings](#)

## **6. Hello-World Using Other Languages and Platforms**

[Introduction](#)  
[Structure](#)

## Android/Kotlin

[Setting up Your Android/Kotlin Bazel Project](#)

[Organizing Your Android App](#)

[Building and Running Your Android App](#)

[Best Practices Using Android/Kotlin in Bazel](#)

## Python

[Setting up Your Python Environment](#)

[Organizing Your Python Project](#)

[Building and Running Your Python App](#)

[Best Practices Using Python in Bazel](#)

## NodeJS/Typescript

[Aspect Build](#)

[Setting up your NodeJS/Typescript Environment](#)

[Organizing Your NodeJS/Typescript Project](#)

[Building and Running Your NodeJS/Typescript App](#)

[Best Practices Using NodeJS/Typescript in Bazel](#)

## Golang

[Setting up Your Golang Environment](#)

[Organizing Your Golang Project](#)

[Building and Running Your Golang App](#)

[Best Practices Using Golang in Bazel](#)

## iOS

[Setting up Your iOS Environment](#)

[Organizing Your iOS Project](#)

[Building and Running Your iOS app](#)

[External Dependencies](#)

[Using iOS Best Practices in Bazel](#)

## Conclusion

## Recommended Readings

## **7. Streamlining Development Workflow**

[Introduction](#)

[Structure](#)

[Code Contribution Workflows](#)

[Feature Branching](#)



[GitFlow](#)

[Trunk-based Development](#)

[Setting Up Continuous Integration with Bazel](#)

[Enabling a Sort of Local CI with Bazel](#)

[CI Worker Set up Models](#)

[Ephemeral Workers](#)

[Single Stateful Worker](#)

[Multiple Stateful Workers](#)

[Hot-pool of Workers](#)

[Sharded Worker Sets](#)

[Remote Build Execution](#)

[Conclusions About CI Worker Models](#)

[Managing Code Quality Tools](#)

[Formatting](#)

[Linting and Static Code Analysis](#)

[Conclusion](#)

[Recommended Reading](#)

## **8. Structuring Monorepos for Success**

[Introduction](#)

[Structure](#)

[Designing an Effective Monorepo Layout](#)

[Directory Structure Best Practices](#)

[Code Sharing and Reusability](#)

[Testing Strategies](#)

[Centralized Configuration Management](#)

[Refactoring and Code Maintenance](#)

[Security Considerations](#)

[Common Pitfalls and Ways to Avoid Them](#)

[Organizing Code into Packages and Modules](#)

[Naming Conventions for Packages and Modules](#)

[Conclusion](#)

[Recommended Reading](#)

## **9. Managing Large Codebases and Scale**

[Introduction](#)

[Structure](#)

[Dealing with Large Monorepo Codebases](#)

[Managing Internal and External Dependencies](#)

[Integration of Code from Other Repositories](#)

[Handling Third-Party Libraries](#)

[Advanced Modularization Strategies](#)

[Code Sharing and Reuse](#)

[Efficient Code Organization and Readability](#)

[Managing Inter-Module Dependencies](#)

[Advanced Strategies for Collaborative Environment Management](#)

[Refining Branching, Merging, and Code Review Practices](#)

[Minimizing Merge Conflicts and Build Breakages](#)

[Navigating Common Challenges](#)

[Upkeeping Bazel Build Configurations](#)

[Debugging Build Issues](#)

[Best Practices and Common Pitfalls](#)

[Performance Optimization for Monorepo Builds](#)

[Setting Up and Configuring RBE for Large-Scale Monorepos](#)

[Strategies for Cache Management and Sharing](#)

[Utilizing Bazel's Profiling Tools to Identify Bottlenecks](#)

[Analyzing Build Performance Data to Pinpoint Inefficient Patterns and Configurations](#)

[Advanced Caching Techniques](#)

[Parallelism and Resource Management](#)

[Dynamic Build Graph Optimization](#)

[Developing Custom-Build Rules for Performance-Critical Scenarios](#)

[Optimizing Existing Build Rules for More Efficient Execution](#)

[Performance Optimization for Monorepo Builds](#)

[Writing Efficient Starlark Code](#)

[Profiling and Optimizing Starlark Scripts](#)

[\*Structuring Monorepos in Version Control Systems\*](#)  
[\*Managing Source Code Changes\*](#)

[Conclusion](#)

[Recommended Reading](#)

## **10. Building and Deploying Services**

[Introduction](#)

[Structure](#)

[Optimizing Container Images Builds](#)

[\*Fine-grained Targets\*](#)

[\*Use of OCI Images\*](#)

[\*Layered Approach\*](#)

[\*OCI Image Building and Exporting\*](#)

[\*Parallelization and Caching\*](#)

[\*Running the Example\*](#)

[API Dependency Management in Microservices](#)

[\*Managing Transitive Dependencies\*](#)

[\*API Versioning Strategies\*](#)

[\*Automated Dependency Updates\*](#)

[Software Configuration Management](#)

[Orchestrating Microservices in a Monorepo](#)

[\*Advanced Microservice Orchestration Techniques\*](#)

[\*Feature Toggling\*](#)

[\*Monitoring and Scaling Microservices\*](#)

[Conclusion](#)

[Recommended Reading](#)

## **11. Monitoring and Debugging Bazel**

[Introduction](#)

[Structure](#)

[Monitoring Bazel Performance](#)

[\*Interpreting Profiling Data for Performance Bottlenecks\*](#)

[\*Utilizing Custom Scripts to Parse and Analyze Profile\*](#)

[\*Data\*](#)

[\*Utilizing Command Line Tools to Analyze Profile Data\*](#)

[Visualizing Profiling Data](#)  
[Understanding and Utilizing Bazel's BEP](#)  
[Converting BEP Output to Prometheus-friendly Format](#)  
[Developing Automated Tests for Performance  
Regressions](#)  
[Implementing Benchmarks and Performance Baselines](#)  
[Continuous Monitoring of Performance Metrics](#)  
[Advanced Usage of Remote Caching and Execution](#)  
[Monitoring Remote Cache Hit Rates](#)  
[Diagnosing Cache Misses and Inefficiencies](#)  
[Optimizing Remote Execution Performance](#)  
[Debugging Techniques for Bazel Rules](#)  
[Step-by-step Approach for Debugging Custom Bazel  
Rules](#)  
[Utilizing Starlark's Debugging Capabilities](#)  
[Best Practices for Logging and Error Handling in Rule  
Development](#)  
[Performance Tuning for Large-scale Monorepos](#)  
[Divide and Conquer](#)  
[Advanced Configuration Settings for Improved  
Scalability](#)  
[Case Studies on Performance Improvements in  
Complex Projects](#)  
[Conclusion](#)  
[Recommended Reading](#)

## **12. Advanced Bazel Concepts**

[Introduction](#)  
[Structure](#)  
[Comprehensive Exploration of Caching](#)  
[Publishing Your Own Bazel Rules](#)  
[Migrating a Maven Project to Bazel](#)  
[Hermeticity](#)  
[Bazel Hot Reload](#)  
[Building Custom Toolchains](#)

[Aspects](#)

[Aliases](#)

[Exploring Experimental Bazel Features](#)

[Conclusion](#)

[Recommended Reading](#)

## **13. Case Studies and Real-World Examples**

[Introduction](#)

[Structure](#)

[Case Study 1: Building a Full Stack Digital Service](#)

[\*Motivation for Adopting Bazel\*](#)

[\*Implementation Strategy\*](#)

[\*Challenges and Solutions\*](#)

[\*Results and Impact\*](#)

[\*Lessons Learned\*](#)

[\*Future Plans and Considerations\*](#)

[\*Outcome\*](#)

[Case Study 2: Building a Serverless Service Platform](#)

[\*Motivation for Adopting Bazel\*](#)

[\*Implementation Strategy\*](#)

[\*Challenges and Solutions\*](#)

[\*Results and Impact\*](#)

[\*Lessons Learned\*](#)

[\*Future Plans and Considerations\*](#)

[\*Outcome\*](#)

[Case Study 3: Using Bazel in a Developer Hub](#)

[\*Motivation for Adopting Bazel\*](#)

[\*Implementation Strategy\*](#)

[\*Challenges and Solutions\*](#)

[\*Results and Impact\*](#)

[\*Lessons Learned\*](#)

[\*Future Plans and Considerations\*](#)

[\*Outcome\*](#)

[Conclusion](#)

## **14. Future Trends and Considerations**

Introduction

Structure

Evolving Practices in Monorepo Development

*Integrating AI and Machine Learning in Bazel Builds*

*Enhanced Remote Caching and Execution Strategies*

*Advanced Dependency Management Techniques*

*Security Enhancements in Monorepo Infrastructure*

*Fostering Collaboration through Enhanced Code*

*Review and Integration Practices*

*Sustainable Development Practices in Monorepo*

*Management*

The Road Ahead for Bazel and Monorepo Tooling

*Bazel's Modular Dependency Management: Bzlmod*

*Planned Features for Bazel 7 and Beyond*

*The Future Path: Bazel 8 and 9*

*Implications for Monorepo Tooling*

*Considerations for Future Development*

Anticipating Challenges and Adapting Strategies

*Migration to Bzlmod*

*Enhanced Performance and Caching*

*Expanded Language and Platform Support*

*Robust Extension Model*

*Adapting to Continuous Updates*

How to Migrate Existing Projects from Bazel 6

Recommended Readings

Conclusion

## **APPENDIX A Bazel Cheat Sheet**

Quick Reference Guide to Bazel Terminology

Quick Reference Guide to Bazel Commands

*Bazel Command Structure*

*Core Commands*

*Advanced Commands*

## **APPENDIX B Additional Resources**

Recommended Books

Online Communities

*Understanding Bazel and Monorepo Communities*

*Participating in the Community*

*Contributing to the Community*

**Index**

# **CHAPTER 1**

## **Introduction**

### **Introduction**

In this chapter, we will delve into the world of Monorepos and their significance in modern software development. We will start by gaining a comprehensive understanding of the Monorepo approach and how it ushers in a true era of continuous integration, breaking down the traditional silos that have plagued Polyrepo structures. We will explore the drawbacks of Polyrepo systems and contrast them with the numerous benefits that a Monorepo can offer. To appreciate the roots of Monorepos, we will take a brief journey through their historical evolution. Along the way, we will address common fears and challenges associated with implementing a Monorepo and discuss the transformative power of Bazel in Monorepo development. Delving into the history, features, and design logic of Bazel, we will help you grasp why it is such a valuable tool in this context. To guide your decision-making process, we will also discuss when to opt for a Monorepo and when it might not be the right fit, as well as when to choose Bazel and when it might not be the best choice. Lastly, we will provide a list of recommended readings to further enrich your knowledge in this exciting realm of software development.

### **Structure**

In this chapter, the following topics will be covered:

- Understanding the Monorepo Approach
- The Power of Bazel in Monorepo Development

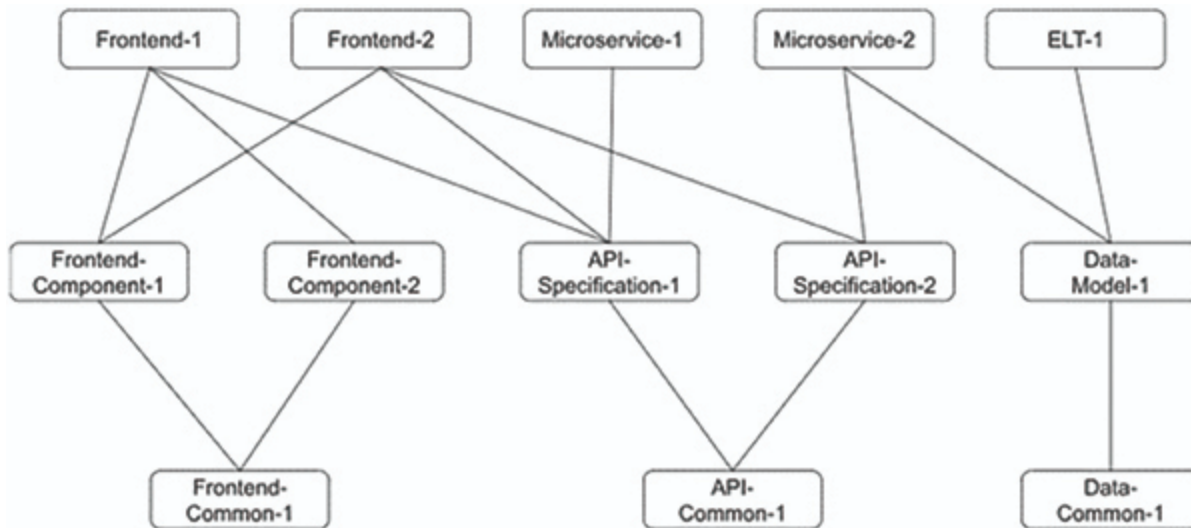


- When to Use Monorepo and When Not to Use it
- Bazel Features

## Understanding the Monorepo Approach

A monorepository, often referred to as a Monorepo, is an approach to software development that involves managing distinct “code parts” with well-defined relationships within a single repository. These “code parts” can vary in granularity, from entire applications to intermediate-sized functional units such as business operations, or even individual architectural components.

While these sections may have dependencies, they typically maintain logical autonomy and are overseen by different teams. Each has clear dependencies on others, as well as on external libraries, resources, and similar elements.

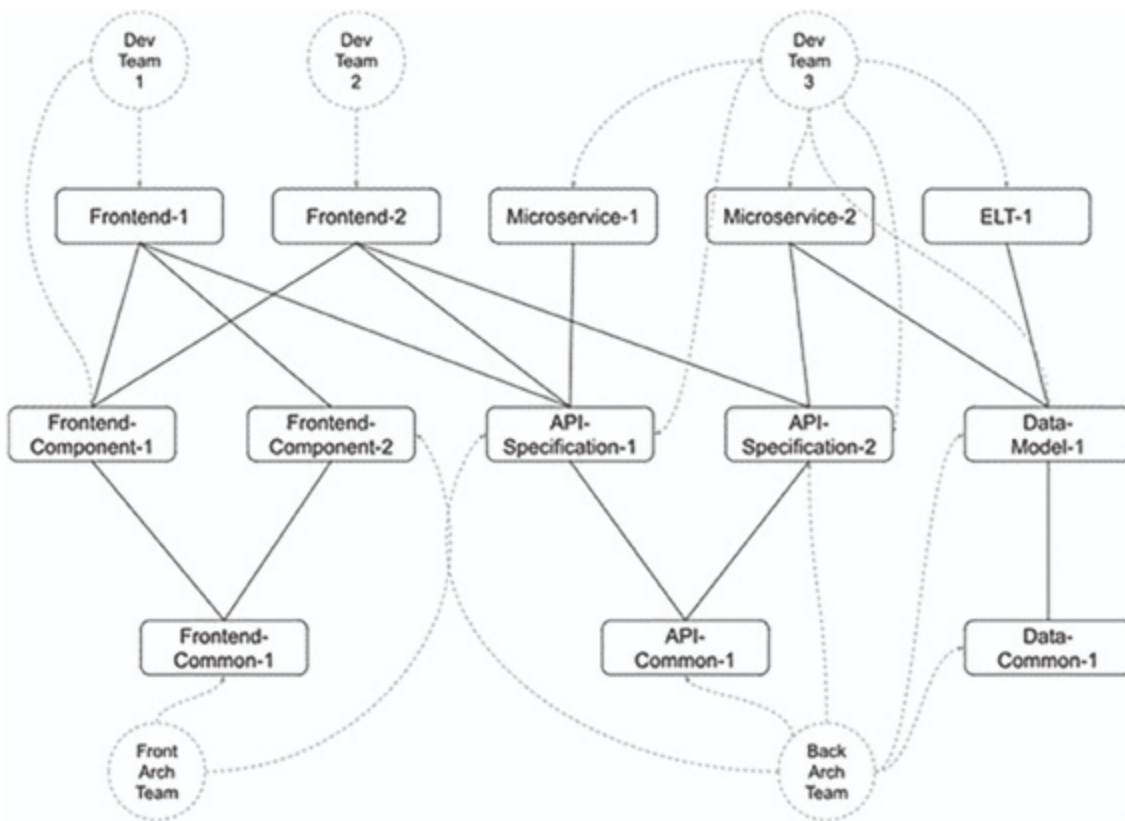


**Figure 1.1:** Application projects and its dependencies with common shared projects

Monorepos are not a silver bullet; there is no universal solution in software development. Yet, by the end of this book, you should understand the potential benefits of a

monorepo, the challenges you might face, and whether it aligns with your organization's needs.

In a monorepo setup, multiple projects are housed within one repository. These projects can depend on each other, enabling code sharing. When you make changes, there is no need to rebuild or retest every project in the monorepo. Instead, focus on rebuilding and retesting only the projects directly affected by your changes. This approach is often called *“incremental builds”*.



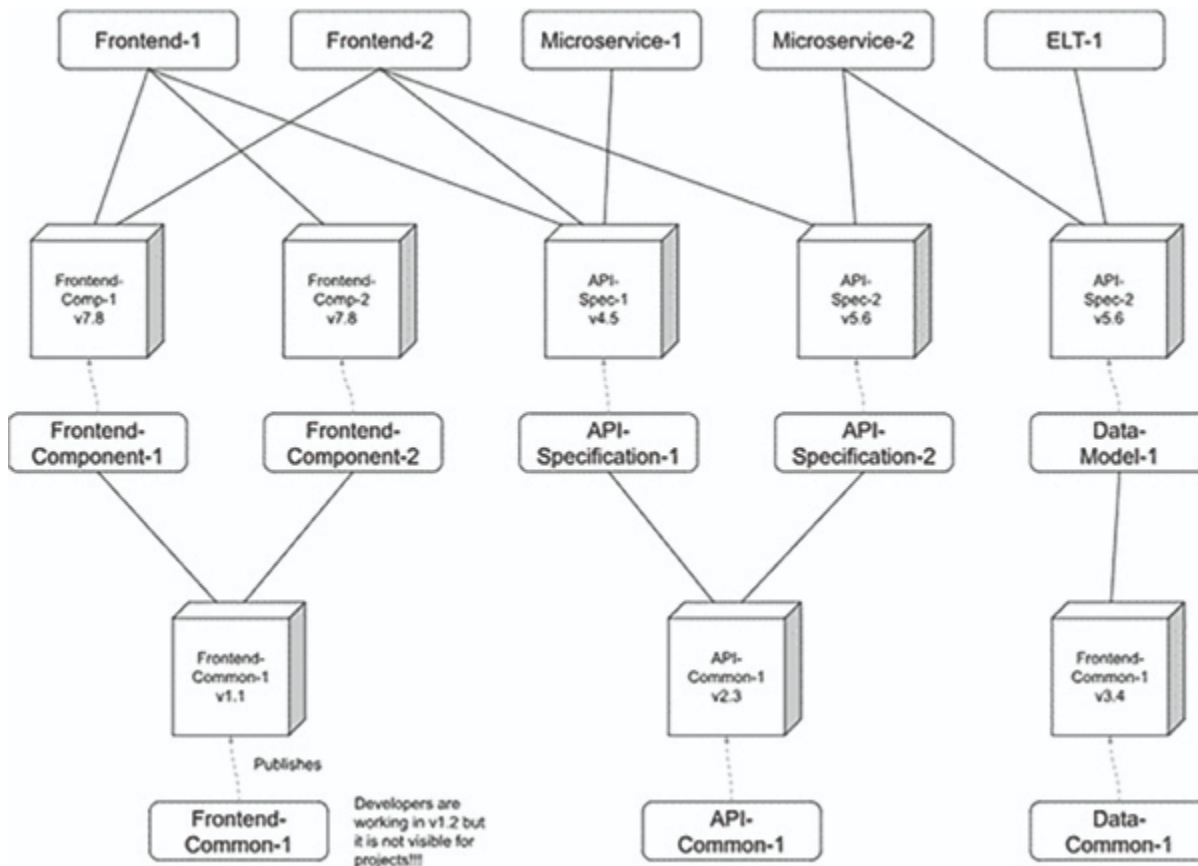
**Figure 1.2:** Teams maintain one or more interdependent projects

Incremental builds grant teams within the monorepo a degree of independence. If two projects do not depend on each other, they remain shielded from one another's impacts. One team can advance with development, testing, PR merges into the main branch, and other related activities without having to run code from the second team. While the second team might face challenges like unstable tests,

poorly typed code, or broken code and tests, these issues will not disrupt the first team's progress.

## Welcome to the True Continuous Integration

A Polyrepo embodies the traditional approach to application development: each team, application, or project maintains its own repository. Generally, each repository yields one build artifact and adheres to a linear build pipeline. Polyrepo configurations often depend on previously published artifacts. Ideally, every repository integrates continuously (CI) with the latest version of its dependencies, facilitating smooth integration.



**Figure 1.3:** At a Polyrepo, components are shared through published artifacts

In a Monorepo, all code changes (commits) belonging to the same code contribution (a complete and accepted feature) is viewed as a pull request, or a merge request in case of using the GIT version control system. These requests must synchronize with the most recent commit in the main branch, but there are similar concepts in other tools. This synchronization ensures that all internal dependencies match the current source code version, and external dependencies align with the newest globally defined version. This rigorous procedure ensures that code contributions integrate fully with the latest changes in the main branch.

## **Drawbacks of Polyrepo**

The Polyrepo approach has gained traction in the industry largely because of a pivotal factor: team autonomy. Teams desire the liberty to choose their libraries, decide the timing of app or library deployments, and determine who can contribute to or use their code.

However, the challenge with Polyrepos is that autonomy often stems from isolation, which can impede collaboration. In contrast, a monorepo achieves autonomy by endorsing detailed project management, supporting incremental builds (as discussed earlier), and implementing pull-request continuous integration with both manual and automated approvals. These approvals might include tools like **CODEOWNERS** files, which assign specific individuals or groups as the custodians of the code in a repository. These custodians are involved during code contributions, especially when changes affect their projects.

In Polyrepo configurations, shared code typically requires a separate repository. Setting this up means configuring the necessary tools, establishing a CI environment, adding authorized contributors to the repository, and creating

mechanisms for package publishing to allow dependencies from other repositories. Additionally, harmonizing conflicting versions of third-party libraries across repositories can be challenging.

Setting up a shared repository within a Polyrepo can be time-intensive. As a result, many teams might decide to create their own versions of common services and components within their respective repositories. While this might be expedient initially, it escalates the effort needed for maintenance, security, and quality control over time as these elements evolve.

Once a shared repository is operational, initiating cross-repository changes to shared libraries becomes an arduous task. Developers have to adjust their environments to enact these alterations across various repositories, each with its unique revision history. This requires significant coordination concerning versioning and package releases.

In a Polyrepo setting, teams often use their unique command sets for tasks such as testing, building, serving, linting, deploying, and more. This inconsistency adds cognitive load, as team members must recall the appropriate commands for different projects.

## **Benefits of Monorepo**

Initiating new projects is streamlined to creating a folder paired with a project descriptor. These projects readily tap into the existing CI setup, negating the need to release versioned packages when all users are housed in the same repository.

In this cohesive setup, every component interacts seamlessly with each commit. As a result, modifications across various projects converge into a single atomic commit—changes are either fully implemented or not at all.

This methodology dispels the notion of breaking changes as any concerns are addressed within the same commit.

Compatibility concerns stemming from projects using clashing versions of third-party libraries are eliminated. All such dependencies are precisely cataloged in a centralized, shared third-party dependency definition file. The threat of a library becoming obsolete due to team unavailability is reduced. Any team can modify a library version (subject to certain automated checks), and the CI system guarantees that all projects stay operational by running extensive unit tests—a safeguard ensured by unit testing.

Moreover, a monorepo fosters developer flexibility among teams. It instills a uniform approach to building and testing applications, even when they are developed using varied tools and technologies. Developers can contribute to projects overseen by different teams, ensuring their changes' safety and compatibility.

## **A Bit of History**

In version control systems' early days, like CVS (Concurrent Versions System) and SVN (Subversion), separate repositories for individual software projects or components were the norm. This methodology was apt when software initiatives were largely compact and independent.

The rise of distributed version control systems, notably Git, in the mid-2000s, ushered in greater flexibility for developers. Git's decentralized architecture empowered them to operate across multiple branches and repositories, simplifying collaboration on expansive codebases.

Google often receives credit for advancing the monorepo paradigm. Internally, they adopted an extensive monorepo around 2008, encompassing nearly all their software projects' source code. This approach afforded Google more streamlined management of its vast codebase through an

internal tool named Piper. The efficiencies, such as enhanced code sharing, unified dependency management, and standardized tooling, spotlighted by Google's Monorepo, sparked intrigue in the wider developer community.

Subsequently, tech behemoths like Facebook, Twitter, and Microsoft embraced Monorepos. They discerned the perks of a consolidated repository to oversee their extensive codebases that spanned varied languages and platforms. To make Monorepos more palatable for entities beyond these giants, a slew of open-source tools and frameworks emerged. These include Gradle Build Tool (by Gradle, Inc), Lage (by Microsoft), Lerna, Nx (by Nrwl), Pants (courtesy of the Pants Build community), Rush (by Microsoft), and Turborepo (by Vercel).

The Monorepo methodology has steadily carved a niche within the developer community. Numerous entities, from start-ups to conglomerates, have transitioned to Monorepos, optimizing their development workflows and bolstering code sharing. The Monorepo discourse remains dynamic, with ongoing deliberations on best practices, tool enhancements, and strategies to navigate the intricacies of vast Monorepos.

## **Typical Fears about Monorepos**

In this section, we will address some of the common apprehensions and concerns that software developers often have when considering the adoption of Monorepos, shedding light on the practical solutions and benefits that can alleviate these typical fears.

### **Scalability Concerns with CI Cycles**

The Polyrepo approach often results in extended CI cycles, requiring complete rebuilds and retests for each commit—a decidedly non-scalable method, especially with numerous projects. However, adopting a monorepo paired with

incremental build tools lets you rebuild and test only the segments affected by changes. While this method offers improved scalability, it is not a magic fix. For large repositories comprising numerous extensive applications, you might experience delays when making changes impacting multiple dependencies. While such scenarios are not frequent, they may necessitate leveraging multiple machines, also referred to as remote workers, to streamline the CI process.

### **Potential Git Limitations**

There is a valid concern that standard Git tools may struggle with repositories comprising millions of files. Yet, it is vital to recognize that most Monorepos do not house thousands of applications. They typically comprise a handful of applications by a singular organization, housing thousands of files with millions of lines of code. Under these conditions, most tools can efficiently handle the workload.

### **Monolithic Deployment Concerns**

A prevalent misconception is equating a Monorepo with forced simultaneous binary releases, based on the thought that “Monoliths are disadvantageous”. The source of this code and deployment considerations are two separate entities. Ideally, CI/CD best practices involve constructing and storing artifacts during the CI phase, deploying these stored items across various environments during deployment. Thus, accessing a repository should not be a requirement during deployment. Emphasizing, a Monorepo does not equate to a monolithic structure. Monorepos enable easy code sharing and inter-project refactoring, simplifying the process of developing libraries, microservices, and micro-frontends. This setup can offer greater deployment flexibility.

### **Unauthorized Code Changes**

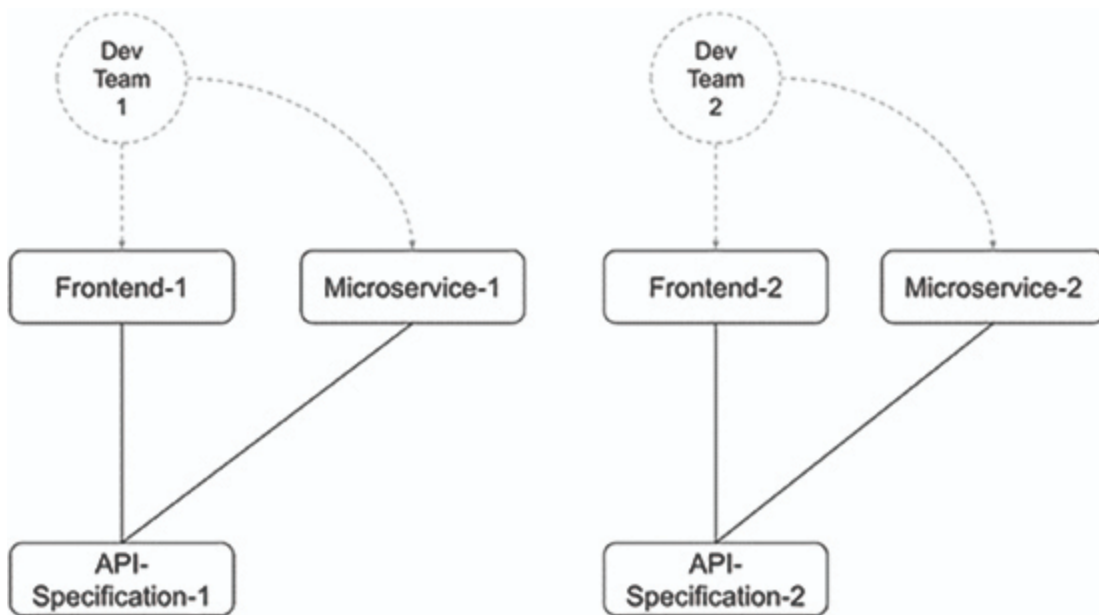


The fear that anyone can modify code within a Monorepo, potentially jeopardizing a team's application without their knowledge, originates from an over-reliance on repository-level permissions. Several tools enable folder-level ownership settings. For instance, GitHub and other Git platforms provide a feature termed **CODEOWNERS**, allowing structured file creation as follows:

```
domains/domain-1/* @john
tools/scripts/* @peter
```

With the specified configuration, a pull request modifying **domain-1** necessitates John's approval. If it is solely about **scripts**, then Peter must give the nod. However, for pull requests touching both **domain-1** and **scripts**, approvals from both John and Peter are essential.

This system enhances control over code ownership. To further understand its efficacy, consider the following example:



**Figure 1.4:** Project visibility in Monorepo helps to manage sharing between teams

In this example, the Dev Team 1 shares an API definition project between its frontend and backend projects. This API

definition project is intentionally private, as both teams aim to preclude other teams from leveraging it. This choice stems from a desire to evade undue inter-team dependencies. If **Dev Team 2** wants to use this API definition, it would obligate **Dev Team 1** to accommodate additional requirements during modifications to the shared library.

In a Polyrepo environment, nothing deters teams from adding “*api-definitions*” to their dependency file (**package.json**, **pom.xml**, and so on). The dilemma? Dev Team 1 remains uninformed as these additions occur in an external repository. Contrastingly, most Monorepo tools, Bazel included, allow for precise library visibility delineation.

**Monorepo is going to turn into a project’s spaghetti** (dependency entanglement), making it challenging to comprehend and maintain applications

There is a prevalent misapprehension that Monorepos inevitably lead to entangled project structures. It is true that in numerous repositories, any given file could technically import another. While structured approaches do emerge through code reviews, these can deteriorate over time, resulting in intricate dependency networks.

For clarity, envision a medium-scale project. Chart out its component dependencies. Upon juxtaposing this with the actual repository, you may discern several surprising interconnections.

However, Monorepos empower developers to construct libraries with clear public APIs. The streamlined library creation process encourages more frequent library usage. Hence, a typical app often gets segmented into multiple libraries, interacting solely via their public APIs.

Monorepos endorse software modularization and detailed granularity. Their structure simplifies both the initiation of new projects and the refactoring of existing ones, particularly when expansion necessitates splitting.

## Typical Challenges in Implementing a Monorepo

Implementing a Monorepo is not a smooth journey. Monorepos have their own challenges, and these are the most common ones.

### **Onboard development teams to new methods and tooling**

Introducing new methods and tools typically comes with a learning curve. Developers must invest time and effort to grasp how these tools function and how they integrate into their workflow. This learning curve can impede productivity during the initial stages. People, in general, are often resistant to change, and developers are no exception. Many are accustomed to their current tools and methods, and introducing new ones can encounter resistance, potentially reducing morale and efficiency.

Integrating new tools and approaches with existing processes can be challenging. A seamless transition might necessitate adjustments to established workflows, and these changes might face opposition from team members set in their ways. Onboarding teams to fresh techniques and tools often demands more resources, like time for training and expertise to oversee the transition. Scarce resources can hinder the onboarding process.

Transitioning to novel methods and tools might interrupt ongoing projects. There might be concerns about potential delays or errors during the induction period. Compatibility challenges between new tools and current systems can pose significant obstacles. It is vital to ensure that new tools mesh well with the existing tech stack. Veteran developers, having profound knowledge of the prevalent tools, may oppose changes they deem superfluous or intrusive.

If the advantages of embracing new tools are not lucidly conveyed or do not offer evident value, team members might hesitate to adopt them. It is pivotal to offer ample training and sustained support for a smooth transition. Lack of adequate training or support can culminate in frustration and reluctance.

### **Choose the good “Go to Monorepo” strategy**

Start by evaluating the scope of your projects. Decide which projects will pioneer the Monorepo approach and establish a roadmap for the rest. Monorepos thrive when handling multiple interconnected projects or components benefiting from shared code. Ensure your projects genuinely demand the integration and code sharing that monorepos provide. When selecting projects, account for your team’s structure and dynamics. Monorepos excel for teams on intertwined projects where collaboration and shared code are paramount.

For every project earmarked for migration, scrutinize your development workflow and procedures. Ascertain if a monorepo aligns with your current CI/CD pipelines and tools. The chosen strategy should enhance your workflow, not impede it. Also, gauge your codebase’s complexity. Projects with intricate interdependencies or significant reliance on shared code can find management simplified in a Monorepo. However, if your codebase is monolithic or made of broadly structured modules, consider refactoring into more detailed components.

Another vital aspect is versioning requirements. Monorepos typically necessitate a versioning system capable of managing distinct components separately. Determine if your team requires detailed version control for various code sections. The Monorepo structure’s design critically influences project scalability. As the codebase grows, Monorepos risk becoming cumbersome. Choose a Monorepo

approach that can handle future expansion without significant performance hindrances. Be prepared for continuous refactoring during the Monorepo's lifespan to optimize its structure.

Examine the tooling and support available for your strategy. Tools and frameworks tailored for Monorepos can simplify both the transition and ongoing management. Explore the presence of these tools within your organization's ecosystem.

Transitioning to a Monorepo or a new methodology necessitates a shift in team culture and practices. Ensure your team embraces this shift, and you have laid out a robust change management plan. A Monorepo promotes long-term code consistency across projects. Ensure your strategy mirrors your organization's long-term vision.

Lastly, remain receptive to experimentation and iterative approaches. The choice between Monorepo and Polyrepo isn't set in stone. Experiment with one strategy and pivot to another if it aligns more with your evolving requirements.

### **Introduce Trunk-based Development**

Monorepos typically do not synergize well with long-lived feature branches. As a result, you might need to adopt a trunk-based development variation and regularly employ `git rebase`. Transitioning to this style can be challenging for some teams as it introduces new practices, like feature toggles.

Trunk-based development is a strategy focusing on simplicity, collaboration, and swift delivery. At its heart, it pushes for continuous code integration into a shared branch, known as the `trunk` or `main` branch. This method contrasts with long-lived feature branches, where developers work separately for extended durations before merging into the primary codebase.

With trunk-based development, code changes are integrated into the `main` branch frequently, sometimes multiple times daily. This keeps the `main` branch updated and stable. Developers are urged to commit small, incremental changes, which eases code reviews, minimizes integration conflicts, and facilitates swift issue detection and resolution. Essential to this approach is automated testing. Automated tests run persistently, offering immediate feedback on code quality and correctness. Feature toggles or flags enable selective activation or deactivation of certain code features, allowing incomplete or experimental features to remain hidden until ready for release. This strategy enhances collaboration as developers consistently work on the `main` branch, with the absence of long-lived branches spurring more frequent code reviews.

Benefits of trunk-based development include:

- Continuous delivery of new features and fixes.
- Reduced software product time-to-market.
- Minimized merge conflicts due to frequent, smaller merges.
- Early issue detection and resolution, curtailing defect-fixing costs.
- Elevated collaboration from a unified codebase.
- Improved code quality and sustainability from concise commits and continuous testing.

However, implementing trunk-based development might mean:

- Transitioning from feature-branch workflows, necessitating culture and practice shifts, including adopting feature toggles.
- Handling complexities as developers use toggles to hide unfinished features.

- Investing significantly in automated testing infrastructure since continuous testing demands ample resources.

Trunk-based development is considered to promote superior code quality and development pace, regardless of repo size. Still, it requires careful consideration. For instance, if you are on a feature branch and the trunk branch (also known as “**master**” or “**main**”) gets new commits, rebasing your feature branch onto the **main** can position your work post those commits.

As for git **rebase**, it is a Git command allowing changes from one branch to be “reapplied” onto another. It is particularly useful in refining commit history and integrating branch changes. Rebasing maintains a linear commit history, unlike merging. While rebasing might encounter conflicts, they can be resolved before the process continues. Crucially, rebasing alters commit history. Pushed commits that are later rebased can create divergent histories, leading to potential confusion. It is wise to **rebase** only unpushed commits to shared repositories.

### **Mind the gap between Monorepo and your current tooling ecosystem**

Adopting a Monorepo in an organization can yield significant advantages, but it is not without hurdles, especially when incorporating existing tools not tailored for Monorepos. A primary concern is ensuring tools, like static code analyzers (SonarQube, Find Bugs, and ESLint) or security linters (Fortify, Checkmarks, Veracode), mesh with the Monorepo structure. Many classic tools presume single-repository settings, making their integration into Monorepos potentially intricate and lengthy. Bringing in non-Monorepo tools often means custom development or outsourcing, leading to added time, cost, and potential vulnerabilities.

Reconfiguring CI/CD pipelines for Monorepos is essential. This may involve unique scripts or CI/CD tools adept at managing a Monorepo's multifaceted build and deployment needs. Tools like Bazel, for instance, need a shared remote cache for all CI workers to bypass full rebuilds and effective repository management across CI cycles. Automated tests must be robust, ensuring a change in one component does not inadvertently disrupt another. Dependency management, especially across various languages or platforms, requires careful handling, and robust version control is crucial. As Monorepos expand, so do concerns over performance. Entire codebase operations can become time-consuming, demanding swift build and deployment techniques to counteract potential lags. Simultaneous issues across various components can complicate monitoring and debugging, necessitating proficient diagnostic tools and practices.

A Monorepo's introduction can mandate a shift in development culture and practices. Teams may need to embrace new methods like trunk-based development or feature toggles, which might encounter resistance. Promoting effective team collaboration within the monorepo setting can be daunting, especially for those accustomed to secluded workflows. Therefore, clear code-sharing and ownership protocols are indispensable.

For a successful transition to Monorepos, organizations should meticulously plan their shift. This entails evaluating tool compatibility, making necessary tool adjustments or swaps, and ensuring teams have ample training and support. The long-term benefits of enhanced code sharing, efficient collaboration, and streamlined dependency management can make navigating these hurdles well worth the effort.

## **Deal with large-scale changes**

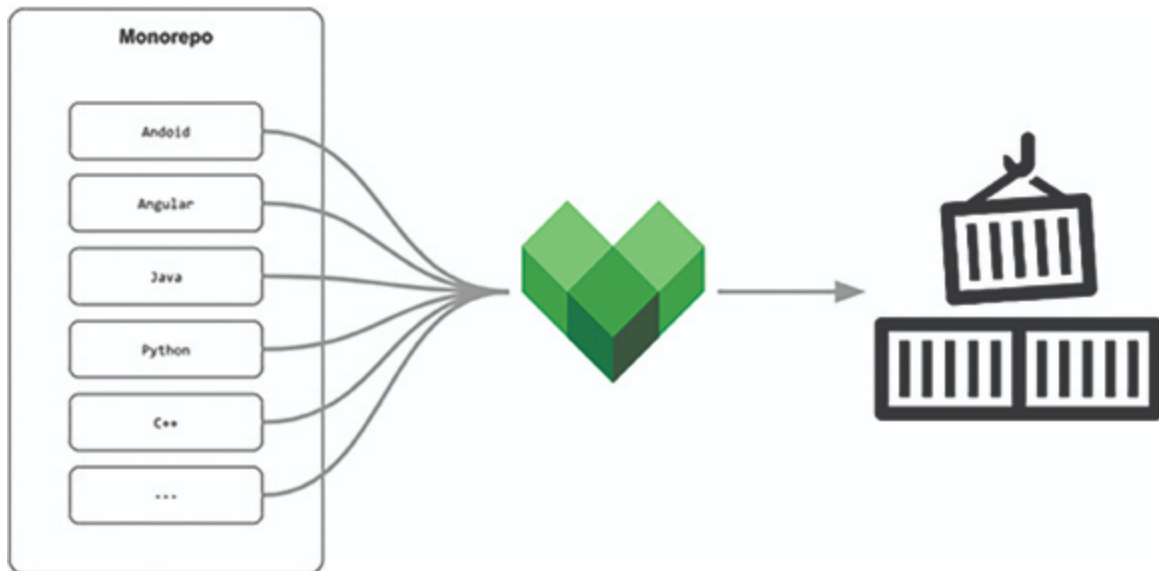


Monorepos offer a streamlined mechanism for executing extensive changes in specific contexts. When refactoring various applications made of multiple libraries, a Monorepo ensures that every component operates seamlessly before finalizing the alteration.

Yet, Monorepos demand a nuanced strategy for sweeping changes, occasionally complicating certain processes. For instance, modifying a shared library affects every application dependent on it. If such a change is drastic and lacks automation, it necessitates backward-compatible adjustments. This means introducing dual versions of parameters, methods, classes, or packages and directing users to transition from the legacy version to the updated one.

## **The Power of Bazel in Monorepo Development**

Monorepo development, which involves managing multiple projects or components within one version control repository, has earned substantial traction recently. Its capacity to enhance collaboration, promote code sharing, and refine dependency management sets it apart. Yet, with the growth and intricacy of monorepos, overseeing the build process across multiple programming languages and platforms can become overwhelming. Enter Bazel, the build tool tailored for such challenges.



**Figure 1.5:** *Bazel in monorepo development*

Bazel's prowess is particularly evident in the realm of Monorepo development, especially when juggling multiple programming languages and platforms. Key attributes, such as its language-neutral approach, hermeticity, adept dependency management, comprehensive platform support, and emphasis on incremental builds and scalability, render it a top pick. Organizations aiming to tap into Monorepo advantages without compromising on build efficiency and dependability will find Bazel indispensable. In harnessing Bazel, development squads can sustain a unified and efficient workflow, supporting remote execution, even amidst the most multifaceted code environments.

## **A Bit of History**

Bazel's origins trace back to Google's in-house tool, Blaze, developed in the mid-2000s. Created to tackle the mounting scalability and efficiency challenges of a rapidly expanding Google, Blaze introduced groundbreaking features. Notable among them were hermetic builds, ensuring build isolation from host environments, and a distributed caching mechanism, minimizing redundant build tasks.

In 2015, Google offered a slice of Blaze to the public, branding it “Bazel.” This open-source venture aimed to share Google’s scalable and reproducible build proficiency with the wider developer community.

The software realm soon noticed Bazel’s robust build, testing capabilities, and multilingual support. Recognized projects like Kubernetes and TensorFlow integrated Bazel into their build and test operations. Bazel’s developers kept broadening its linguistic scope, enhancing its appeal for polyglot projects. With contributions pouring in from diverse sectors, while Google remained a primary contributor, Bazel morphed into a collaborative open-source initiative.

In CI/CD pipelines, Bazel’s merits shine. Its commitment to hermetic, reproducible code building and testing complements contemporary software development ethos. Industry giants like Uber, Dropbox, and Pinterest have incorporated Bazel into their tech stack, signifying its pivotal role in sophisticated software infrastructures. With Bazel continually adapting to the dynamic software milieu, Google and its community persistently roll out enhancements, refining its capabilities, and expanding its versatility.

## **Bazel Features**

Bazel is a distinguished build system created especially for Monorepos. It champions the principles of efficiency, scalability, and reproducibility in software development. Notably, it finds its niche in dealing with expansive codebases and intricate projects. Here are the core features that define Bazel’s proficiency in building and overseeing software ventures:

- **Local Computation Caching:** This refers to Bazel’s adeptness in preserving and repurposing the outcomes of file and task operations. For individuals operating on the same computer, there is no necessity to redo builds

or tests for the same tasks. When a specific command is given, it gets executed initially, caches the result, and subsequent invocations utilize this cached outcome.

- **Local Task Orchestration:** Bazel is equipped to orchestrate tasks in a sequential as well as concurrent manner.
- **Distributed Computation Caching:** This feature ensures that cached results are spread across varied environments. In essence, within an entire organization, including the CI agents, redundant builds or tests for similar components become obsolete.
- **Distributed Task Execution:** Bazel can disseminate a command over several machines, all while preserving the feel of operating it on a singular machine. It is noteworthy that Bazel's application here is incredibly sophisticated, capable of managing repositories with billions of code lines. Yet, setting it up might pose certain complexities.
- **Transparent Remote Execution:** Bazel boasts the ability to unobtrusively run commands across multiple systems during local development, a unique trait that distinguishes it from its counterparts.
- **Impact Analysis:** By assessing the potential ripple effect of changes, it permits the targeted execution of build and test processes for the impacted entities. Bazel might not natively offer this, but tools like target-determinator exploit Bazel's query language to fill this void.
- **Workspace Analysis:** Bazel has the prowess to fathom the project structure within the workspace sans extra configurations. Though Bazel expects developers to craft BUILD files manually, several firms have pioneered tools to auto-generate these files by scrutinizing the workspace content.

- **Dependency Visualization:** With Bazel, one can visualize project/task interdependencies through graphical illustrations. This interface is interactive, bolstered by Bazel's proprietary query language, ensuring precise information sifting.
- **Code Segmentation:** Bazel facilitates easy sharing of code segments. Any directory can be flagged as a project and shared. This sharing is anchored by Bazel's build protocols, promising a seamless developer interaction.
- **Uniform Tooling:** Regardless of the tech stack—be it diverse JavaScript frameworks or languages like Go, Rust, and Java—Bazel guarantees a consistent experience. Its extensible nature, empowered by build rules, functions much like modules catering to varied tech landscapes.
- **Code Generation:** While tools like NX or Pants natively support code generation, Bazel, in contrast, taps into external generators, offering a broader scope.
- **Project Regulation and Visibility:** Bazel allows the crafting of rules to dictate dependency pathways within a repository. Developers have the liberty to earmark projects for their teams exclusively. Plus, tech-based categorization ensures no overlap between backend and frontend components. Bazel's visibility protocols ensure a clear demarcation between what is shared and what is private.

Wrapping up, Bazel stands tall as a formidable build system addressing the nuances of today's software development landscape. With its unwavering commitment to performance, reproducibility, and adaptability, it has won the favor of entities with sizable and multifaceted code repositories. Be it a Monorepo, a multi-language endeavor, or a distributed workforce, Bazel's arsenal promises to

elevate the development process, ensuring impeccable and swift builds.

## **Understanding the Logic Behind Bazel's Design**

Bazel is not just another build system; its foundational philosophy distinguishes it in a crowded ecosystem. To truly appreciate Bazel's uniqueness, it is pivotal to grasp the foundational principles steering its design, including:

- **Reproducibility:** A non-negotiable principle for Bazel. It understands the significance of consistent build results across varied settings. This is accomplished through rigorous dependency tracking and hermetic builds, ensuring they remain untouched by external influences.
- **Scalability:** Tailored for heft and complexity, Bazel is adept at managing voluminous code repositories. Conceived with Google's colossal repositories in mind, its intrinsic design and caching facilities seamlessly support projects sprawling across thousands of developers.
- **Incrementality:** At Bazel's heart lies the ethos of incremental builds. With every code modification, it meticulously recompiles and tests only those segments directly impacted, slashing build durations, especially for hefty codebases.
- **Polyglot Support:** In acknowledging the diverse linguistic tapestry of modern software development, Bazel natively accommodates an array of programming languages. Its adaptability extends to welcoming additional languages when the need arises.
- **Flexibility and Extensibility:** Bazel's build doctrines are articulated in Starlark, reminiscent of Python. This

design choice empowers developers to craft project-specific build rules. Moreover, its malleable nature welcomes third-party integrations, enriching its capabilities.

- **Hermeticity:** Bazel's allegiance to hermeticity ensures builds are insulated from host environments, relying solely on explicitly stated dependencies. This strategy amplifies the consistency, reliability, and repeatability of builds.
- **Parallelism:** Bazel's prowess in exploiting parallelism is unparalleled. Whether it is distributing tasks across multiple cores or spanning multiple machines, it optimizes build periods and boosts efficiency.
- **Dependency Analysis:** With sophisticated dependency scrutiny, Bazel meticulously discerns and traces dependencies. This precision is vital to pinpoint elements requiring rebuilds following changes, reinforcing its incremental build prowess.
- **Community and Collaboration:** Bazel is nurtured by a robust open-source community. A mosaic of contributors from myriad organizations fuels its evolution, ensuring it benefits from a plethora of insights and experiences.

To encapsulate, Bazel's approach is shaped by its unwavering commitment to core software development tenets like reproducibility, scalability, and incrementality. It is designed to cater to the demands of expansive, intricate, and multilingual software ventures without compromising on dependability and efficiency. Bazel's steadfastness to these ideals has cemented its position as an indispensable ally for developers and organizations, regardless of project size.

## [When to Use a Monorepo](#)

You want to opt for Monorepo if you identify the following needs:

- **Simplified Dependency Management:** If multiple projects share common dependencies, a Monorepo can simplify dependency management, ensuring that every project is using the same version of a shared library.
- **Atomic Changes Across Projects:** If there are changes that need to be made atomically across multiple projects, a Monorepo can make this easier, as you can modify multiple projects in a single commit.
- **Code Sharing and Reusability:** It becomes simpler to share code across projects in a Monorepo. This can encourage more modular and reusable code.
- **Easier Refactoring:** When you refactor shared code or libraries, it is easier to do in a Monorepo because you can instantly see and update all the places where it's used.
- **Unified Versioning:** With a Monorepo, you can have a single version number for all the projects. This makes it straightforward to understand which versions of different projects are compatible.
- **Improved Collaboration:** Developers can work across multiple projects without switching between different repositories. This can improve team collaboration, as it is easier to coordinate and understand the broader ecosystem.

## [When Not to Use a Monorepo](#)

Be careful choosing a Monorepo if you find the following scenarios:

- **You are unable to refactor applications:** As your codebase grows, build times and tooling can become slower, especially if not optimized for a Monorepo



structure. This can lead to longer test and integration times.

- **You do not have enough collaboration with other involved teams:** Having everything in one place can be overwhelming, especially for new developers who might not need to know about all parts of the system.
- **Tighter Coupling:** While not inherent to monorepos, there is a risk that projects can become more tightly coupled, making them harder to split apart in the future.
- **Security Concerns about reading all the available code:** In a Monorepo, everyone typically has access to all the code. This can be a security risk if certain projects have more sensitive code.
- **Vendor Tooling:** Some version control systems (like Git) might not perform well with extremely large repositories, though there are tools and strategies (like Git's sparse checkout) to mitigate these issues.
- **Overhead in CI/CD and you do not have enough collaboration with DevOps teams:** Continuous integration and deployment can become more complex as you will need to determine which projects need to be rebuilt and redeployed based on the changed files.

The decision to use a monorepo depends largely on the specific needs and challenges of your projects and team. Some large organizations like Google, Facebook, and Twitter use Monorepos effectively, while others prefer a more modular approach with multiple repositories.

It is essential to weigh the pros and cons and consider factors such as the size of your team, the interconnectedness of your projects, your CI/CD setup, and your overall development workflow.

## **When to Use Bazel**

Bazel is more than just a tool; it is a specialized solution designed to address challenges that surface when an organization hits certain growth ceilings. When faced with these unique challenges, Bazel stands out as arguably the best solution available.

Though Bazel's adoption comes with a significant cost, contributions from industry giants like Google, Apple, Twitter, VMWare, Alibaba, Tencent, Adobe Cloud, Uber, Lyft, SpaceX, Spotify, Pinterest, Tinder, and Reddit are rapidly reducing this barrier. However, it is worth noting that Bazel might not be the most suitable choice for smaller teams, such as a startup with a handful of engineers.

The journey of transitioning to Bazel is becoming less daunting, thanks to an influx of resources. The tech community is seeing a surge in open-source tools, extensive documentation, tech talks, and blog posts focusing on Bazel migration. Moreover, professional services are emerging to assist with Bazel infrastructure setup, crafting custom build rules, and training teams to decipher Bazel's intricate functionalities.

Choosing to adopt Bazel should always be a pragmatic decision, grounded in the needs of your organization. It is crucial to resist the allure of technological hype or the fear of missing out. Instead, take a close look at your business operations and development process. Identify any bottlenecks, hypothesize solutions, and if Bazel appears to be the missing piece to your puzzle, then consider integrating it into your workflow.

## **When Not to Use Bazel**

Bazel is an undeniably powerful building and CI solution, best suited for large and complex projects. However, it

might not be the best fit for every team or project. Let us delve into scenarios where Bazel might not be the ideal choice:

- **Small Teams and Startups:** If you are working within a small team, managing a simple project, or navigating the early stages of a startup, Bazel's inherent complexities might outweigh its advantages.
- **Resource Implications:** Adopting Bazel is not just about using a tool. It demands a strategic recruitment plan to onboard Bazel experts. Additionally, you must invest in training programs to familiarize your current team with Bazel. Infrastructure also comes into play, as running Bazel at scale requires specific setups. In some cases, you might even need to build custom software to fully harness Bazel's capabilities.
- **Single Programming Language Projects:** If your project revolves around a single programming language, particularly one that already boasts robust building tools, Bazel might be overkill. Languages like Go come with native building tools, while languages like Python have a plethora of effective external building tools available.

Bazel's strength shines in large-scale, intricate projects. However, for many scenarios, especially those on a smaller scale, the associated complexities and overheads might not justify its adoption. It is crucial to evaluate your project's unique requirements, gauge your team's familiarity with the tool, and consider long-term maintenance aspects before committing to a build system.

## **Conclusion**

In this chapter, we learned that while monorepos come with challenges like the need for stricter trunk-based

development, specialized CI/CD pipelines, and careful consideration of large-scale changes, they offer significant advantages. These benefits include ensuring all code works together at each commit, facilitating cross-functional code verification, promoting modularity, simplifying dependency management, streamlining toolchain setup, and enhancing developer experience with workspace-aware code editors and IDEs. Additionally, monorepos provide more deployment flexibility, enable granular ownership policies, and offer improved code structure, all while scaling well using familiar tools. Ultimately, the decision to adopt a monorepo structure depends on your specific needs and priorities, but the potential benefits are undeniable.

In the next chapter, we will embark on a journey to demystify Bazel, empowering software developers to harness its full potential for efficient and reproducible builds.

## **Recommended Readings**

- Monorepo  
<https://monorepo.tools/>
- Benefits and Challenges of Monorepo Development Practices | CircleCI  
<https://circleci.com/blog/monorepo-dev-practices/>
- Understanding Monorepos. Introduction | by Roman Sypchenko | Medium  
<https://medium.com/@r.sipchenko/understanding-monorepos-ad9c4ac7b504>
- What is Monorepo? (and should you use it?) - Semaphore  
<https://semaphoreci.com/blog/what-is-monorepo>
- Understanding Monorepos - Ionic Blog  
<https://ionic.io/blog/understanding-monorepos>

- Monorepos in Git | Atlassian Git Tutorial  
<https://www.atlassian.com/git/tutorials/monorepos>
- Misconceptions about Monorepos: Monorepo != Monolith | by Victor Savkin | Nx Devtools  
<https://blog.nrwl.io/misconceptions-about-monorepos-monorepo-monolith-df1250d4b03c>
- About Code Owners - GitHub Docs  
<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners>
- Go Monorepo with Bazel - Uber  
<https://www.uber.com/en-IN/blog/go-monorepo-bazel/>
- How We Designed our Continuous Integration System to be more than 50% Faster  
<https://medium.com/pinterest-engineering/how-we-designed-our-continuous-integration-system-to-be-more-than-50-faster-b70a59342fe2>

# CHAPTER 2

## Getting Started with Bazel

### Introduction

With our current understanding of the key concepts, we are well-prepared to navigate the world of Monorepos and see Bazel's role as the primary tool for incremental builds. This chapter will guide you through setting up Bazel in your local development setup, followed by a thorough exploration of Bazel's fundamental components.

### Structure

In this chapter, the following topics will be covered:

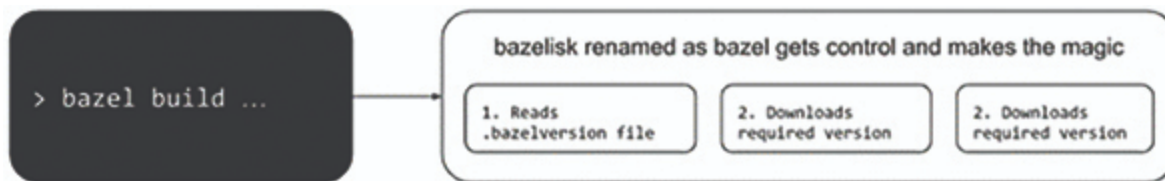
- Installing and Setting Up Bazel
- Bazel Basics: Targets, Workspaces, and BUILD Files
- Building Your First Bazel Project (Java)

### Installing and Setting Up Bazel

Enhancing the already impressive official Bazel documentation available at [bazel.build/docs](https://bazel.build/docs) will be challenging. The *“Installation”* section at [bazel.build/start](https://bazel.build/start) offers detailed guidance for installing Bazel on all mainstream operating systems, as well as within Docker containers and configurations of major IDEs.

We recommend installing Bazel and keeping it updated using Bazelisk. Written in Go, Bazelisk serves as a wrapper for Bazel. It automatically selects the appropriate Bazel version for your current working directory, retrieves it from the official server if necessary, and then transparently

relays all command-line arguments to the actual Bazel binary. You can use Bazelisk as you would Bazel.



**Figure 2.1:** Bazelisk wrapping Bazel

## Installing Bazelisk

For macOS users, it is straightforward:

```
brew install bazelisk
```

This installer automatically adds `bazelisk` to the `PATH`, making it available as both `bazelisk` and `bazel`.

Frontend developers may be interested to know that `Bazelisk` is also available via `npm`:

```
npm install -g @bazel/bazelisk
```

For Windows users, we recommend using Windows Subsystem for Linux and follow the next Linux instructions.

For Linux users, the process requires a few more steps:

1. If `wget` is not already installed, get it by running:

```
apt update && apt upgradeapt install wget
```

2. Download the desired version of `Bazelisk`. For example, to get version `v1.18.0`:

```
wget https://github.com/bazelbuild/bazelisk/releases/download/v1.18.0/bazelisk-linux-arm64
```

**(Note:** You can select a different version from `Bazelisk`'s releases on `GitHub` by replacing the version number in the above URL.)

3. Change the downloaded file's privileges to executable:

```
chmod +x bazelisk-linux-arm64
```

4. Move the file to the desired directory and rename it to Bazel:

```
sudo mv bazelisk-linux-arm64 /usr/local/bin/bazel
```

5. Verify the binary is accessible in your `$PATH`:

```
which bazel
```

This should return:

```
/usr/local/bin/bazel
```

It is advised to set it up as the Bazel binary in your `PATH` (for example, copy it to `/usr/local/bin/bazel`) for seamless operation, as demonstrated above. This way, you will never have to stress about updating Bazel to the latest version.

## **Building Your First Bazel Project (Java)**

Embarking on the journey of creating your first Bazel project is an exciting venture that combines the intricacies of development with the power and efficiency of Bazel's build system. Whether you are a seasoned developer or a newcomer eager to dive into the world of efficient builds, this section will guide you step-by-step. From initial setup to understanding core concepts and, finally, seeing your code come to life.

If you would prefer not to follow the step-by-step example, you can clone the book's repository from GitHub at <https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel> and experiment with the Bazel commands.

Let's start by creating a new directory for the Monorepo and navigate to it, run from the terminal:

```
mkdir my_bazel_project  
cd my_bazel_project
```



Now, let's add a `.bazelversion` file to configure `bazelisk` to use a specific Bazel version:

```
echo "6.3.2" > .bazelversion
```

A **WORKSPACE** is a directory that contains the source files for one or more software projects, as well as Bazel's build outputs. It defines the project root, allowing Bazel to recognize all the related components of the project. This directory contains files named **WORKSPACE** that instruct Bazel on how the project is organized. We will delve deeper into this in the next section, but for now, let us just create an empty **WORKSPACE** file:

```
touch WORKSPACE
```

Let us create a folder for the first project and navigate to it:

```
mkdir -p apps/hello-world
```

Now we can create source folders following the maven standard directory layout (Maven - Introduction to the Standard Directory Layout). This is not mandatory but recommended.

```
mkdir -p apps/hello-world/src/main/java/com/mybazelproject/helloworld
```

And we add a source code file at `apps/hello-world/src/main/java/com/mybazelproject/helloworld/HelloWorld.java` by using your preferred editor or IDE and adding :

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

In order to build this, we need a **BUILD** file at the `apps/hello-world` directory. Files named **BUILD** (or **BUILD.bazel**) are present in every directory of the workspace containing code that needs to be built. They contain build rules that tell Bazel how to build different parts of the project. We will discuss it

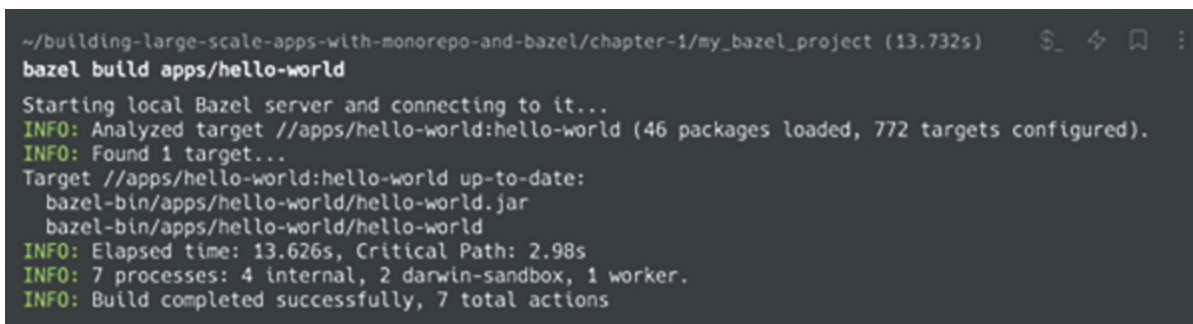
more in the next section, but at this point let us create a file at `apps/hello-world/BUILD.bazel` containing:

```
java_binary(  
  name = "hello-world",  
  srcs =  
    ["src/main/com/mybazelproject/helloworld/HelloWorld.java"],  
  main_class = "com.mybazelproject.helloworld.HelloWorld",  
)
```

Now, from the root of your project directory in the terminal. Build your Java project with **Bazel** by running:

```
bazel build apps/hello-world
```

And we are going to get something like:



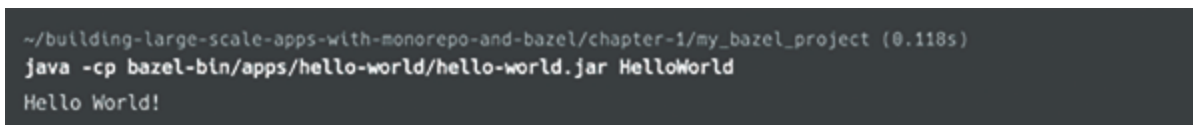
```
~/building-large-scale-apps-with-monorepo-and-bazel/chapter-1/my_bazel_project (13.732s) $ bazel build apps/hello-world  
Starting local Bazel server and connecting to it...  
INFO: Analyzed target //apps/hello-world:hello-world (46 packages loaded, 772 targets configured).  
INFO: Found 1 target...  
Target //apps/hello-world:hello-world up-to-date:  
  bazel-bin/apps/hello-world/hello-world.jar  
  bazel-bin/apps/hello-world/hello-world  
INFO: Elapsed time: 13.626s, Critical Path: 2.98s  
INFO: 7 processes: 4 internal, 2 darwin-sandbox, 1 worker.  
INFO: Build completed successfully, 7 total actions
```

**Figure 2.2:** Execution output from running a Bazel build

Once built, you can run your Java application with:

```
java -cp bazel-bin/apps/hello-world/hello-world.jar HelloWorld
```

And you will get:



```
~/building-large-scale-apps-with-monorepo-and-bazel/chapter-1/my_bazel_project (0.118s) $ java -cp bazel-bin/apps/hello-world/hello-world.jar HelloWorld  
Hello World!
```

**Figure 2.3:** Execution output from running the generated binary

So, let us run again:

```
bazel build apps/hello-world
```

Nothing changes this time so, as an incremental building tooltime, Bazel finishes blazing fast::

```
~/building-large-scale-apps-with-monorepo-and-bazel/chapter-1/my_bazel_project (0.126s)
bazel build apps/hello-world
INFO: Analyzed target //apps/hello-world:hello-world (0 packages loaded, 0 targets configured).
INFO: Found 1 target...
Target //apps/hello-world:hello-world up-to-date:
  bazel-bin/apps/hello-world/hello-world.jar
  bazel-bin/apps/hello-world/hello-world
INFO: Elapsed time: 0.082s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
```

**Figure 2.4:** Execution output from running a Bazel build

You have now built your first application using Bazel. While this is a foundational step, several questions remain: How to incorporate more source files? How to handle third-party libraries? How can we manage dependencies with other projects? And many more. But before diving into these, let us revisit the core concepts in the next section.

## **Bazel Basics**

Now, let us move back to the conceptual world in order to understand the core concepts of Bazel: Workspaces, BUILD Files, Packages, and Targets.

### **WORKSPACE File**

A Bazel workspace is fundamental to understanding the build and test processes orchestrated by Bazel. It is where the journey of crafting software with Bazel begins.

A Bazel workspace is a directory on your file system that encompasses the source files and the build outputs of one or more software projects. It is the environment where Bazel operates to recognize, compile, and test your code. The presence of a special file named **WORKSPACE** (which might be empty) at the root of a directory is what identifies that directory as a Bazel workspace.

The workspace directory serves as the root for Bazel operations. All the paths in Bazel target labels are relative to

this root. All source files, scripts, and assets related to a project reside inside the workspace. It is where you will find your Java, C++, Python, or any other source code files that Bazel will build or test.

When Bazel builds a target, the resulting binaries, libraries, and other outputs are stored in a set of subdirectories within the workspace. These directories are typically named `bazel-out`, `bazel-bin`, `bazel-testlogs`, among others.

The workspace is where Bazel centralizes external dependencies definition. By utilizing the `WORKSPACE` file, developers can declare dependencies from various sources, including Git repositories, HTTP archives, and more. Bazel fetches and stores these consistently, ensuring builds are reproducible.

The `WORKSPACE` file can also be used to set up local configurations for the build environment, specify required tools, and establish build constraints. The `WORKSPACE` file serves as a beacon for Bazel. Its existence denotes the root of the workspace. This file contains directives that help Bazel understand the structure of the software projects, where to find dependencies, and how to fetch them. While it might start as an empty file for basic projects (as we used during the previous exercise), for larger, more intricate projects, it plays a pivotal role in managing dependencies and custom-build rules.

Let us see a `WORKSPACE` file example:

```
# Importing external repositories
load("@bazel_tools//tools/build_defs/repo:http.bzl",
      "http_archive")

# Define external dependencies

# Example 1: Fetching a remote Git repository
http_archive(
    name = "rules_sass",
```

```

    urls =
    ["https://github.com/bazelbuild/rules_sass/archive/master.tar
    .gz"],
    strip_prefix = "rules_sass-master",
)
# Example 2: Fetching a remote HTTP archive (not a Git
repository)
http_archive(
    name = "io_bazel_rules_docker",
    urls =
    ["https://github.com/bazelbuild/rules_docker/archive/4.2.0.ta
    r.gz"],
    strip_prefix = "rules_docker-4.2.0",
)
# Example 3: Fetching an external workspace from a Git
repository
http_archive(
    name = "my_external_workspace",
    urls = ["https://github.com/example/my-external-
    workspace/archive/main.tar.gz"],
    strip_prefix = "my-external-workspace-main",
)
# You can also specify additional configurations for external
# dependencies, such as fetching specific versions, etc.
# Example 4: Fetching a specific version of an external
repository
http_archive(
    name = "external_repo",
    urls = ["https://github.com/example/external-
    repo/archive/v1.0.0.tar.gz"],
    strip_prefix = "external-repo-1.0.0",
)

```

## **Explanation:**

In the `WORKSPACE` file, the journey begins with the import of essential Bazel build rules for external repositories. The load statement serves to introduce the `http_archive` rule from Bazel's toolkit.

Subsequently, you will define your external dependencies leveraging the `http_archive` rule. This prevalent rule fetches external source code, seamlessly integrating it into your Bazel build.

Let us delve into a few examples:

- **Example 1:** Fetch a remote Git repository (`rules_sass`) and pinpoint the URL for the repository's archive file.
- **Example 2:** Procure a remote HTTP archive (`io_bazel_rules_docker`) in a manner akin to the previous example.
- **Example 3:** Highlight the process of obtaining an external workspace (`my_external_workspace`) from a Git repository. This comes into play when the objective is to encompass an entire Bazel workspace from an external origin.
- **Example 4:** Outline the procedure to fetch a distinct version of an external repository (`external_repo`) by providing the URL to a specific release or tag.

These methodologies illustrate the typical patterns used to delineate external dependencies within a Bazel `WORKSPACE` file. Depending on the unique requisites of your project, you have the flexibility to bring in a variety of external repositories and tweak configurations as deemed necessary. Stay tuned for in-depth explorations in the subsequent sections.

It is essential to understand that a Bazel workspace encapsulates its environment. Things outside the workspace are generally invisible to Bazel unless explicitly specified. This encapsulation ensures that builds are hermetic, that is,

they are consistent and reproducible since they are not affected by external factors.

In summary, Bazel's workspace is the foundational ground for Bazel's operations. It provides structure, consistency, and a controlled environment, ensuring that software is built and tested in a predictable and efficient manner.

## **BUILD Files**

Bazel's **BUILD** files are central to the build and test process orchestrated by Bazel. These files act as blueprints, instructing Bazel on the construction of the software.

A **BUILD** file is a plain-text document that resides in a directory, containing instructions — in the form of build rules — that guide Bazel to generate one or more build outputs from the directory's source files (and potentially from other directories).

Key Aspects of **BUILD** files include:

- **Granular Configuration:** Bazel encourages fine-grained dependencies, allowing you to detail the minimal set of input files required for a build or test. **BUILD** files facilitate this, enabling developers to fine-tune individual build targets and their specific dependencies.
- **Directory Association:** Every directory that houses source files built directly by Bazel should have its dedicated **BUILD** file, promoting modular and scalable build configurations.
- **Target Specification:** Within a **BUILD** file, you will define entities termed "targets." A target could be a singular source file, a compiled library, an executable binary, a test suite, or even a data package.
- **Rule Utilisation:** Targets are shaped using "*rules*". Bazel offers a broad spectrum of rules, such as

`java_binary` for Java executables, `cc_library` for C++ libraries, or `py_test` for Python test suites. These rules dictate the kind of output Bazel generates.

- **Dependency Declaration:** Besides outlining the build outputs, BUILD files enable the declaration of dependencies on other targets. If target A relies on target B, Bazel ensures B is signalling before constructing A.
- **Load Statements:** BUILD files can incorporate load statements at their outset for custom rules or rules from external sources, signalling where these rules can be located.

The architecture and syntax of a BUILD file also include elements like:

- **Labels:** Every target is pinpointed by a unique label within the workspace. Labels adhere to the `@repository//path:target_name` template. For most local workspace targets, this condenses to `//path:target_name`.
- **Attributes:** BUILD file rules possess several attributes. Common ones encompass `name` (mandatory for all rules), `srcs` (which lists source files), and `deps` (which announces dependencies on other targets).
- **Glob Function:** Employed to identify multiple files using wildcards, the `glob` function is advantageous for incorporating numerous source files without itemizing each separately.

### Simple example:

In the context of a straightforward Java project, a BUILD file might resemble:

```
java_binary(  
  name = "hello-world",  
  srcs = glob(["*.java"]),  
  main_class = "com.example.helloworld.HelloWorld",
```



)

Here, the `java_binary` rule instructs Bazel to craft a runnable Java binary.

And here is a common **BUILD** file structure with an example:

```
# Importing build rules and macros
load("@rules_cc//cc:defs.bzl", "cc_library", "cc_binary")
load("@io_bazel_rules_docker//container:container.bzl",
      "docker_image")

# Define build targets

# Example 1: C++ Library
cc_library(
    name = "my_cpp_library",
    srcs = ["my_cpp_source.cc", "my_cpp_header.h"],
    hdrs = ["my_cpp_header.h"],
    visibility = ["//visibility:public"],
)

# Example 2: C++ Binary
cc_binary(
    name = "my_cpp_binary",
    srcs = ["my_cpp_binary.cc"],
    deps = [":my_cpp_library"],
)

# Example 3: Docker Image
docker_image(
    name = "my_docker_image",
    base = "@ubuntu_bionic//image",
    files = ["Dockerfile", ":my_cpp_binary"],
    entrypoint = ["/my_cpp_binary"],
)
```

### **Explanation:**

In the **BUILD** file, you start by importing build rules and macros that you need for your project. These imports are often specific to the programming language or framework

you are working with. In this example, we import rules for C++ (`cc_library` and `cc_binary`) and Docker (`docker_image`).

- **Example 1** defines a C++ library target named `my_cpp_library`. It specifies the source files (`srcs`), header files (`hdrs`), and sets visibility to public. The library can be referenced by other targets within the same **BUILD** file or other **BUILD** files in your project.
- **Example 2** defines a C++ binary target named `my_cpp_binary`. It specifies the source files (`srcs`) and declares a dependency on `:my_cpp_library`. This means that this binary target depends on the `my_cpp_library` target defined in Example 1.
- **Example 3** defines a Docker image target named `my_docker_image`. It specifies the base image to use (`base`), the Dockerfile (`files`), and an entrypoint for the container. Additionally, it depends on the `:my_cpp_binary` target, indicating that the Docker image should include the compiled binary produced by `my_cpp_binary`.

While the **WORKSPACE** file establishes the foundation, signifying the root of the Bazel project and overseeing external dependencies, **BUILD** files are the stars within individual directories, elaborating on the actual build procedure.

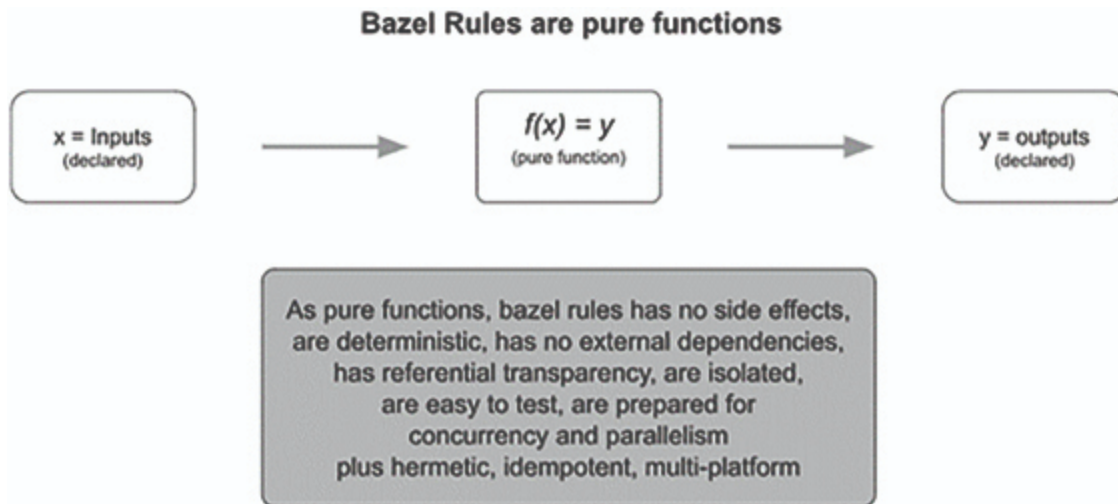
In summary, Bazel's **BUILD** files constitute the crux of the build configuration, offering intricate maps that guide Bazel in the nuanced task of transforming source code into operational software, ensuring accuracy, consistency, and efficiency.

## [Build Rules](#)

In the realm of Bazel, build rules stand as the linchpin between raw source code and the software artifacts (like binaries or libraries) they metamorphose into. Essentially, rules dictate how Bazel transmutes source files into outputs.

To truly grasp Bazel's prowess, it is crucial to delve deep into the intricacies of these build rules.

At their core, build rules are recipes. They describe the transformation process — which inputs (source files or dependencies) to take, the operations to perform on these inputs (like compilation or linking), and the expected outputs (executables, libraries, or other files).



**Figure 2.5:** Bazel rules as pure functions

Each rule comprises:

- **Inputs:** These could be source files, headers, libraries, or other dependencies.
- **Outputs:** The resulting files after the rule is executed—like binaries, libraries, or archives.
- **Actions:** The procedures Bazel must undertake to transform inputs into outputs. This might involve invoking a compiler, linker, or other tools.

When Bazel identifies a rule in a **BUILD** file, it adds the rule's outputs to its build graph. If any of these outputs is missing or if any of its inputs has changed, Bazel executes the rule's actions to regenerate the outputs.

Rules contain attributes that provide specific data for their execution. Common attributes include:

- **name**: A unique identifier for the rule within the BUILD file.
- **srcs**: Lists the source files.
- **deps**: Declares dependencies on other targets or rules.
- **visibility**: Dictates which other packages (if any) can depend on this rule.
- ... and many more, depending on the specific rule.

Bazel comes bundled with a rich set of predefined rules, also known as Bazel Core rules. For instance:

- **java\_library**: Defines a set of Java classes and resources.
- **cc\_binary**: Represents a C++ executable.
- **py\_test**: A test rule for Python code.

Each language or platform Bazel supports generally has a suite of associated core rules.

While Bazel provides a vast array of built-in rules, you might encounter situations where you need a bespoke solution. Bazel's extensible nature lets you craft custom rules using its Starlark configuration language. These rules, once defined, can be used just like the built-in ones.

One of Bazel's standout features is its incremental build capability, which hinges on its understanding of dependencies. Rules declare their dependencies explicitly, allowing Bazel to build only what's necessary when a change occurs, instead of rebuilding everything.

Each rule generates specific outputs, which can, in turn, be inputs for other rules. This chaining forms a dependency tree. At the end of this chain, Bazel produces the final build artifact.

Bazel's build rules are the beating heart of its build and test processes. They encapsulate the intricate details of software construction, allowing developers to achieve fast, reproducible builds. Understanding these rules is paramount for harnessing the full power of Bazel in any software development endeavor.

## **Types of build rules**

Build rules in Bazel are typically organized into families based on the programming language they support. For instance, the rules `cc_binary`, `cc_library`, and `cc_test` pertain to the creation of C++ binaries, libraries, and test frameworks, respectively. This pattern of nomenclature extends across various languages, exemplified by the prefix `'java_*` for Java-related rules. While many of these rules are delineated in the Build Encyclopedia (see <https://bazel.build/reference/be/overview>), Bazel allows the customization and creation of new rules.

The `*_binary` rules are designed to compile executable programs. Upon completion of a build, the generated executable is placed in the build tool's binary output tree, correlating with the rule's label. For example, an executable from the rule `//my:program` would be found at `$(BINDIR)/my/program`.

For certain languages, `*_binary` rules additionally generate a `runfiles` directory. This directory aggregates all the files specified in the rule's `data` attribute, as well as those in its transitive dependency closure, facilitating deployment.

The `*_test` rules serve as a derivative of the `*_binary` rules, tailored for conducting automated tests. These rules generate test programs that signal success by returning zero. The associated `runfiles` tree for tests contains essential files for runtime access, ensuring that tests operate in a controlled environment. For example, a `cc_test`

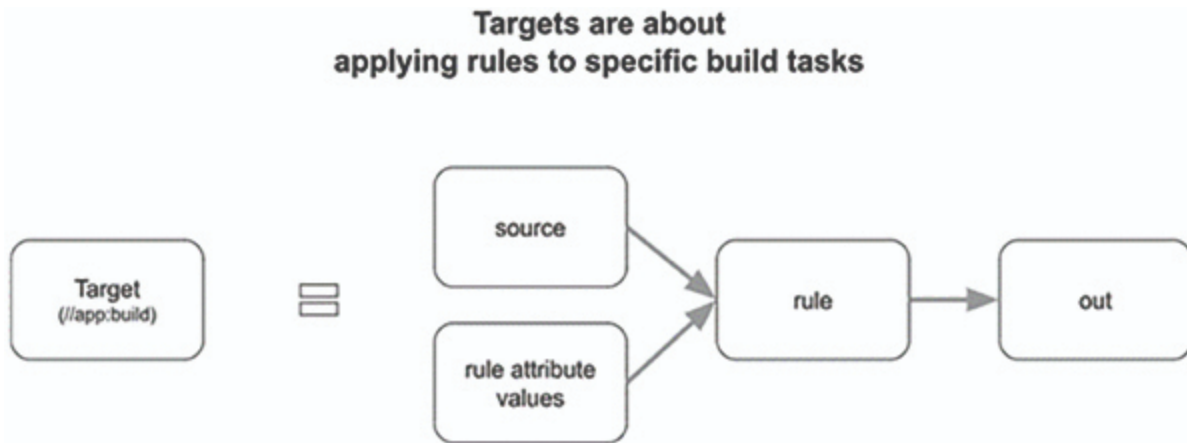
program can access files specified in its data attribute during execution.

Lastly, `*_library` rules define separately compiled units within a language's ecosystem. These rules allow for modular development, where libraries can be interdependent or be leveraged by binaries and tests, adhering to the principles of separate compilation.

## Targets

At its core, Bazel is a tool designed to provide efficient, reproducible, and correct builds. To accomplish this, it uses a series of instructions, written by developers, to determine what needs to be built and how. These instructions are defined in terms of "targets".

In the world of Bazel, a build target represents a set of source files and a set of rules that determine how those files should be transformed into output products. This could be anything from compiled binaries and libraries to documentation and deployable packages.



**Figure 2.6:** Target as sources + rule

Every target is defined within a BUILD file using a specific kind of rule. Each rule's structure looks somewhat like a function call in a typical programming language, with the

rule name serving as the function name and its attributes serving as arguments.

For example:

```
java_binary(  
  name = "my_app",  
  srcs = ["MyApp.java"],  
  deps = [":my_library"],  
)
```

Here:

`java_binary` is the rule.

`name`, `srcs`, and `deps` are attributes of this rule.

Every target is identified by a unique label. A label is essentially a path to the target, which includes its package path and its name.

The format is: `//path/to/package:target_name`

In our preceding example. if the BUILD file is in the root of the workspace, the label for the target would be `//:my_app`.

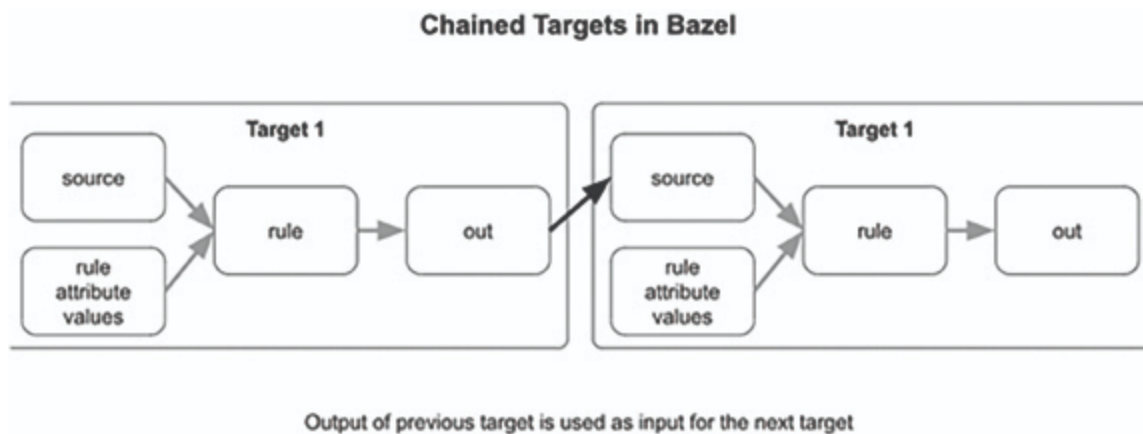
While targets can represent a wide variety of things, they typically fall into a few categories:

- **Source Targets:** These are simply references to source files.
- **Rule Targets:** These are the most common targets and are produced by applying a rule to some source files and dependencies.
- **Package Group Targets:** Used to define sets of packages.

Targets often depend on other targets. For instance, a binary might depend on a library. These dependencies are usually specified using the `deps` attribute. Bazel ensures that dependencies are built before the dependent target.

Each rule, when applied to its sources and dependencies, produces one or more outputs. For a `java_binary` rule, the primary output is a runnable Java binary. For a `java_library` rule, it's a Java archive (**.jar** file).

Bazel allows you to specify which other packages (if any) can depend on a given target using the visibility attribute. This lets large projects enforce modular design and encapsulation.



**Figure 2.7:** Chained targets

Sometimes, the same target needs to be built in multiple ways. For instance, you might have debug and release builds. Bazel handles this using build configurations. Targets can be associated with specific configurations using the `cfg` attribute.

Bazel's build targets are foundational to its design and functionality. By providing a granular and efficient way to define what needs to be built and how, Bazel allows developers to have precise control over the build and test processes, ensuring reproducibility and correctness across various environments and platforms.

## [Labels](#)

We have already explained labels in the previous pages, but let us dive deeper. In Bazel's universe, everything is a



target: a file, a rule, a test suite, and so on. Every target has a unique identifier, which is termed a “*label*”. *These labels are used to refer to targets within Bazel.*

A Bazel label is conventionally structured in the following way:

```
@repository//path:target_name
```

Let us break this down:

- **@repository**: This is the name of the external repository. For targets in the main repository (that is, your workspace), this portion can be omitted. For targets in local projects, you will not typically need this.
- **//path**: This is the relative path from the root of the Bazel workspace to the directory containing the target. The double slashes (//) indicate the root of the workspace.
- **:target\_name**: This is the name of the target itself. It corresponds to the name attribute of the rule in a **BUILD** file.

Examples:

```
//my/app:app_binary
```

This label refers to a target named **app\_binary** located in the **my/app** directory of the main repository.

```
@my_repo//lib/utils:utility_funcs
```

This label refers to a target named **utility\_funcs** in the **lib/utils** directory of an external repository named **my\_repo**.

While the preceding representation is the canonical label, Bazel does allow for abbreviated forms:

- If the target name is the same as the package’s directory, you can omit the target name. So, **//my/app:app** can be abbreviated to **//my/app**.

- If the target is in the main repository, you can omit the repository name. So, `@main_repo//my/app:app` can just be `//my/app:app`.

Labels in Bazel are always absolute, which means they always refer to the same target, irrespective of where they are used. However, in `BUILD` files, you can use relative labels. If you are in the `//my/app` directory and want to reference a target in `//my/utils`, you can just use `:utils`. Bazel will understand this as `//my/app:utils`.

By using labels, you precisely pinpoint the target you are referring to, eliminating ambiguity. This ensures that Bazel's builds are reproducible and consistent. As projects grow and evolve, their structure may change. Labels provide a stable way to refer to targets, regardless of the project's size or complexity. With labels, you can easily define interdependencies between various components of your software. This modularity facilitates efficient incremental builds, as Bazel can quickly determine which parts of the codebase are affected by a change.

In essence, labels in Bazel are much more than mere identifiers. They embody Bazel's philosophy of precision, clarity, and efficiency. Understanding and mastering the concept of labels is crucial for harnessing the full power of Bazel as a build-and-test system.

## [Packages](#)

Central to Bazel's organization and structuring are "*packages*". A package in Bazel is a collection of related files and a `BUILD` file. The package also includes all the files in its directory, up to but excluding the next directory in the filesystem tree that contains a `BUILD` file (that is, subdirectories without `BUILD` files are considered part of the package).

Every package contains a **BUILD** file, defining how the software is constructed from its source files. This file is essential for Bazel to recognize the directory as a package.

The boundary of a package is determined by its containing directory. All subdirectories within that directory, which do not have their own **BUILD** file, are considered a part of the package. If a subdirectory does contain its own **BUILD** file, it is treated as a separate package.

```
/my_project/  
|-- WORKSPACE  
|-- BUILD  
|-- lib/  
|   |-- BUILD  
|   |-- helper.cc  
|   `-- helper.h  
|-- app/  
|   |-- main.cc  
|   |-- BUILD  
  
|-- test/  
|   |-- test_main.cc  
|   |-- BUILD
```

Packages are named by the relative path to their directory from the workspace's root directory, where the top-most **WORKSPACE** file resides. For instance, if a package's directory path from the workspace root is **apps/example/project**, then the package name is **//apps/example/project**.

Packages allow for fine-grained control over builds. **By dividing a large project into smaller packages, developers can make small changes without needing to rebuild the entire project.** Bazel will only rebuild the parts of the project that depend on the modified package, optimizing the build times.

Bazel packages offer strict control over dependencies with visibility rules. These rules determine which other packages

can reference the current package. By default, packages are private, meaning they cannot be referenced by other packages unless specified otherwise. This ensures that unintended interdependencies are avoided.

Apart from the packages defined in the local workspace, Bazel can also refer to external packages. These are usually dependencies defined in the `WORKSPACE` file that are fetched from external sources, such as GitHub repositories or package managers like npm, maven, and so on.

Every target within a package is identified by a unique label. This label is crucial for referencing particular targets across different packages and looks like `//path/to/package:target_name`. For example, `//src/java/com/example/project:my_lib` references the `my_lib` target in the `src/java/com/example/project` package.

Using packages provides modularity as each package can be seen as a module that can be built, tested, and deployed independently. It also brings parallelism as Bazel can build or test multiple packages in parallel, leveraging multi-core CPUs. Finally, by using packages we could get clean Dependency Graphs as packages make it easier to maintain a clear and organized dependency graph, helping avoid “*dependency hell*”.

Packages in Bazel are more than just a structural unit; they are foundational to how Bazel approaches the build process. By understanding packages, developers can better organize their codebase, manage dependencies efficiently, and harness the full power of Bazel’s optimized, parallel, and incremental builds.

## Queries

Bazel’s query language allows you to inspect various aspects of your build configurations. Whether it is understanding the dependency graph, seeing how different

rules interact, or analyzing which targets might be affected by a change, Bazel's query functionality provides a robust toolset.

Bazel can provide a detailed view of the dependency graph. This is especially useful in large projects where tracking dependencies manually can be daunting. So, before making changes, you can use Bazel's query system to understand which parts of your project might be affected. This aids in risk assessment and planning.

Bazel supports multiple build configurations, and with queries, you can understand how different configurations might impact your build.

The most basic **query** could be to ask Bazel about all the dependencies of a **target**:

```
bazel query 'deps(//path/to:target)'
```

Bazel's query language supports various filters. For instance, if you want to see all the Java libraries a **target** depends on:

```
bazel query 'kind(java_library, deps(//path/to:target))'
```

As we learned before, Bazel's rules come with various attributes (like **srcs**, **deps**, and so on). You can query based on these attributes. For example, to find all targets that depend on a specific file:

```
bazel query 'attr(srcs, "path/to/specific/file",  
//path/to:target)'
```

Also, Bazel can tell you which targets depend on a given target. This is called a **reverse dependency** query. It can be immensely useful when refactoring or deleting targets:

```
bazel query 'rdeps(//path/to:..., //path/to:specific_target)'
```

To understand the magic behind the query system, let us review the underlying principles:

- **Graph-based:** At its core, Bazel's build and test system is a Directed Acyclic Graph (DAG). Each node in this graph represents a build or test action, and the edges represent the dependencies between these actions. The query system navigates this graph.
- **Precision:** Bazel's queries are precise because of its strict dependency management. There is no ambiguity; if a target is returned in a query result, you can trust that it is relevant to your query.
- **Speed:** The speed of Bazel's query system comes from its optimized in-memory graph representation. Bazel does not need to access files on disk or re-evaluate build rules. It uses the graph to answer queries.

While Bazel's query language appears powerful, it can be intricate, particularly for detailed queries. It might require time to grasp the language's nuances. By default, a query encompasses the entire workspace. However, for substantial projects, this can lead to extended query durations. Limiting the scope often accelerates the process.

Bazel's query system is a potent tool, especially for large codebases and projects. By providing insights into the build and dependency structure, it aids developers and build engineers in ensuring efficient and error-free builds. As with any robust tool, there is a learning curve, but the depth of insights it provides is well worth the effort.

## [Dependencies](#)

One of the essential attributes in these build rules is the `deps` (short for dependencies) attribute. It specifies other targets that the current target depends on. For instance, if you have a C++ program, the dependencies might be the various `.cc` (source) and `.h` (header) files required to compile that program.

As we learned before, targets can be source files or rules. A rule might produce an executable or a library, or it might perform an action like copying files. By defining dependencies, Bazel knows the order in which tasks need to be executed. For example, before compiling a C++ program, Bazel needs to ensure that the required libraries are built. When you make a change to your source code and rebuild, Bazel uses the dependency graph to intelligently determine which parts of the codebase need to be recompiled, allowing for faster builds.

Example:

```
cc_binary(  
  name = "hello-world",  
  srcs = ["hello-world.cc"],  
  deps = [":hello-greet"],  
)
```

In the preceding **BUILD** file, there's a `cc_binary` rule which is used for building C++ binaries. The target name is `"hello-world"`, and it has a source file `hello-world.cc`. The `deps` attribute indicates that this target depends on another target named `"hello-greet"` (which could be a library).

The dependency (`deps`) attribute in Bazel's **BUILD** file is crucial for maintaining the order of builds and ensuring that Bazel can perform efficient and incremental builds. By correctly specifying the dependencies, you tell Bazel the exact relationship between different parts of your code, allowing Bazel to optimize the build process. As a junior developer, always ensure that your `deps` attributes are up-to-date to prevent build errors and to leverage Bazel's efficiency to the fullest.

## [Bazel Sandboxing](#)

At its core, Bazel is a build tool. Think of it as a highly specialized chef that prepares dishes (software builds)

according to recipes (build rules). But like any great chef, Bazel needs a clean kitchen to ensure there is no cross-contamination between dishes. In Bazel's world, this clean kitchen is the **sandbox**.

Imagine you are building a sandcastle in a sandbox at a playground. The walls of the sandbox ensure that your sand stays within its confines, and that you do not mix it up with leaves, dirt, or other things from the playground. Also, these walls ensure that all required artifacts to cook are inside the sandbox and we do not need anything else from outside. Similarly, in software, a sandbox is a controlled environment where specific tasks run, isolated from the rest of the system.

Why does bazel use sandboxing?

- **Reproducibility:** Bazel wants every build to produce the same result, regardless of where or when you run it. By isolating builds in a sandbox, Bazel ensures that the build only has access to declared dependencies and cannot be affected by random files on your system.
- **Safety:** By running tasks in a sandbox, Bazel can prevent unintended side effects. It is like making sure a kid with paints does not accidentally color on the walls outside the sandbox.
- **Parallelism:** Since builds are isolated, Bazel can run multiple builds or tests in parallel without them interfering with each other. It is like having multiple sandboxes in a playground, each with its own kid building a unique sandcastle.

So, let us review a task lifecycle:

1. When Bazel starts a build, it creates a temporary environment: just like how each sandcastle gets its own spot in a sandbox, each build or test gets its own isolated environment on the disk.



2. Bazel then populates this environment only with the files and tools declared as dependencies, injecting only what is needed. It is like giving the kid-specific buckets and tools to make the sandcastle and nothing more.
3. With everything set, Bazel runs the build or test.
4. Once done, it clears out the temporary environment, ensuring there is no residue left behind.

While sandboxing offers many advantages, it also comes with some overhead. Creating and tearing down isolated environments can add to the build time, especially if not optimized. However, the benefits of reproducibility and safety often outweigh the slight increase in build time.

## Conclusion

The journey through this chapter provides developers with a foundational understanding and the necessary tools to effectively utilize Bazel for their projects. Beginning with the installation of Bazel and Bazelisk, developers are equipped to set up and initiate their first Bazel project, with a special focus on Java. The chapter meticulously explores the core components of Bazel, including Workspaces, BUILD Files, and Targets, each playing a pivotal role in the construction and management of build processes. Through the examination of **WORKSPACE** files and **BUILD** Files, developers learn to define and configure their projects, while the discussion on Build Rules, Targets, Labels, Packages, Queries, and Dependencies offers insight into the granular control and flexibility afforded by Bazel. Furthermore, the concept of Bazel sandboxing is introduced, highlighting Bazel's commitment to providing reproducible and reliable builds in an isolated environment, thereby minimizing the risk of external interferences and inconsistencies. The chapter concludes with a set of recommended readings, aimed at further deepening the developer's understanding

and mastery of Bazel. Armed with this knowledge, developers are empowered to harness the full potential of Bazel, navigating through its features with confidence and employing it as a robust and efficient build and test tool for their diverse development needs.

In the next chapter, we will dive deeper into Bazel's Rule-Based Build System, unlocking its potential for optimizing and streamlining your software development workflows. You will learn how to customize build and compilation rules, tailoring them to suit your project's unique requirements. Additionally, we will explore the intricacies of writing and executing a `genrule`, allowing you to harness Bazel's power for specialized tasks, and address any custom needs that may not be adequately served by default rules.

## **Recommended Readings**

- Build programs with Bazel  
<https://bazel.build/run/build>
- Configurable Build Attributes | Bazel  
<https://bazel.build/configure/attributes>
- Query quickstart | Bazel  
<https://bazel.build/query/quickstart>
- Open-source Bazel Build Tutorial, Examples, and Advantages  
<https://semaphoreci.com/blog/bazel-build-tutorial-examples>
- Bazel Installation Guide  
<https://docs.bazel.build/versions/main/install.html>
- Getting Started with Bazel  
<https://docs.bazel.build/versions/main/getting-started.html>

- Bazelisk: A Bazel Launcher  
<https://github.com/bazelbuild/bazelisk>
- Using Bazelisk to Manage Bazel Versions  
<https://blog.bazel.build/2019/02/11/bazelisk.html>
- Bazel Java Tutorial  
<https://docs.bazel.build/versions/main/tutorial/java.html>
- Building Java Projects with Bazel  
<https://www.baeldung.com/bazel-build-java>
- Introduction to Bazel: Building a Java Project  
<https://docs.bazel.build/versions/main/tutorial/java.html>
- Bazel Concepts and Terminology  
<https://docs.bazel.build/versions/main/build-ref.html>
- Targets, Labels, and Packages  
[https://docs.bazel.build/versions/main/build-ref.html#packages\\_targets\\_and\\_rules](https://docs.bazel.build/versions/main/build-ref.html#packages_targets_and_rules)
- Bazel Query Command  
<https://docs.bazel.build/versions/main/query.html>
- Managing Dependencies  
<https://docs.bazel.build/versions/main/external.html>
- Bazel Sandboxing Mechanism  
<https://docs.bazel.build/versions/main/sandboxing.html>
- Bazel: Build and Test Software of Any Size, Quickly and Reliably  
<https://www.oreilly.com/library/view/bazel/9781492042487/>
- Building Large Angular Applications with Bazel  
<https://blog.nrwl.io/building-large-angular-applications-with-bazel-883b7d9dc0e4>

# CHAPTER 3

## Bazel Build Rules and Configuration

### Introduction

In this chapter, we will embark on a journey to explore the intricacies of Bazel's rule-based build system. As a software developer, understanding the inner workings of Bazel is crucial for optimizing your development workflow. We will start by introducing you to your first Bazel rule, breaking down its components and explaining how it fits into the larger ecosystem. Next, we will delve into the art of customizing build and compilation rules, discussing why customization is necessary and the key components involved in creating your own rules. Whether you need to fine-tune your build process or address unique project requirements, we will show you how to solve custom needs not served by default rules. You will also learn how to write and execute a genrule, an essential tool for generating files during the build process. Throughout this chapter, we will demystify Bazel configuration, covering the `WORKSPACE` file, `BUILD` file, Bazel flags, and `.bazelrc` file, including their syntax and best practices. We will wrap up with commonly used options and provide a conclusion to help you solidify your understanding of Bazel's rule-based build system, along with a list of recommended readings for further exploration.

### Structure

In this chapter, the following topics will be covered:

- Exploring Bazel's Rule-Based Build System
- Customizing Build and Compilation Rules
- Writing and Executing a Genrule
- Solving any Custom Needs not Served by Default Rules

## **Exploring Bazel's Rule-based Build System**

At the heart of Bazel's design lies its rule-based build system, a unique feature that makes it particularly adept at handling vast and intricate codebases. This section dives deep into the intricacies of Bazel's rules and the power they hold.

From our discussions in earlier sections, we understand that rules in Bazel define the steps transforming inputs and dependencies into outputs. Specifically, each rule delineates:

- Its input requirements, including source files, libraries, and other dependencies.
- The procedure to process these inputs, like compilation or linking.
- The resulting outputs, be they executables, libraries, or other files.

These rules mandate the explicit declaration of dependencies, fostering precision in dependency tracking. Such granularity allows Bazel to execute exceptionally efficient incremental builds. By accurately determining codebase dependencies, Bazel can selectively rebuild components affected by a change.

Bazel's agnostic approach towards language ensures it can virtually support any platform or language. Its extensibility is rooted in its rule system. While Bazel offers core rules for mainstream languages like Java, C++, and Python, its

capacity for extension through custom rules caters to other languages or specific platform build processes.

Starlark, inspired by Python and initially termed Skylark, serves as Bazel's scripting language. It plays a pivotal role in forging custom build rules and amplifying Bazel's capabilities.

One of the highlights of Bazel's rules and build process is the assurance of hermetic builds. Outputs remain insulated from external influences, guaranteeing consistency, reproducibility, and uniformity across diverse environments.

Bazel's toolchain concept facilitates the abstraction and management of compilers, linkers, and similar tools. Rules can solicit a specific toolchain, and Bazel's intrinsic logic will identify the apt tool based on the prevailing platform and configuration.

A key advantage of Bazel's rules is the support for features like remote caching and execution. The content-addressable caching mechanism means that if a build identical to the current one has been executed elsewhere, Bazel can fetch the outputs without redoing the work. We will delve deeper into this in upcoming sections.

Furthermore, Bazel's rules are tailored to endorse cross-compilation scenarios, facilitated by stipulating relevant toolchains.

## [Your First Bazel Rule](#)

This section will guide you through the process of creating a simple Bazel rule using Starlark.

First, create a new directory for your project and initialize it with a **WORKSPACE** file:

```
mkdir my_bazel_project
cd my_bazel_project
touch WORKSPACE
```

Next, create a file named `my_rule.bzl` in the root of your project. This file will contain our custom rule.

Inside `my_rule.bzl`, start by defining a simple rule that generates a file with custom text:

```
def _my_rule_impl(ctx):
    output = ctx.outputs.output
    content = ctx.attr.content
    ctx.actions.write(output, content)

my_rule = rule(
    implementation = _my_rule_impl,
    attrs = {
        "content": attr.string(),
    },
    outputs = {
        "output": "%{name}.txt",
    },
)
```

In this code:

- `my_rule` is the rule we are creating by using core function `rule`.
- `_my_rule_impl` is the implementation function. It describes what the rule does.
- `ctx` is the rule context, which provides access to rule attributes, declared outputs, and actions.
- The `actions.write` method writes a string to an output file.
- The rule function defines the rule with its implementation, attributes, and outputs.

Now, let us use this rule. Create a **BUILD** file in the root of your project with the following content:

```
load("//:my_rule.bzl", "my_rule")

my_rule(
```

```
    name = "hello",  
    content = "Hello, Bazel!",  
)
```

The `load` function imports the custom rule from our `my_rule.bzl` file. The `my_rule` target will generate a file named `hello.txt` with the content `"Hello, Bazel"`.

Run the following command in your project directory:

```
bazel build :hello
```

Bazel will process the `hello` target, execute your custom rule, and generate the `hello.txt` file.

After building, you should find the `hello.txt` file in the `bazel-bin` directory. Check its content:

```
cat bazel-bin/hello.txt
```

You should see:

```
Hello, Bazel!
```

Congratulations, you have just created your first custom Bazel rule using Starlark! While this is a simple example, Bazel's flexibility allows you to create more complex and powerful rules tailored to your specific build and test needs. Dive deeper into Starlark's documentation and Bazel's rule concepts to unlock even more potential!

## [Dissecting a Core Bazel Rule](#)

When working with Bazel, understanding the design and mechanics of core rules is essential for efficient builds and extensibility. There are many core Bazel rules, such as `java_library` or `java_binary`, but let us dissect `java_plugin` as it has a good ratio of complexity and length. Let us dive deep into this rule, uncovering its design and inner workings.

`java_plugin` is a core Bazel rule for defining Java plugins as annotation processors. Annotation processors are tools that can generate, analyze, and manipulate code based on



annotations present in the source files. The `java_plugin` rule tells Bazel how to handle and use these processors during Java builds.

`java_plugin` has various attributes that configure its behavior, including:

- **deps:** This attribute specifies dependencies for the annotation processor, which often includes libraries the processor needs to link against.
- **processor\_class:** The fully qualified name of the annotation processor class. This class should extend `javax.annotation.processing.AbstractProcessor`.
- **generates\_api:** A boolean indicating whether the annotation processor generates API (publicly accessible code). This affects how the rule interacts with the `java_library`'s `strict_deps` attribute.
- **data:** Specifies files that the annotation processor needs at runtime.
- There are also some other attributes such as `neverlink`, `plugins`, `resources` as can be checked in [https://bazel.build/reference/be/java#java\\_plugin](https://bazel.build/reference/be/java#java_plugin)

In its internal design it was considered as priority:

- **Dependency Tracking:** Bazel optimizes builds through granular dependency tracking. When a `java_library` or `java_binary` rule uses a `java_plugin`, Bazel ensures that any change in the plugin or its dependencies triggers a rebuild of dependent Java targets.
- **Processor Path:** When Bazel compiles Java targets dependent on a `java_plugin`, it sets up the processor path, ensuring the annotation processor can be discovered and applied during the compilation.
- **API Generation:** If `generates_api` is set to true, Bazel treats the generated code as part of the public API. This

is crucial because it can influence the rebuilds of other dependent targets.

- **Data Dependency:** Sometimes, annotation processors need additional data files (for example, configurations or resources). The data attribute ensures that these files are available to the processor during runtime.

In the Bazel source code, the `java_plugin` rule is written using Starlark and utilizes various internal functions and utilities provided by Bazel's Java rules ecosystem. The rule's definition in Starlark leverages functions for Java compilation, setting up the processor path, handling dependencies, and managing outputs.

Let us have a look at the rule's source code (full version at [https://github.com/bazelbuild/bazel/blob/master/src/main/starlark/builtins\\_bzl/common/java/java\\_plugin.bzl](https://github.com/bazelbuild/bazel/blob/master/src/main/starlark/builtins_bzl/common/java/java_plugin.bzl)):

```
load(":common/java/basic_java_library.bzl",
    "basic_java_library",
    "construct_defaultinfo")
load(":common/java/java_library.bzl", "JAVA_LIBRARY_ATTRS",
    "JAVA_LIBRARY_IMPLICIT_ATTRS")
load(":common/rule_util.bzl", "merge_attrs")
load(":common/java/java_semantics.bzl", "semantics")
load(":common/java/java_info.bzl", "JavaPluginInfo")
def bazel_java_plugin_rule(
    ctx, srcs = [], data = [], generates_api = False,
    processor_class = "", deps = [], plugins = [],
    resources = [], javacopts = [], neverlink = False,
    proguard_specs = [], add_exports = [], add_opens = []):
    target, base_info = basic_java_library(
        ctx, srcs, deps, [], # runtime_deps
        plugins, [], # exports
        [], # exported_plugins
        resources, [], # resource_jars
```

```

    [], # classpath_resources
    javacopts, neverlink, proguard_specs = proguard_specs,
    add_exports = add_exports, add_opens = add_opens,
)
java_info = target.pop("JavaInfo")

# Replace JavaInfo with JavaPluginInfo
target["JavaPluginInfo"] = JavaPluginInfo(
    runtime_deps = [java_info],
    processor_class =
        processor_class if processor_class else None,
    # ignore empty string (default)
    data = data,
    generates_api = generates_api,
)
target["DefaultInfo"] = construct_defaultinfo(
    ctx, base_info.files_to_build, base_info.runfiles,
    neverlink,
)
target["OutputGroupInfo"] =
    OutputGroupInfo(**base_info.output_groups)
return target

def _proxy(ctx):
    return bazel_java_plugin_rule(
        ctx, ctx.files.srcs, ctx.files.data, ctx.attr.generates_api,
        ctx.attr.processor_class, ctx.attr.deps, ctx.attr.plugins,
        ctx.files.resources, ctx.attr.javacopts, ctx.attr.neverlink,
        ctx.files.proguard_specs, ctx.attr.add_exports,
        ctx.attr.add_opens,
    ).values()

JAVA_PLUGIN_ATTRS = merge_attrs(
    JAVA_LIBRARY_ATTRS,
    {
        "generates_api": attr.bool(),
        "processor_class": attr.string(),
    }
)

```

```

    "output_licenses": attr.license() if hasattr(attr,
    "license") else attr.string_list(),
},
remove_attrs = ["runtime_deps", "exports",
"exported_plugins"],
)
JAVA_PLUGIN_IMPLICIT_ATTRS = JAVA_LIBRARY_IMPLICIT_ATTRS
java_plugin = rule(
    _proxy,
    attrs = merge_attrs(
        JAVA_PLUGIN_ATTRS,
        JAVA_PLUGIN_IMPLICIT_ATTRS,
    ),
    provides = [JavaPluginInfo],
    outputs = {
        "classjar": "lib{name}.jar",
        "sourcejar": "lib{name}-src.jar",
    },
    fragments = ["java", "cpp"],
    toolchains = [semantics.JAVA_TOOLCHAIN],
)

```

Let us break it down step-by-step:

### 1. Loading dependencies:

These load statements are importing functions, constants, and other symbols from various Starlark (Bazel's extension language) files. They are setting up the necessary tools and information to define and implement the custom rule.

### 2. `bazel_java_plugin_rule` function:

This is a helper function that constructs a Bazel target for the `java_plugin` rule. It primarily utilizes the `basic_java_library` rule to create a standard Java target but customizes it for annotation processing.

### 3. `_proxy` function:

This is the main implementation function for the `java_plugin` rule. Bazel invokes this function when processing the rule. It takes the provided attributes from the rule and forwards them to the `bazel_java_plugin_rule` function, then returns the resulting target.

### 4. `JAVA_PLUGIN_ATTRS` and `JAVA_PLUGIN_IMPLICIT_ATTRS`:

These constants define the attributes that users can (or must) set when they use the `java_plugin` rule in their BUILD files. They are built upon the attributes from the standard `java_library` rule but add, modify, and remove some to fit the needs of a Java plugin.

### 5. `java_plugin` rule definition:

This is where the custom rule is formally defined. It specifies:

- Implementation function: `_proxy`
- Attributes: Merging of the explicit and implicit attributes.
- Outputs: Naming conventions for the generated `.jar` files.
- Other metadata: Like the toolchains it relies on, and the rule fragments it needs.

Here is an explanation of the key concepts:

- **JavaInfo and JavaPluginInfo**: `JavaInfo` encapsulates all Java-specific information about a Bazel target, like its compile-time and runtime dependencies. `JavaPluginInfo` is likely a custom data structure specific to this rule that holds data about the Java annotation processor.
- **Attributes (attrs)** in Bazel allow users to provide parameters to rules. For instance, `srcs` to specify source files, `deps` to declare dependencies, and so on.

- **Neverlink:** If set to True, the rule will generate artefacts that should not be included in the final binary during linking. This is useful for libraries that should only be available at runtime and not embedded in the binary.
- **Toolchains:** These encapsulate the tools (like compilers) and their configurations. The rule relies on the Java toolchain, which provides tools for compiling and linking Java code.
- **Fragments:** Rule fragments allow Bazel to fetch only the necessary parts of the build graph. Here, it uses the “java” fragment (for Java-specific build configurations) and the “cpp” fragment (likely due to some underlying C++ dependencies or configurations).

While not directly a coding aspect, the following example shows how it could be used typically paired with a **java\_library**:

```
java_plugin(  
  name = "my_annotation_processor",  
  processor_class = "com.example.MyProcessor",  
  deps = [":processor_dependencies"],  
)  
  
java_library(  
  name = "use_processor",  
  srcs = ["MyClass.java"],  
  plugins = [":my_annotation_processor"],  
)
```

In this example, when **MyClass.java** is compiled, the **MyProcessor** annotation processor defined in the **java\_plugin** rule is applied.

The **java\_plugin** rule is a representation of Bazel’s powerful, extensible design. By understanding its internal mechanics and design, you learn how to design and code your own skylark rules.

To sum it up, Bazel's rule-driven build system is a medley of speed, accuracy, and adaptability. By distilling the build process into clear, modular, and reusable rules, Bazel promises efficient and trustworthy builds across expansive and diverse codebases. Whether navigating a monolithic codebase at a major tech company or spearheading a burgeoning project in a startup, harnessing Bazel's rules can revolutionize your build and test protocols.

## Customizing Build and Compilation Rules

Bazel's strength lies in its extensibility. While it provides a vast array of predefined rules to cover standard use cases, it is almost inevitable that, at some point, one would need custom rules to cater to unique requirements. Let us dive into customizing build and compilation rules in Bazel.

### **Why Customize?**

The reasons for customization can vary, explained as follows:

- **Non-standard Tools and Languages**

Bazel supports a wide array of languages and tools out-of-the-box, but some niche or proprietary languages or tools may not have existing support.

Suppose your company developed a domain-specific language (DSL) for financial calculations. Bazel does not know about this DSL by default. By creating custom rules, you can instruct Bazel on how to compile, test, and package code written in this DSL.

- **Optimized Build Behavior**

The default behavior might not always be the most optimized for your specific project. Custom rules can reduce build times or resource usage.

Imagine having a monolithic application that undergoes frequent small changes. Instead of rebuilding everything, a custom rule could be developed to intelligently cache and rebuild only the affected components, saving significant time.

- **Code Generation**

Add custom code generators or wrap existing ones into Bazel rules to chain them in the building process.

If you have an API-first approach, you could add code-generation rules to the project hosting the OpenAPI specification so when specs are changed, code for API providers and API consumers are re-generated automatically.

- **Integration with Proprietary Systems**

Businesses sometimes rely on proprietary or legacy systems for version control, dependency management, or deployment. Integrating these systems with Bazel can streamline the development and release process.

If your company uses a proprietary artefact repository, a custom rule can be created to fetch dependencies directly from this repository, ensuring a seamless integration with existing infrastructure.

- **Enforcing Best Practices and Standards**

Custom rules can be used to ensure that the team adheres to specific coding practices, naming conventions, or architectural patterns.

For large teams, you could implement a rule that scans every Java class for a specific annotation. If a class is missing this annotation, the build could fail, ensuring that the codebase maintains a consistent standard.

- **Specific Post-Build or Deployment Actions**

After building or testing, projects might require certain actions like notifications, reporting, or custom



deployments.

For continuous integration (CI) purposes, after running tests, you could customize Bazel to generate a detailed report, push it to a dashboard, and notify the team on Slack about the build and test results.

- **Unique Dependency Mechanisms**

While Bazel handles dependencies efficiently, there might be cases where projects have a unique or complex dependency graph that does not fit the standard mould.

Suppose your application is split across multiple repositories with interdependencies based on the latest commit hashes. Custom rules could fetch the correct versions of these inter-repo dependencies to ensure consistent builds.

## Key Components in Rule Creation

In earlier discussions, we introduced some of the pivotal components of Bazel rules. Now, let us review and introduce additional ones:

- **Rule Definition:** This is the core of any Bazel rule. It is a function that returns a rule object. The rule definition specifies the rule's behavior and how it interacts with other rules.
- **Attributes:** These are the inputs to a rule. They define the dependencies, sources, and other configurations that the rule needs. Common attribute types include:
  - **label:** A reference to another target (for example, a dependency).
  - **label\_list:** A list of labels.
  - **string:** A simple string value.
  - **int:** An integer value.

- `bool`: A boolean value.
- **Implementation Function**: This is where the logic of the rule resides. It is a Starlark function that Bazel invokes when the rule is executed. The function receives a context (`ctx`) object, which provides access to the rule's attributes, outputs, and other utilities.
- **Outputs**: These are the files that the rule produces. In the implementation function, you can define outputs using the `ctx.actions.write` or `ctx.actions.run` methods, among others.
- **Providers**: These are data structures that rules can use to pass information to dependent rules. For example, a rule that compiles C++ code might produce a provider that contains information about the compiled object files.
- **Toolchains**: Toolchains encapsulate the tools (like compilers) and their configurations. They allow rules to be written in a way that is agnostic to the specifics of the underlying tools, making builds more portable across different environments.
- **Aspects**: These are a way to intercept and modify the behavior of other rules. They are useful for cross-cutting concerns like code generation or analysis.
- **Default Values**: For attributes that are not mandatory, you can provide default values. This makes the rule easier to use by reducing the amount of configuration a user needs to provide.
- **Docstrings**: Just like in Python, you can (and should) provide docstrings for your rules and attributes. This makes it easier for other developers to understand and use your custom rules.
- **Private Attributes**: These are attributes that start with an underscore (`_`). They are used for internal details of

the rule, like specifying a tool that the rule uses internally.

- **Mandatory versus Optional Attributes:** While defining attributes, you can specify whether an attribute is mandatory or optional. Mandatory attributes must be provided by the user, while optional ones can be omitted.
- **Rule Visibility:** By default, rules are private to the package they are defined in. You can change this by setting the visibility attribute, allowing other packages to depend on your rule.

Creating custom rules in Bazel involves understanding and combining these components in a way that achieves the desired build or test behavior. As a Bazel developer, mastering these components will enable you to extend Bazel's capabilities and tailor it to your specific needs.

## [Solving any Custom Needs not Served by Default Rules](#)

Bazel, as a versatile build and test tool, comes with a rich set of default rules that cater to a wide range of common development scenarios. However, there might be instances where your project has unique requirements not directly served by these out-of-the-box rules. In such cases, Bazel's extensibility shines, allowing you to craft custom solutions. This section will guide you through the process of addressing any custom needs that are not met by Bazel's default rules.

Before diving into creating custom rules, it is essential to:

- **Identify the Need:** Clearly define what you are trying to achieve. Is it a new type of compilation, a unique packaging requirement, or perhaps a specialized testing scenario?

- **Research Existing Solutions:** Before reinventing the wheel, check if someone in the community has already created a rule or tool that addresses your need. Bazel's community is vast, and often, shared extensions or rules can be found on platforms like GitHub.

If no existing solutions fit your needs, it is time to create your custom rule. Recall the key components of rule creation:

- Rule Definition
- Attributes
- Implementation Function
- Outputs
- Providers, and more

(Refer to the previous section on "*Key Components in Rule Creation*" for a detailed breakdown.)

As you embark on creating a custom rule within Bazel, we would like to share some advice based on my experiences and insights:

- **Start Small:** Begin with a basic version of your rule. For instance, if you are creating a rule to compile a new language, start by handling a single source file without any dependencies.
- **Iterate and Expand:** Once the basic version works, incrementally add features and handle more complex scenarios.
- **Test Thoroughly:** Ensure that your custom rule works in various scenarios. Consider edge cases and ensure compatibility with other rules.
- **Document:** Write clear documentation for your rule, explaining its purpose, attributes, expected inputs, and outputs. This will be invaluable for both your future self and other developers.

Often, when your custom rule relies on external tools or binaries, it is essential to utilize Bazel Toolchains. These toolchains in Bazel allow you to define and configure the tools your rule depends on, ensuring both portability and reproducibility. Additionally, when integrating these external tools, it is crucial to ensure that they do not interfere with other parts of your build, which can be achieved by isolating their effects.

After creating your custom rule, you might want to think about its broader impact. If you believe your rule could benefit others, think about open-sourcing it. The Bazel community is known to thrive on shared extensions and tools. It is also beneficial to engage with this community to gather feedback on your rule, leading to potential improvements and ensuring it has broader compatibility.

Lastly, as Bazel continues to evolve, staying updated becomes paramount. Always keep an eye on Bazel's release notes since changes in the platform might have implications for your custom rules. And if you decide to share your rule with the community, it is essential to clearly indicate which versions of Bazel your rule is compatible with.

Bazel's power lies not just in its default offerings but also in its extensibility. By understanding the framework and following best practices, you can tailor Bazel to any custom need, ensuring efficient and reproducible builds for even the most unique project requirements.

## **Writing and Executing a Genrule**

Bazel's ``genrule`` is a versatile tool that allows developers to generate files as part of the build process. It is a catch-all rule that can be used when no other rule seems to fit the bill. This section will guide you through the process of writing and executing a ``genrule`` in Bazel.

One of the primary advantages of using `genrule` is its flexibility. Since it is not tied to any specific language or toolchain, it can be employed for a wide range of file generation tasks. This universality means that when no other rule seems to fit your requirements, `genrule` often comes to the rescue. Additionally, it offers a straightforward way to integrate custom shell commands into the build process, allowing developers to leverage existing scripts or utilities without the need to write a custom Bazel rule.

On the other hand, this flexibility can also be seen as a double-edged sword. The very nature of `genrule` being a catch-all solution means it might not be optimized for specific tasks. For instance, other specialized rules in Bazel might offer better performance or more features for particular tasks, such as compiling code in a specific language.

Another potential drawback is the portability concern. Commands used in `genrule` are often shell commands, which might behave differently across various systems. This can lead to inconsistencies in builds, especially in cross-platform development environments. Developers need to ensure that the commands they use in `genrule` are portable and yield consistent results across all targeted platforms.

Lastly, while **`genrule`** is excellent for simple file generation tasks, it might not be the best choice for more complex scenarios. As the complexity of the generation logic increases, maintaining it within the confines of a `genrule` command can become challenging. In such cases, writing a custom rule or using a more specialized rule might be a more maintainable approach.

A `genrule` has several key attributes:

- `name`: A unique name for the rule.
- `srcs`: A list of input files that the command uses.
- `outs`: A list of output files that the command produces.

- ``cmd``: The command or script to run. It can reference the source files, output files, and other tools available in the build environment.

Here is a simple example of a ``genrule`` that converts a ``.txt`` file to uppercase and outputs a ``.uppercase.txt`` file:

```
genrule(  
  name = "convert_to_uppercase",  
  srcs = ["input.txt"],  
  outs = ["output.uppercase.txt"],  
  cmd = "tr '[:lower:]' '[:upper:]' < $(SRCS) > $(OUTS)",  
)
```

In this example:

- The ``srcs`` attribute specifies the input file ``.input.txt``.
- The ``outs`` attribute declares the output file ``.output.uppercase.txt``.
- The ``cmd`` attribute uses the ``tr`` command to transform the content of the input file to uppercase and writes it to the output file.

So, let us create a Bazel project to test it :

1. Create a new directory with an empty **WORKSPACE** file:

```
mkdir my_genrule_project  
cd my_genrule_project  
touch WORKSPACE
```

2. Create a **BUILD.bazel** file and copy the rule content shown before.
3. Create a **input.txt** file with a text example including upper and lower cases.
4. To execute the ``genrule``, you would typically use the ``bazel build`` command followed by the target:

```
bazel build :convert_to_uppercase
```

```
INFO: Analyzed target //:convert_to_uppercase (0 packages loaded, 0 targets configured).
INFO: Found 1 target...
Target //:convert_to_uppercase up-to-date:
  bazel-bin/output.uppercase.txt
INFO: Elapsed time: 0.076s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
```

**Figure 3.1:** Console output from executing a bazel build

5. Once executed, Bazel will run the specified command and produce the output file in the `bazel-bin` directory.

```
cat bazel-bin/output.uppercase.txt
```

```
THIS IS A TEXT EXAMPLE TO SHOW A SIMPLE GENRULE FOR CONVERTING A FILE TO UPPERCASE.
```

**Figure 3.2:** Console output from executing the generated binary

These suggestions will assist you in crafting efficient genrules.

1. **Use Bazel's Built-in Variables** Bazel provides several built-in variables like `$(SRCS)` and `$(OUTS)` that you can use in the `cmd` attribute. These variables make it easier to reference input and output files.
2. **Keep It Simple** While `genrule` is powerful, it is best used for simple file generation tasks. For more complex scenarios, consider writing a custom rule or using a more specialized rule.
3. **Ensure Portability** Since `genrule` commands are often shell commands, ensure they are portable across different systems, especially if you are working in a cross-platform environment.

Beyond `$(SRCS)` and `$(OUTS)`, there are other useful built-in variables that can be used within the `cmd` attribute to reference various aspects of the build environment. These variables simplify the process of writing `genrule` commands by allowing developers to dynamically reference inputs, outputs, and other tools. Here is an overview of some of



Bazel's built-in variables that are particularly useful for writing a `genrule`:

- `$(SRCS)`: This variable represents all the source files listed in the `srcs` attribute of the `genrule`. If there are multiple sources, they are space-separated. It is especially useful when the command needs to process all input files without referencing them individually.
- `$(<)`: This variable stands for the first entry in the `srcs` list. It is handy when the `genrule` is designed to work with a single source file. If `OUTS` hold multiple files then it will throw a build error.
- `$(OUTS)`: Represents all the output files listed in the `outs` attribute. Like `$(SRCS)`, if there are multiple outputs, they are space-separated. This variable is essential when the command produces multiple output files and needs to reference them collectively.
- `$(@)`: Refers to the first entry in the `outs` list. It is useful when the `genrule` produces a single output file.
- `$(location label)`: This variable allows you to get the path to a file or rule specified by `label`. It is particularly useful when the `genrule` depends on other targets or needs to reference tools or files not listed in `srcs` or `outs`.
- `$(locations label)`: Similar to `$(location)`, but it returns a space-separated list of paths if the label refers to a rule with multiple outputs.
- `$(GENDIR)`: Refers to the directory where Bazel stores generated files. It can be useful if you need to reference other generated files or directories during the execution of the `genrule`.
- `$(BINDIR)`: Points to the directory where Bazel stores binary files. This can be useful when referencing compiled binaries or tools that are part of the build.

- `$(RULEDIR)`: This variable gives the path to the sandbox directory for the ``genrule``. It can be useful for intermediate processing or temporary file storage during the rule's execution.

Using these built-in variables can significantly streamline the process of writing ``genrule`` commands in Bazel. They ensure that the commands are more readable and maintainable, and they abstract away the specifics of the build environment, making the rules more portable and less prone to errors.

The ``genrule`` in Bazel offers a flexible way to generate files as part of the build process. By understanding its key components and best practices, you can leverage its capabilities to cater to custom file generation needs in your projects.

## **Bazel Configuration**

Bazel is a powerful build and test tool that offers a flexible and extensible system for defining and managing project builds. One of the key strengths of Bazel is its configuration system, which allows developers to customize build behavior to suit their specific needs. In this section, we will dive deep into Bazel's configuration system, exploring its various facets and how developers can leverage them for optimal build performance and flexibility.

Bazel's configuration system is designed to be both powerful and user-friendly. At its core, it revolves around the concept of "*build configurations*", which are sets of flags and settings that determine how Bazel should execute builds and tests.

## **WORKSPACE File**

This file serves as the entry point for Bazel and defines the project's external dependencies. The **WORKSPACE** file resides at the root of your Bazel project. It identifies the directory it resides in as the root of a Bazel workspace, hence the name. This file is used to:

- Define the project's external dependencies.
- Configure various build settings.
- Specify the location of other repositories or sub-projects.

A typical **WORKSPACE** file might look something like this:

```
workspace(name = "my_project")

load("@bazel_tools//tools/build_defs/repo:http.bzl",
      "http_archive")

http_archive(
    name = "dependency_name",
    urls = ["http://example.com/dependency.zip"],
    sha256 = "abcd1234...",
)
```

In this example:

- `workspace(name = "my_project")` gives a name to the workspace.
- The `load` function imports rules from Bazel's built-in tools.
- `http_archive` is a rule that fetches and extracts a zip archive from a URL.

## External Dependencies

Bazel supports various rules to fetch external dependencies:

- `http_archive`: For fetching and extracting zip and tar archives.
- `git_repository`: For checking out a git repository.

- `local_repository`: For referencing local directories as dependencies.

## Adhering to Best Practices

It is imperative to maintain clarity in the `WORKSPACE` file by only incorporating direct dependencies. Bazel, with its efficiency, will autonomously fetch transitive dependencies. Another crucial aspect is version pinning. By consistently pinning dependencies to a specific version using the `sha256` attribute, one ensures reproducibility. For values that are frequently used, such as version numbers, it is advisable to employ a variable at the commencement of the `WORKSPACE` file.

## Venturing into Advanced Topics

Bazel's configuration language, known as Starlark, introduces the possibility of custom logic in the `WORKSPACE` file, proving beneficial for intricate projects. For expansive projects, the `WORKSPACE` file can be fragmented into multiple files utilizing the `load` statement.

## [BUILD File](#)

In previous sections, we delved into the topic of `BUILD` (also known as `BUILD.bazel`) files. In this section, we will introduce some best practices to enhance your understanding:

- **Keep BUILD Files Small:** It is easier to manage and understand smaller `BUILD` files. Consider splitting large ones by functionality or module.
- **Use Fine-grained Targets:** Instead of having one large target, it is often beneficial to have multiple smaller targets. This allows for better caching and parallelism.
- **Avoid Absolute Paths:** Always use relative paths in your `BUILD` files. This ensures portability across different machines and environments.

- **Globbing:** To avoid listing every source file individually, Bazel provides the glob function. It allows you to include all files matching a pattern.

```
cc_library(  
  name = "all-cpp-files",  
  srcs = glob(["*.cc"]),  
)
```

The **BUILD** file is a cornerstone of Bazel's build process. By understanding its structure and best practices, developers can harness the full power of Bazel to create efficient, reliable, and scalable builds.

## Bazel Flags

Flags in Bazel allow developers to modify the behavior of Bazel commands, making the build and test processes more adaptable to various needs.

### **Types of Flags**

There are two primary types of flags in Bazel. The first is the "*Startup Flags*" which affect the Bazel server's behavior. These flags must be specified before the command, as seen in the example

```
bazel --batch build //...
```

The second type is the "*Command Flags*" which influence the behavior of the Bazel command itself. These are specified after the command, demonstrated by

```
bazel build --sandbox_debug //...
```

### **Commonly Used Flags**

- **--config:** This flag allows you to use predefined sets of options, known as configurations.
- **--sandbox\_debug:** If you are facing issues with sandboxing, this flag can help by keeping the sandbox directories instead of deleting them.

- `--jobs`: Specifies the number of concurrent jobs, for example, `--jobs=4` would use 4 cores.
- `--test_output`: Controls the output of test results. Options include `errors`, `all`, and `summary`.
- `--define`: Allows you to override a build variable, for example, `--define ENV=production`.
- `--remote_cache`: Specifies the address of the remote cache, useful for distributed builds.
- `--profile`: Outputs a profile file that can be analyzed for build performance.

### Tips for Using Flags

- **Use Configurations**: Instead of manually specifying a long list of flags every time, use the `--config` flag to define and use configurations in your `.bazelrc` file.
- **Avoid Overloading**: While it is tempting to use many flags for fine-grained control, it can make your build process complex. Use flags judiciously.
- **Documentation**: Always refer to the official Bazel documentation for a comprehensive list of flags and their descriptions.
- **Custom Flags**: Bazel allows you to define custom flags using Starlark. This can be useful for project-specific configurations.

Bazel flags offer a powerful way to customize and control your build and test processes. By understanding and using these flags effectively, developers can harness the full potential of Bazel, ensuring efficient and reproducible builds.

### [.bazelrc File](#)

The `.bazelrc` file provides a way to specify default values for Bazel command-line options. These options can be set at a

global level, for specific commands, or even for specific environments.

## Location

The `.bazelrc` file can be located in:

- The workspace directory.
- The user's home directory as `~/.bazelrc`.
- System-wide (for example, `/etc/bazel.bazelrc`).

Bazel reads these files in the order mentioned, with later files overriding earlier ones.

## Syntax

The syntax for the `.bazelrc` file is straightforward. Each line specifies a command and the options for that command:

```
command:option=value
```

For example:

```
build --cpu=x86_64
```

This sets the `--cpu` option to `x86_64` for the `build` command.

Let us see a more complete example:

```
# Use JDK17
build --java_language_version=17
build --java_runtime_version=remotejdk_17
build --tool_java_language_version=17
build --tool_java_runtime_version=remotejdk_17
build --jobs=4
test --jobs=4
```

Here is an explanation of each line in the file:

- `build --java_language_version=17`: This line sets the Java language version to 17 for all Java compilation actions during the build process.

- `build --java_runtime_version=remotejdk_17`: This line sets the Java runtime version to the remote JDK 17 for all Java targets during the build process.
- `build --tool_java_language_version=17`: This line sets the Java language version to 17 for tools that are used during the build process.
- `build --tool_java_runtime_version=remotejdk_17`: This line sets the Java runtime version to the remote JDK 17 for tools that are used during the build process.
- `build --jobs=4`: This line limits the number of concurrent jobs (actions) that Bazel will run during the build process to 4. This is useful for controlling the resource usage of the build process, especially on machines with limited CPU resources.
- `test --jobs=4`: Similar to the `build --jobs=4` line, this line limits the number of concurrent test actions that Bazel will run to 4.

## Best Practices

- **Environment Specific Configuration**: Use conditional flags to set options based on the environment. For instance, you might want different build configurations for development and production.

```
build:dev --compilation_mode=fastbuild
build:prod --compilation_mode=opt
```

- **Avoid Hardcoding Paths**: If you need to reference paths, use workspace-relative paths or utilize Bazel's built-in `$(location)` expansion.
- **Use Comments**: The `.bazelrc` file supports comments using the `#` symbol. Use comments to explain complex configurations or to temporarily disable certain options.

```
# This is a comment.
# build --option_to_be_disabled
```



- **Shared Configuration:** If you are working in a team, it is beneficial to have a shared `.bazelrc` file in the workspace directory. This ensures consistent builds across different developer machines.
- **User-specific Overrides:** Developers can have their own `~/.bazelrc` for personal overrides. This is useful for settings that are specific to a developer's machine or preferences.
- **Avoid Excessive Configuration:** While the `.bazelrc` file is powerful, avoid over-configuring. Only add options that are necessary for your project.

## Commonly Used Options

- **--config:** This allows you to define a compound set of options in the `.bazelrc` file and then refer to them with a single `--config` flag on the command line.  
`build:myconfig --cpu=x86_64 --compilation_mode=opt`
- **--remote\_cache:** Specifies the address of the remote cache. Useful for distributed builds.
- **--disk\_cache:** Specifies a directory where Bazel can read and write actions' output files.
- **--sandbox\_debug:** Disables sandboxing for easier debugging.

The `.bazelrc` file is a powerful tool in the Bazel ecosystem, allowing developers to fine-tune their build and test processes. By understanding its capabilities and following best practices, teams can achieve consistent and efficient builds across different environments and machines.

## Conclusion

Bazel's configuration system offers developers a robust and flexible framework for defining and managing builds. By

understanding and effectively leveraging this system, developers can achieve faster, more reliable, and more customizable builds, enhancing their development workflow and productivity. Whether you are a seasoned Bazel user or just getting started, a deep understanding of Bazel's configuration system is essential for getting the most out of this powerful tool.

In the upcoming chapter, we will delve into the world of testing strategies in Bazel, equipping you with the knowledge to ensure the reliability and efficiency of your software development process. We will explore a range of topics, from writing and running unit tests in languages like Java, to managing and reporting unit test coverage. Additionally, we will address critical aspects like performance testing, user acceptance tests (end-to-end), and best practices for achieving optimal test isolation and parallelism, empowering you to build robust and scalable test suites within your Bazel-based projects.

## **Recommended Readings**

- Bazel Overview  
<https://docs.bazel.build/versions/main/bazel-overview.html>
- Creating a New Rule  
<https://docs.bazel.build/versions/main/skylark/tutorial-creating-a-rule.html>
- Rules  
<https://docs.bazel.build/versions/main/be/rules.html>
- Writing Bazel Rules  
<https://docs.bazel.build/versions/main/skylark/writing-rules.html>
- Genrule

<https://docs.bazel.build/versions/main/be/general.html#genrule>

- Bazel Configuration

<https://docs.bazel.build/versions/main/skylark/config.html>

- External Dependencies

<https://docs.bazel.build/versions/main/external.html>

- BUILD Files

<https://docs.bazel.build/versions/main/build-ref.html>

- Bazel User Manual

<https://docs.bazel.build/versions/main/user-manual.html>

- Best Practices

<https://docs.bazel.build/versions/main/best-practices.html>

# CHAPTER 4

## Testing Strategies in a Monorepo

### Introduction

In a Monorepo, multiple projects coexist in a single repository, sharing dependencies and tools. Bazel is particularly adept at managing Monorepo complexities. This chapter will explore various testing strategies within a Monorepo environment using Bazel, focusing on efficiency, scalability, and reliability.

### Structure

In this chapter, we will cover the following topics:

- Writing and Running Tests with Bazel
- Achieving Test Isolation and Parallelism

### Testing Strategies in Bazel

Bazel promotes hermetic testing, where tests are isolated and deterministic. Hermetic tests interact minimally with the external environment, reducing flakiness and improving reproducibility. In order to write hermetic tests:

- Minimize reliance on global state and external systems.
- Use mock objects and services to simulate external dependencies.

Bazel supports fine-grained tests, allowing developers to run specific test targets. This granularity facilitates faster feedback loops during development. Define small and focused test targets to take advantage of this feature.

## **Efficient Testing Strategies**

Bazel caches test results, skipping tests whose dependencies have not changed since the last run. This caching mechanism significantly speeds up the testing process in Monorepos. Ensure that tests are deterministic to leverage test caching effectively.

Bazel can execute tests in parallel, distributing test tasks across multiple CPU cores or remote machines. Parallel testing is crucial for Monorepos with extensive test suites, as it reduces the overall test runtime.

## **Scalable Testing Strategies**

For large-scale Monorepos, consider using Bazel's remote execution feature. Remote execution offloads test tasks to a cluster of machines, providing scalability and reducing the load on local resources.

Test sharding splits a test suite into smaller chunks, or shards, that can be run concurrently. Sharding is beneficial for lengthy test suites, as it distributes the testing load and shortens the feedback loop.

## **Reliable Testing Strategies**

Bazel provides tools for identifying and managing flaky tests, which are tests that produce inconsistent results. Address flaky tests promptly to maintain the reliability of your test suite.

Analyze test results systematically to identify patterns and trends in test failures. Bazel offers various options for test result output and analysis, aiding in the quick diagnosis and resolution of issues.

## **Implementing Testing Strategies**

Create custom Bazel test rules to define how tests should be executed and what dependencies they require. Test rules provide a flexible way to configure and run tests in a Bazel-managed Monorepo.

Integrate Bazel with your Continuous Integration (CI) system to automate the testing process. Bazel's efficient and scalable testing features are particularly valuable in a CI environment, where rapid feedback is essential.

Testing in a Monorepo with Bazel requires a strategic approach to handle the unique challenges and opportunities presented by the Monorepo structure. By understanding and implementing the testing strategies outlined in this chapter, developers can ensure efficient, scalable, and reliable testing processes in their Bazel-managed Monorepos.

## [Writing and Running Unit Tests with Bazel](#)

### **Creating a simple Unit Test in Java**

First, let us establish a new Bazel Monorepo to accommodate our class and unit test.

```
mkdir bazel_unit_testing
cd bazel_unit_testing/
mkdir -p app/hellotest/src/main/java/com/hellotest
mkdir -p app/hellotest/src/test/java/com/hellotest
```

Then, let us add the following files:

- **.bazelrc**

```
common --enable_bzlmod

build --java_language_version=11
build --java_runtime_version=remotejdk_11
build --tool_java_language_version=11
build --tool_java_runtime_version=remotejdk_11
```

The `.bazelrc` file with the specified content serves multiple purposes. Firstly, with `common --enable_bzlmod`, it

activates Bazel's module system by enabling the `MODULE.bazel` file to define dependencies. This is crucial for dependency resolution and fetching. Additionally, the file sets the Java language and runtime versions for both the build and tool to 11 using `--java_language_version=11`, `--java_runtime_version=remotejdk_11`, `--tool_java_language_version=11`, and `--tool_java_runtime_version=remotejdk_11`. These configurations ensure that the build process and the tools used during the build are compatible with Java version 11, providing consistency and preventing potential issues related to version discrepancies.

- **.bazelversion**

```
6.2.1
```

- **WORKSPACE**

```
load("@maven//:defs.bzl", "pinned_maven_install")
pinned_maven_install()
```

The `WORKSPACE` file with the content provided is utilized for managing dependencies in a Bazel project. By loading the `defs.bzl` file from the `@maven` repository, it imports the `pinned_maven_install` function. When `pinned_maven_install()` is called, it installs and pins the Maven dependencies for the project. Pinning dependencies means that the specific versions of the dependencies are locked or fixed, ensuring consistency and preventing automatic updates that could break the build. This process is crucial for maintaining a stable and predictable build environment, as it mitigates the risks associated with dependency version changes that might occur with Maven repositories.

- **BUILD.bazel**

```
# empty
```

- **MODULE.bazel**

```
bazel_dep(name = "rules_jvm_external", version = "5.3")
```

```
# To update maven dependencies, update the lines below and
then run:
# bazel run @unpinned_maven//:pin
maven =
use_extension("@rules_jvm_external//:extensions.bzl",
"maven")
maven.install(
  artifacts = [
    "junit:junit:4.13.2",
  ],
  lock_file = "://maven_install.json",
  repositories = [
    "https://maven.google.com",
    "https://repo1.maven.org/maven2",
  ],
)
use_repo(maven, "maven", "unpinned_maven")
```

The `MODULE.bazel` file in question is utilized for dependency management in a Bazel project. Initially, it declares a dependency on `rules_jvm_external` version 5.3 using the `bazel_dep` function, which is essential for working with JVM-based projects. The file then employs the `use_extension` function to load the `maven` extension from `@rules_jvm_external//:extensions.bzl`, which is used to install Maven artifacts. Specifically, it installs the `junit:junit:4.13.2` artifact, as listed in the `artifacts` parameter.

The `lock_file` parameter points to `://maven_install.json`. This file contains all dependencies including the transitive ones and is created by running the `bazel @maven//:pin`, in a format that `rules_jvm_external` can use later. You will check this file into the repository.

The repository's parameter lists two Maven repositories from where the artifacts can be fetched. Finally, the `use_repo` function is called to create a repository named `unpinned_maven` using the previously defined `maven` extension, which can be used later in the Bazel project for building and testing.



- **apps/hellotest/BUILD.bazel**

```
java_library(  
    name = "build",  
    srcs = glob(["src/main/java/**/*.java"]),  
    resources = glob(["src/main/resources/*."]),  
    visibility = ["//visibility:public"],  
    deps = [  
    ],  
)  
  
java_test(  
    name = "test",  
    srcs =  
    ["src/test/java/com/hellotest/HelloTestMainTest.java"],  
    test_class = "com.hellotest.HelloTestMainTest",  
    deps = [  
        ":build",  
    ],  
    runtime_deps = [  
        "@maven//:junit_junit",  
    ],  
    size = "small",  
    resources = glob(["src/test/resources/*."]),  
)
```

The **BUILD.bazel** file in question defines two Bazel build targets: a **java\_library** and a **java\_test**. The **java\_library** target, named **"build"**, compiles Java source files located under **src/main/java** and includes resources from **src/main/resources**. It is made publicly visible to other Bazel targets within the workspace. The **java\_test** target, named **"test"**, specifies a Java test source file located at **src/test/java/com/hellotest/HelloTestMainTest.java** and sets **com.hellotest.HelloTestMainTest** as the test class to be run. This test target depends on the previously defined **"build"** target and the external JUnit library from Maven, with runtime dependencies specified, including JUnit from

Maven. The test size is designated as “small”, and it also includes resources from `src/test/resources`.

- **app/hellotest/src/main/java/com/hellotest/HelloTestMain.java**

```
package com.hellotest;

public class HelloTestMain {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Please provide a name as an
            argument");
            System.exit(1);
        }
        System.out.println(greet(args[0]));
    }

    public static String greet(String name) {
        return "Hello, " + name + "!";
    }
}
```

The `main` method checks if at least one argument is provided through the command line; if not, it prints an error message (“Please provide a name as an argument”) and terminates the program with an exit status of 1. If an argument is provided, the `main` method calls the `greet` method with the provided argument, which then returns a greeting string concatenated with the argument (the name). This greeting string is then printed to the console. For example, if the argument is “John”, the program will output “Hello, John”.

- **app/hellotest/src/test/java/com/hellotest/HelloTestMainTest.java**

```
package com.hellotest;

import org.junit.Test;
import static org.junit.Assert.*;

public class HelloTestMainTest {
    @Test
```

```
public void testGreet() {  
    String result = HelloTestMain.greet("World");  
    assertEquals("Hello, World!", result);  
}  
}
```

Within this file, there is a method named `testGreet` annotated with `@Test`, signifying it as a test method for the JUnit testing framework. The `testGreet` method calls the `greet` function of the `HelloTestMain` class, passing the string "World" as an argument. The returned result from `HelloTestMain.greet("World")` is then compared to the expected string "Hello, World" using the `assertEquals` method from the JUnit framework. If the returned result matches the expected string, the test will pass; otherwise, it will fail.

In subsequent chapters, we will delve into learning about external dependencies in depth.

Prior to launching the application, execute the following at the command line:

```
bazel run @unpinned_maven//:pin
```

This command will resolve and retrieve all the necessary dependencies. Afterward, we can initiate our app test by executing:

```
bazel test //app/hellotest:test
```

The result of this command is shown in [Figure 4.1](#):

```
~/building-large-scale-apps-with-monorepo-and-bazel/chapter-4/bazel_unit_testing (50.828s)
bazel test //app/hellotest:test
Extracting Bazel installation...
Starting local Bazel server and connecting to it...
WARNING: Download from https://maven.google.com/org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar failed
WARNING: Download from https://maven.google.com/org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar failed
WARNING: Download from https://maven.google.com/junit/junit/4.13.2/junit-4.13.2.jar failed
WARNING: Download from https://maven.google.com/junit/junit/4.13.2/junit-4.13.2-sources.jar failed
INFO: Analyzed target //app/hellotest:test (81 packages loaded, 1780 targets configured).
INFO: Found 1 test target...
INFO: From Building app/hellotest/libbuild.jar (1 source file):
warning: [options] system modules path not set in conjunction with -source 11
INFO: From Building external/rules_jvm_external~5.3/private/tools/java/com/github/bazelbuild/rules_jvm_external/external/hamcrest/hamcrest-core.jar (1 source file):
warning: [options] system modules path not set in conjunction with -source 11
INFO: From Building app/hellotest/test.jar (1 source file):
warning: [options] system modules path not set in conjunction with -source 11
INFO: From Building external/rules_jvm_external~5.3/private/tools/java/com/github/bazelbuild/rules_jvm_external/external/junit/junit.jar (1 source file):
warning: [options] system modules path not set in conjunction with -source 11
Target //app/hellotest:test up-to-date:
  bazel-bin/app/hellotest/test.jar
  bazel-bin/app/hellotest/test
INFO: Elapsed time: 50.613s, Critical Path: 5.26s
INFO: 27 processes: 9 internal, 13 darwin-sandbox, 5 worker.
INFO: Build completed successfully, 27 total actions
//app/hellotest:test PASSED in 0.6s

Executed 1 out of 1 test: 1 test passes.
```

**Figure 4.1:** Test execution for hellotest project

By the way, the **WARNING** logs visible here occur because certain libraries could not be located at `maven.google.com`, though they were resolved from alternative source `https://repo1.maven.org/maven2` as defined in `MODULE.bazel`.

To view a test log, simply open it with your preferred editor by navigating to `$(bazel info bazel-testlogs)/app/hellotest/test/test.log`, or you can display its contents directly in the terminal using the `cat` command:

```
cat $(bazel info bazel-testlogs)/app/hellotest/test/test.log
```

In this example, the command `bazel info bazel-testlogs` retrieves the path where test logs are stored, to which you append the relative path to the specific test log you wish to view.

## Managing Multiple Unit Tests

The previous approach works for managing one unit test only, and requires a new target for each unit test we want to add. So, this is not scalable. If we want to execute many unit tests, we need a TestSuite.

A JUnit Test Suite is a collection of test cases that are bundled together to be executed as a group. Test suites are an excellent tool when you have multiple test classes and you want to organize, manage, and run them together. In JUnit, you can create a test suite using the `@RunWith` and `@Suite` annotations. The `@RunWith(Suite.class)` annotation tells JUnit to run the class as a suite, while the `@Suite.SuiteClasses({TestClass1.class, TestClass2.class})` annotation holds an array of test classes that will be part of the suite. When the suite is run, it will execute all test methods in all test classes listed in the `@Suite.SuiteClasses` annotation, allowing for more organized and efficient testing, especially in large projects with numerous test classes.

```
// TestSuiteExample.java
package com.example;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    Test1.class,
    Test2.class
})
public class TestSuiteExample {
    // the class remains empty,
    // used only as a holder for the above annotations
}
```

However, manually maintaining these TestSuites is not scalable. Would it not be more efficient if we could automate the generation of TestSuite code? Furthermore, imagine the convenience if we encapsulate this automated process into a Bazel rule. Let us proceed by creating the following files:

- `tools/rules/junit4_test_suite` folder including an empty `BUILD.bazel` file just to let Bazel consider it as a project
- `tools/rules/junit4_test_suite.bzl` including this content:

```
load("@rules_java//java:defs.bzl", "java_test")

_TEST_SUITE_TEMPLATE = """import org.junit.runners.Suite;
import org.junit.runner.RunWith;
@RunWith(Suite.class)
@Suite.SuiteClasses({%s})
public class %s {}
"""

_VALID_PACKAGE_PREFIX = ("org", "com", "edu")

def _GetIndex(l, val):
    for i, v in enumerate(l):
        if val == v:
            return i
    return -1

def _GetClassName(fname):
    fname = [x.path for x in fname.files.to_list()][0]
    toks = fname[:-5].split("/")
    findex = -1
    for s in _VALID_PACKAGE_PREFIX:
        findex = _GetIndex(toks, s)
        if findex != -1:
            break
    if findex == -1:
        fail("%s does not contain any of %s" % (fname,
            _VALID_PACKAGE_PREFIX))
    return ".".join(toks[findex:]) + ".class"

def _impl(ctx):
    classes = ",".join(
        [_GetClassName(x) for x in ctx.attr.srcs],
    )
    ctx.actions.write(output = ctx.outputs.out, content =
        _TEST_SUITE_TEMPLATE % (
            classes,
```

```

        ctx.attr.outname,
    ))
    _GenSuite = rule(
        attrs = {
            "srcs": attr.label_list(allow_files = True),
            "outname": attr.string(),
        },
        outputs = {"out": "%{name}.java"},
        implementation = _impl,
    )
def junit4_test_suite(name, srcs, **kwargs):
    s_name = name.replace("-", "_") + "TestSuite"
    _GenSuite(
        name = s_name,
        srcs = srcs,
        outname = s_name,
    )
    java_test(
        name = name,
        test_class = s_name,
        srcs = srcs + [":" + s_name],
        **kwargs
    )

```

It begins by loading the `java_test` rule from `@rules_java//java:defs.bzl`. The file contains a template string `_TEST_SUITE_TEMPLATE` for generating Java code that imports the necessary JUnit classes and annotations to define a test suite. It also defines a tuple `_VALID_PACKAGE_PREFIX` containing valid package prefixes. The file contains functions `_GetIndex` and `_GetClassName` to manipulate and retrieve the class name from the provided file names. The `_impl` function generates the test suite Java code by iterating over the source files, retrieving their class names, and inserting them into the template. This function writes the generated code to an output file. The `_GenSuite` rule is defined to implement the `_impl` function, taking source files and an output name as attributes. Finally,

the `junit4_test_suite` macro is defined to create a test suite. It generates a test suite name, invokes `_GenSuite` to generate the test suite Java code, and then defines a `java_test` target using the generated test suite and the provided source files.

- Let us replace `app/hellotest/BUILD.bazel` content with this:

```
load("//tools/rules:junit4_test_suite.bzl",
      "junit4_test_suite")

java_library(
    name = "build",
    srcs = glob(["src/main/java/**/*.java"]),
    resources = glob(["src/main/resources/*."]),
    visibility = ["//visibility:public"],
    deps = [
    ],
)

junit4_test_suite(
    name = "test",
    srcs = glob(["src/test/java/**/*.java"]),
    deps = [
        ":build",
    ],
    runtime_deps = [
        "@maven//:junit_junit",
    ],
    size = "small",
    resources = glob(["src/test/resources/*."]),
)
```

So, here we are using the previously created `junit4_test_suite` rule to redefine the `test` target.

Now is time to refactor our app to include two classes and two tests.

- `app/hellotest/src/main/java/com/hellotest/GreeterService.java`

```
a
package com.hellotest;
```



```
public class GreeterService {
    public String greet(String name) {
        return "Hello, " + name + "!";
    }
}
```

- **app/hellotest/src/main/java/com/hellotest/HelloTestMain.java**

```
package com.hellotest;

public class HelloTestMain {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Please provide a name as an
            argument");
            return;
        }
        GreeterService service = new GreeterService();
        System.out.println(service.greet(args[0]));
    }
}
```

- **app/hellotest/src/test/java/com/hellotest/GreeterServiceTest.java**

```
package com.hellotest;

import org.junit.Test;
import static org.junit.Assert.*;

public class GreeterServiceTest {
    @Test
    public void testGreet() {
        GreeterService service = new GreeterService();
        String result = service.greet("World");
        assertEquals("Hello, World!", result);
    }
}
```

- **app/hellotest/src/test/java/com/hellotest/HelloTestMainTest.java**

```
package com.hellotest;
```

```

import org.junit.Test;
import static org.junit.Assert.*;

public class HelloTestMainTest {
    @Test
    public void testGreet() {
        HelloTestMain.main(new String[]{"World"});
    }

    @Test
    public void testEmptyGreet() {
        HelloTestMain.main(new String[]{});
    }

    @Test
    public void testInstance() {
        HelloTestMain main = new HelloTestMain();
    }
}

```

Afterwards, we can initiate our app test by executing:

```
bazel test //app/hellotest:test
```

Refer to [Figure 4.2](#), for app test initiation.

```

~/building-large-scale-apps-with-monorepo-and-bazel/chapter-4/bazel_unit_test_suite (38.443s)
bazel test //app/hellotest:test
Starting local Bazel server and connecting to it...
INFO: Analyzed target //app/hellotest:test (81 packages loaded, 1783 targets configured).
INFO: Found 1 test target...
INFO: From Building app/hellotest/test.jar (3 source files):
warning: [options] system modules path not set in conjunction with -source 11
INFO: From Building external/rules_jvm_external~5.3/private/tools/java/com/github/bazelbuild/
warning: [options] system modules path not set in conjunction with -source 11
INFO: From Building app/hellotest/libbuild.jar (2 source files):
warning: [options] system modules path not set in conjunction with -source 11
INFO: From Building external/rules_jvm_external~5.3/private/tools/java/com/github/bazelbuild/
warning: [options] system modules path not set in conjunction with -source 11
Target //app/hellotest:test up-to-date:
  bazel-bin/app/hellotest/test.jar
  bazel-bin/app/hellotest/test
INFO: Elapsed time: 38.292s, Critical Path: 5.42s
INFO: 28 processes: 10 internal, 13 darwin-sandbox, 5 worker.
INFO: Build completed successfully, 28 total actions
//app/hellotest:test PASSED in 0.4s

Executed 1 out of 1 test: 1 test passes.

```

**Figure 4.2:** Test execution for hellotest project

On the console, the target execution result is displayed as PASSED. However, behind the scenes, the two unit test classes and their four methods have been executed.

```
cat $(bazel info bazel-testlogs)/app/hellotest/test/test.log
Starting local Bazel server and connecting to it...
exec ${PAGER:-/usr/bin/less} "$@" || exit 1
Executing tests from //app/hellotest:test
-----
---
JUnit4 Test Runner
.GreeterServiceTest
.HelloTestMainTest
Please provide a name as an argument
.HelloTestMainTest
Hello, World!
.HelloTestMainTest
Time: 0.018
OK (4 tests)
```

## Reporting Unit Test Coverage

Code coverage is a metric that helps developers understand the percentage of their codebase that is covered by their unit tests. It provides insights into areas of the code that might be at risk due to lack of testing. In a Monorepo setup with Bazel, reporting unit test coverage for Java unit tests can be streamlined and efficient. This section will guide you through the process of generating and interpreting these reports.

### **Prerequisites**

Before diving into the coverage reporting, ensure you have `lcov` installed:

```
lcov --version
```

`lcov` is a graphical front-end for GCC's coverage testing tool `gcov`. It provides a comprehensive way to visualize code coverage information, making it easier for developers to

identify areas in their codebase that are not well-tested. `lcov` collects coverage data from `gcov` and generates HTML pages that offer a user-friendly representation of the coverage information, including line, function, and branch coverage metrics.

To install `lcov` on a Linux-based system, you can typically use the package manager associated with your distribution. For example, on a system that uses `apt` (like Ubuntu), you would run

```
sudo apt-get install lcov
```

Or in MacOS you could run

```
brew install lcov
```

For other operating systems or package managers, the installation command might differ, so it is advisable to refer to the official documentation or relevant resources for specific installation instructions.

## **Generating Coverage Reports with Bazel**

Follow these steps to generate coverage reports for your Java unit tests:

### **1. Run Bazel with the Coverage Option**

Use the `bazel coverage` command followed by the target you want to test. For instance, if you have a target named `//my-java-project:all`, you would run:

```
bazel coverage //my-java-project:all
```

### **2. Specify a Coverage Tool**

Bazel supports multiple coverage collection tools. For Java, the default tool is `lcov`. Ensure you have `lcov` installed or specify another tool using the `--coverage_report_generator` flag.

```
bazel coverage --combined_report=lcov //app/hellotest:test
```

### **3. Generate the Report**

Once the coverage command completes, Bazel will produce a coverage report in the `bazel-out/` directory. This

report is in the `lcov` format, which can be converted to other formats or viewed using various tools.

## Interpreting the Coverage Report

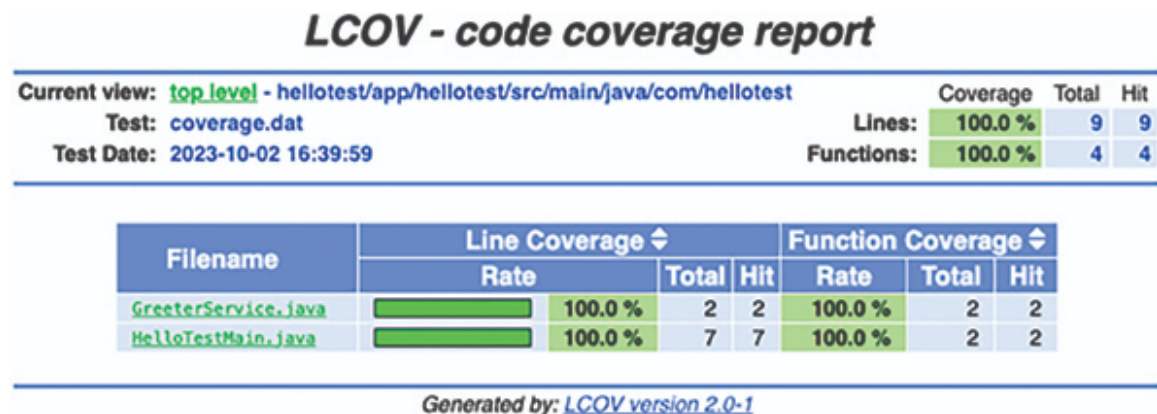
- **Viewing the LCOV Report**

You can view the `lcov` report using a tool like `genhtml`:

```
genhtml -o output_directory bazel-  
out/_coverage/_coverage_report.dat
```

This will generate an HTML report in the specified output directory.

```
open genhtml/index.html
```



**Figure 4.3:** Generated test coverage report

- **Understanding the Metrics**

- **Line Coverage:** The percentage of lines of code that were executed by the tests.
- **Function Coverage:** The percentage of functions or methods that were called during the tests.
- **Branch Coverage:** The percentage of branches (like `if` or `switch` statements) that were tested.

- **Identifying Areas for Improvement:** Look for classes or methods with low coverage percentages. These are areas where you might want to write additional tests or review existing tests for completeness.

## Best Practices

- **Continuous Integration (CI):** Integrate coverage reporting into your CI pipeline. This ensures that coverage is consistently monitored and reported with every code change.
- **Coverage Thresholds:** Set minimum coverage thresholds and fail the build if coverage drops below these thresholds. This ensures that the codebase maintains a certain level of test coverage.
- **Review Coverage Reports Regularly:** Make it a habit to review coverage reports regularly, especially after significant code changes.

Coverage reporting is an essential aspect of maintaining a healthy and robust codebase, especially in a Monorepo setup with Bazel. By understanding how to generate and interpret these reports, you can ensure that your Java projects are well-tested and resilient against regressions. Remember, while high coverage is a good indicator, it is the quality of the tests that truly matters. Aim for meaningful tests that validate the functionality of your code, rather than just chasing high coverage numbers.

## Performance Testing

Performance testing is a crucial aspect of software development that ensures your Java applications run efficiently and effectively under specified workloads. Bazel, a fast and reliable build tool, is an excellent choice for managing and running performance tests on Java projects due to its advanced caching, parallel execution features, and fine-grained build graph.

Tools like JMH (Java Microbenchmarking Harness) can be integrated with Bazel for microbenchmarking Java code.

Let us copy our previous project and add JMH dependency by editing:

- **MODULE.bazel**

```
...
maven.install(
  artifacts = [
    ...
    "org.openjdk.jmh:jmh-core:1.23",
    "org.openjdk.jmh:jmh-generator-annprocess:1.23",
  ],
  ...
)
...
```

- Update the dependencies by running:

```
bazel run @unpinned_maven//:pin
```

- Add JMH java plugin and a new benchmark target in the BUILD.bazel

```
load("//tools/rules:junit4_test_suite.bzl",
      "junit4_test_suite")

java_library(
  name = "build",
  srcs = glob([
    "src/main/java/**/*.java",
    "src/test/java/**/*Benchmark.java",
  ]),
  resources = glob(["src/main/resources/*.*"]),
  visibility = ["//visibility:public"],
  deps = [
    "@maven//:org_openjdk_jmh_jmh_core",
  ],
  plugins = [":jmh_annotation_processor"],
)
...

java_binary(
  name = "benchmark",
  main_class = "org.openjdk.jmh.Main",
  runtime_deps = [":build"],
```

```

    plugins = [":jmh_annotation_processor"],
  )
  java_plugin(
    name = "jmh_annotation_processor",
    deps =
      ["@maven//:org_openjdk_jmh_jmh_generator_annprocess"],
    processor_class =
      "org.openjdk.jmh.generators.BenchmarkProcessor",
    visibility = ["//visibility:private"],
  )

```

- Let us add a new java class at

```

app/hellotest/src/test/java/com/hellotest/GreeterServiceBenchmark.java

```

```

package com.hellotest;

import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.State;

@State(Scope.Thread)
public class GreeterServiceBenchmark {

    @Param({"Alice", "Bob", "Charlie", "Diana"})
    public String name;

    private final GreeterService greeterService = new
    GreeterService();

    @Benchmark
    public void greetBenchmark() {
        greeterService.greet(name);
    }
}

```

- Now it is time to launch it by running:

```

bazel run //app/hellotest:benchmark

```



```
bazel run //app/hellotest:benchmark
Starting local Bazel server and connecting to it...
INFO: Analyzed target //app/hellotest:benchmark (84 packages loaded, 1554 targets configured).
INFO: Found 1 target...
Target //app/hellotest:benchmark up-to-date:
  bazel-bin/app/hellotest/benchmark.jar
  bazel-bin/app/hellotest/benchmark
INFO: Elapsed time: 30.171s, Critical Path: 0.51s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Running command line: bazel-bin/app/hellotest/benchmark
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.openjdk.jmh.util.Utils (file:/private/var/tmp/_bazel_jrai/5.3-maven-maven/org/openjdk/jmh/jmh-core/1.23/processed_jmh-core-1.23.jar) to field java.io.0
WARNING: Please consider reporting this to the maintainers of org.openjdk.jmh.util.Utils
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
# JMH version: 1.23
# VM version: JDK 11.0.15, OpenJDK 64-Bit Server VM, 11.0.15+10-LTS
# VM invoker: /private/var/tmp/_bazel_jrai/70b94bcda407a2e6e835e5fa68034124/external/rules_java-
# VM options: <none>
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: com.hellotest.GreeterServiceBenchmark.greetBenchmark
# Parameters: (name = Alice)
```

**Figure 4.4:** Benchmark execution for hellotest project

JMH is a comprehensive performance benchmarking tool that exceeds the coverage of this book. For a more in-depth understanding, it is strongly recommended referring to their official documentation.

## **User Acceptance Tests (End-to-end)**

User Acceptance Testing (UAT) is a pivotal phase in the software development lifecycle, ensuring that the developed software meets user expectations and requirements. In the context of a Monorepo and utilizing Bazel as a build and test tool, UAT can be orchestrated efficiently to validate that the applications and libraries are functioning as intended in an integrated environment. This chapter will delve into the intricacies of setting up and running UATs with Bazel in a Monorepo setup.

In a Monorepo, multiple projects coexist in a single repository, sharing dependencies and tools, which can be leveraged to

create a unified and streamlined UAT process. User Acceptance Tests in this context should validate:

- **Interoperability:** Ensure that shared libraries and services function correctly across all applications in the Monorepo.
- **Consistency:** Validate that common components provide consistent behavior and UI across different applications.
- **Dependency Management:** Confirm that changes in shared dependencies do not adversely affect applications.

Integrate Bazel-driven UATs into your Continuous Integration/Continuous Deployment (CI/CD) pipelines to ensure that UAT is consistently performed with every change to the Monorepo. This ensures that any regression or issues are identified and addressed promptly.

### **Challenges and Best Practices**

- **UAT Standalone Application:** This method involves handling UAT tests as a separate, independent application that produces its own executable binary. This allows it to run across various environments. Additionally, this UAT application can be launched with particular parameters to run either all tests or a selected group of them. Such flexibility is advantageous for specific situations such as smoke testing, performance evaluations, and more.
- **Flakiness:** Address test flakiness by ensuring that tests are deterministic and robust against minor UI/UX changes.
- **Data Management:** Ensure that test data is anonymized and does not contain sensitive information.
- **Scalability:** As the Monorepo grows, optimize Bazel configurations to ensure that UAT execution remains efficient and fast.
- **Maintenance:** Regularly review and update UATs to align with evolving user requirements and application features.

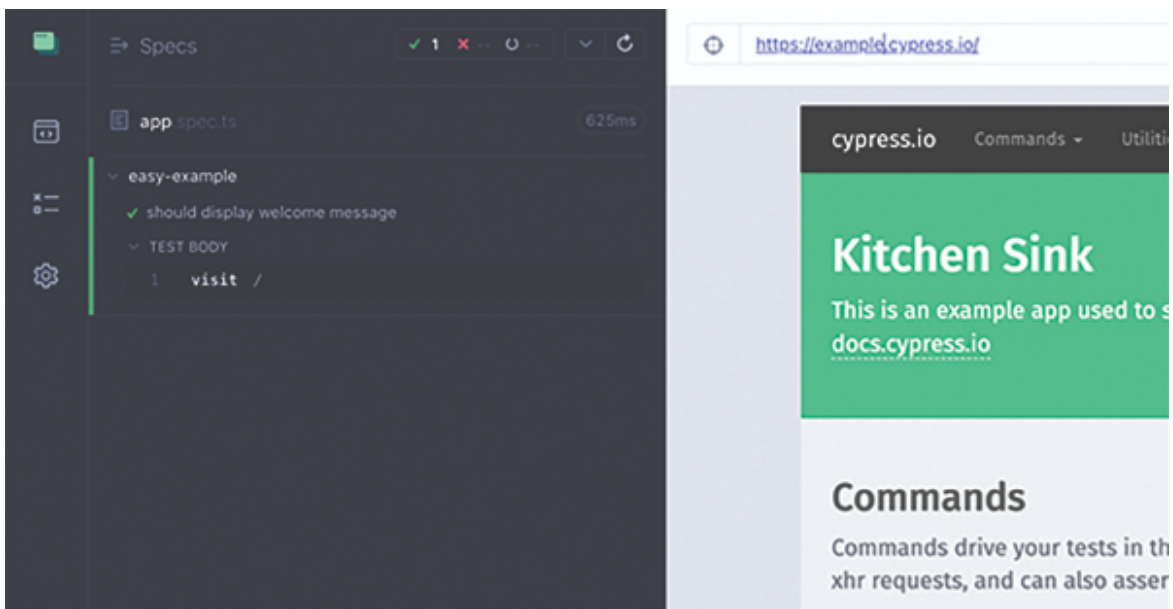
## Using Cypress

We have added a simple UAT sample using Cypress to the book's GitHub repository under the path `/chapter-4/bazel_uat_cypress`. This plain easy example only opens an example website at <https://example.cypress.io/>.

You could run it by executing:

```
bazel run //apps/frontend-e2e:cypress_dev
```

You are going to see the report at your preferred web browser, as shown in [Figure 4.5](#):



**Figure 4.5:** End-to-end test execution report

And at the command line:

```
bazel run //apps/frontend-e2e:cypress_dev
INFO: Analyzed target //apps/frontend-e2e:cypress_dev (0 packages loaded, 0 targets configured).
INFO: Found 1 target...
Target //apps/frontend-e2e:cypress_dev up-to-date:
  bazel-bin/apps/frontend-e2e/cypress_dev.sh
INFO: Elapsed time: 0.203s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Running command line: bazel-bin/apps/frontend-e2e/cypress_dev.sh open

Starting js_run_devserver //apps/frontend-e2e:cypress_dev
Syncing...
12 files synced in 8 ms
Running '../..node_modules/.bin/cypress open' in /var/folders/dq/5tkn7knx5ylc1yj3jhlng55xpwq_m/

GET /_/ 200 1.821 ms - -
GET /_/assets/index.082d7324.css 200 1.572 ms - -
GET /_/assets/polyfills.f25d072f.js 200 1.933 ms - -
GET /_/assets/index.0c6d2dd2.js 200 1.817 ms - -
GET /_/cypress/runner/cypress_runner.css 200 0.699 ms - -
GET /_/assets/Specs.c24a87b4.js 200 3.381 ms - 485
GET /_/assets/Index.014905b8.css 200 2.767 ms - -
GET /_/assets/Index.38585ba0.js 200 2.929 ms - -
GET /_/assets/CreateSpecModal.315eb5ce.css 200 1.285 ms - 368
GET /_/assets/ResultCounts.b6f074af.js 200 1.283 ms - -
GET /_/assets/user-outline_x16.e44b33a8.js 200 1.156 ms - -
GET /_/assets/SpecPatterns.5e3237c4.js 200 1.329 ms - -
```

**Figure 4.6:** End-to-end test execution

UAT is crucial in affirming that the developed software aligns with user expectations and requirements. In a Monorepo, utilizing Bazel to orchestrate UATs ensures efficiency, consistency, and reliability in validating that all applications and libraries function cohesively and as intended. By integrating UATs into the development and deployment workflows, teams can ensure that user satisfaction and product quality are perpetually at the forefront.

## **Achieving Test Isolation and Parallelism**

In the world of software development, speed and reliability are paramount. As projects grow, so does the time it takes to test and build them. Bazel, with its unique approach to dependency management and build optimization, offers a solution to this challenge. One of the standout features of Bazel is its ability to achieve test isolation and parallelism,

ensuring that tests run quickly and reliably. This section delves into how Bazel accomplishes this and how you can leverage its capabilities to supercharge your testing process.

## Test Isolation

Test isolation ensures that each test runs in a controlled environment, unaffected by other tests or external factors. This is crucial for the reliability of tests, as it ensures that they only fail when there is an actual problem with the code.

### **How Bazel Ensures Test Isolation:**

1. **Sandboxing:** Bazel runs each test in a sandboxed environment. This means that tests cannot access files or network resources outside of their declared dependencies. This prevents tests from inadvertently depending on external state.
2. **Explicit Dependency Declaration:** In Bazel, you must declare all dependencies explicitly. This ensures that each test has everything it needs to run and nothing more. It also means that Bazel can accurately determine which tests need to be rerun when a dependency changes.
3. **Hermeticity:** Bazel's emphasis on hermetic builds (builds that are isolated from the external environment) ensures that tests are repeatable. The same test will produce the same result, regardless of where and when it is run.

## Test Parallelism

Parallelism involves running multiple tests simultaneously to reduce the total test runtime. Bazel's approach to parallelism is both efficient and flexible.

### **How Bazel Achieves Test Parallelism:**

1. **Fine-grained Dependency Analysis:** Bazel's dependency graph is at the core of its parallelism capabilities. By understanding the relationships between

different parts of the codebase, Bazel can determine which tests can be run in parallel without interfering with each other.

2. **Dynamic Test Scheduling:** Bazel dynamically schedules tests based on the available resources. If you have a multi-core machine or a distributed build system, Bazel will take full advantage of it to run as many tests in parallel as possible.
3. **Remote Execution and Caching:** Bazel can offload test execution to remote machines, allowing for even greater parallelism. Combined with Bazel's caching mechanism, this means that previously run tests (with unchanged dependencies) can be skipped entirely, further speeding up the testing process.

## **Best Practices for Maximizing Test Isolation and Parallelism**

- **Keep Tests Small and Focused:** The smaller and more focused a test is, the faster it will run, and the easier it will be to parallelize with other tests.
- **Avoid Global State:** Global state can introduce dependencies between tests that Bazel cannot account for. Always aim for stateless tests.
- **Use Bazel's Test Tags:** Bazel allows you to tag tests (for example, ``integration``, ``unit``, ``slow``). You can use these tags to group and manage your tests, ensuring that they are scheduled appropriately.
- **Monitor and Optimize:** Regularly monitor your test runtimes and look for opportunities to optimize. Bazel provides tools and reports that can help identify bottlenecks in your testing process.

## **Conclusion**

In this chapter, we learned that achieving test isolation and parallelism is crucial for maintaining a fast and reliable testing

process, especially as your codebase grows. Bazel, with its advanced dependency analysis and dynamic scheduling, offers a robust solution to this challenge. By understanding and leveraging Bazel's capabilities, you can ensure that your tests are both reliable and efficient.

In the next chapter, we will cover the knowledge and skills required to proficiently manage both internal and external dependencies, master the utilization of Bazel workspaces for defining and managing dependencies, enforce versioning and compatibility strategies within a Monorepo, and seamlessly integrate Bazel into various IDEs (integrated development environments).

## **Recommended Readings**

- Test encyclopedia | Bazel:  
<https://bazel.build/reference/test-encyclopedia>
- How to Set Up a Bazel Testing Configuration: The Comprehensive Guide for Scala and Java  
<https://virtuslab.com/blog/bazel-testing-configuration-comprehensive-guide/>
- Bazel: Testing  
<https://docs.bazel.build/versions/main/test.html>
- Bazel: Best Practices for Rules and Tests  
<https://docs.bazel.build/versions/main/best-practices.html>
- Efficient Testing with Bazel  
<https://blog.bazel.build/2019/06/07/more-efficient-test-discovery.html>
- Scaling Your Testing with Bazel  
[https://www.youtube.com/watch?v=2xqkF\\_1t-6E](https://www.youtube.com/watch?v=2xqkF_1t-6E)
- Bazel: Persistent Test Workers  
<https://docs.bazel.build/versions/main/persistent-workers.html>

- Implementing Bazel Test Targets  
[https://docs.bazel.build/versions/main/be/general.html#test\\_suite](https://docs.bazel.build/versions/main/be/general.html#test_suite)
- Writing Bazel Tests  
<https://docs.bazel.build/versions/main/test-encyclopedia.html>
- Java Testing with Bazel  
<https://docs.bazel.build/versions/main/tutorial/java.html>
- Managing Dependencies in Bazel  
<https://docs.bazel.build/versions/main/build-ref.html#dependencies>
- Bazel: Code Coverage  
<https://docs.bazel.build/versions/main/test-encyclopedia.html#code-coverage>
- Performance Testing with Bazel  
<https://docs.bazel.build/versions/main/skylark/performance.html>
- End-to-End Testing Frameworks Compatible with Bazel  
[https://bazelbuild.github.io/rules\\_nodejs/examples/web\\_testing.html](https://bazelbuild.github.io/rules_nodejs/examples/web_testing.html)
- Test Isolation in Bazel  
<https://docs.bazel.build/versions/main/test-encyclopedia.html>
- Bazel: Parallel Test Execution  
<https://docs.bazel.build/versions/main/test-encyclopedia.html#test-execution-environment>
- Best Practices in Bazel for Test Isolation  
<https://docs.bazel.build/versions/main/best-practices.html>



## **CHAPTER 5**

# **Dependency Management and Versioning**

### **Introduction**

In the ever-evolving landscape of software development, the management of dependencies, both internal and external, has emerged as a critical aspect of project success. This chapter delves into the intricacies of managing these dependencies within the Bazel ecosystem. We begin by exploring the nuances of internal dependencies - those which are part of your monorepo, and external dependencies - components sourced from outside your repository. Understanding how to effectively balance and manage these dependencies is essential for maintaining a robust and efficient build system. The chapter will guide you through the complexities of conflict resolution, ensuring that your project remains stable and consistent amidst the myriad of library versions and external modules.

As we progress, we will uncover the innovative approach of Bazel MODULES, a modern solution for handling external dependencies in Bazel. This section will provide detailed insights into declaring dependencies with MODULES and configuring an air-gapped Bazel build - a crucial aspect for secure, offline development environments. Furthermore, we will delve into the strategies for enforcing versioning and compatibility in a monorepo setting, which is key to maintaining a coherent codebase over time. Additionally, this chapter covers the practicalities of querying dependencies and generating dependency graphs, providing you with the tools to visualize and understand the intricate web of your project's

dependencies. To ensure a comprehensive learning experience, we will also explore how to integrate Bazel into various Integrated Development Environments (IDEs), enhancing your development workflow. Concluding with a summary and recommended readings, this chapter aims to equip you, the software developer, with the knowledge and skills to master dependency management and versioning in Bazel, paving the way for more efficient and streamlined software development processes.

## **Structure**

In this chapter, we will discuss the following topics:

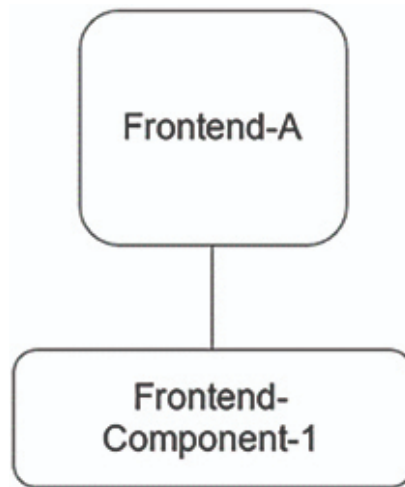
- Managing Internal and External Dependencies
- Bazel Workspaces: Defining and Handling Dependencies
- Enforcing Versioning and Compatibility in a Monorepo
- Target and Package Visibility
- Integrating Bazel within an IDE

## **Managing Internal and External Dependencies**

In the world of software development, dependency management is a crucial aspect that ensures the seamless integration and functioning of various components. When working with a monorepo and Bazel, understanding how to manage both internal and external dependencies becomes even more vital. This chapter will delve into the intricacies of managing these dependencies, offering best practices and insights.

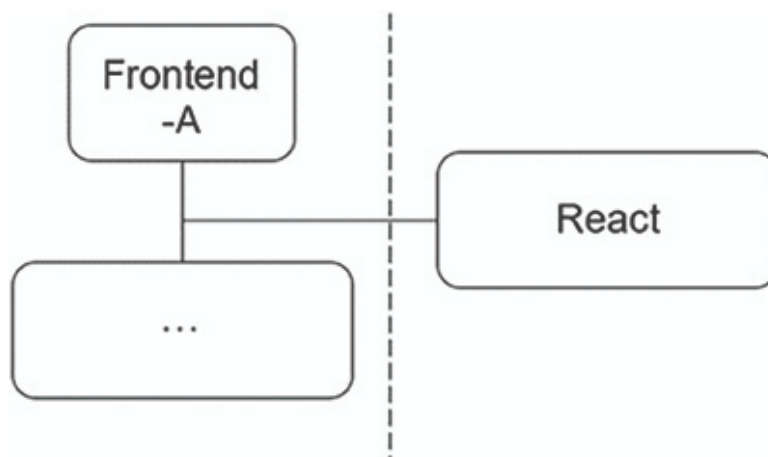
Internal dependencies refer to the relationships between different parts of your codebase within the monorepo. For instance, if you have a library `Frontend-Component-1` and an application `Frontend-A` in the same monorepo, and `Frontend-A`

uses Frontend-Component-1, then Frontend-A has an internal dependency on Frontend-Component-1.



**Figure 5.1:** Internal dependency

External dependencies are the libraries, frameworks, or tools that your codebase relies on but are not part of your monorepo. Examples include third-party libraries like React, Lodash, or TensorFlow.



**Figure 5.2:** External dependency

## **Internal Dependencies**

A monorepo, by definition, houses multiple projects, libraries, and tools under a single repository. This structure offers several benefits, such as code sharing, atomic commits, and

simplified dependency management. However, it also presents unique challenges, especially when it comes to managing dependencies between internal projects.

Before diving into management techniques, it's essential to understand what internal dependencies are. In a monorepo, internal dependencies refer to the reliance of one project or module on another within the same repository. For instance, a web application might depend on a shared utility library, or a microservice might rely on a common API definition.

Some of the benefits of managing internal dependencies in a monorepo are as follows:

- **Code Reusability:** Properly managed dependencies allow for code to be reused across multiple projects, reducing redundancy and improving the developer experience compared with a multi-repo scenario.
- **Consistency:** Shared libraries ensure that common functionalities are consistent across projects.
- **Efficient Builds:** With tools like Bazel, only the changed parts of the codebase and their direct dependencies need to be rebuilt, leading to faster build times.

Bazel's primary strength lies in its ability to understand the dependency graph of your projects. Here's how Bazel aids in managing internal dependencies:

- **BUILD Files:** Every project or library in a Bazel-managed monorepo has a **BUILD** file. This file defines the project's dependencies, both internal and external. By explicitly stating dependencies, Bazel can determine what needs to be rebuilt when a change occurs.
- **Fine-grained Dependencies:** Instead of depending on an entire project, Bazel allows projects to depend on specific targets (for example, a particular library or binary). This granularity ensures that builds are as efficient as possible.

- **Visibility Controls:** Bazel's `BUILD` files allow you to specify who can depend on a particular target using the visibility attribute. This feature ensures that certain libraries remain private to specific projects, enforcing modular design.

## Best Practices

Here are some pivotal strategies to consider during internal dependency management:

- **Explicit Dependency Declaration:** It's imperative to specify all dependencies in your `BUILD` files. Depending on indirect or transitive dependencies can compromise the robustness of your builds.
- **Controlled Visibility:** Harness the power of Bazel's visibility features to guarantee that projects hinge solely on the essentials. This not only fosters a modular architecture but also wards off accidental dependencies.
- **Library Major Versioning:** Within a monorepo, assigning major versions to shared libraries is a prudent move. It empowers projects to lean on consistent versions and transition to the ones being deeply refactored and non-retro-compatible anymore.
- **Automated Dependency Verification:** Implement tools and Bazel add-ons to proactively identify and rectify obsolete or superfluous dependencies, ensuring a streamlined dependency structure.
- **Comprehensive Dependency Documentation:** Uphold a detailed record elucidating the objectives and application of shared libraries. Such insights will equip other teams with the knowledge to effectively utilize shared resources.

## External Dependencies

A monorepo, by definition, houses multiple projects, each potentially having its own set of external dependencies. Bazel offers a systematic approach to handle these dependencies, ensuring consistency, reproducibility, and efficient builds. In this section, we will delve into the intricacies of managing external dependencies using Bazel in a monorepo setup.

### **Key Points:**

- **Direct Dependencies:** Libraries and tools directly used by your project.
- **Transitive Dependencies:** Libraries that your direct dependencies depend on.

Managing dependencies efficiently in a monorepo is essential for several reasons. Firstly, it ensures consistency by making sure all projects within the monorepo utilize the same version of a given dependency. Secondly, it guarantees reproducibility, meaning that builds and tests yield identical results regardless of the machine they're executed on. Lastly, it promotes efficiency by preventing the redundant downloading of the same dependency or the repeated building from its source.

Bazel uses a unique approach to manage external dependencies, focusing on hermeticity and reproducibility.

- **WORKSPACE File:** The heart of dependency management in Bazel is the **WORKSPACE** file. This file resides at the root of your monorepo and defines all external dependencies. Each dependency is represented by a rule, which tells Bazel how to fetch and build it.
- **Repository Rules:** Bazel uses repository rules to fetch and/or build external dependencies. Some common rules include:
  - **http\_archive:** Fetches a tarball and extracts it.
  - **git\_repository:** Clones a Git repository.
  - **local\_repository:** Points to a local directory outside the monorepo.

## Adding an External Dependency

In the `WORKSPACE` file, use a suitable repository rule to define the dependency. For example, to add a dependency on a library hosted on GitHub:

```
git_repository(  
  name = "example_dependency",  
  remote = "https://github.com/example/example_dependency.git",  
  tag = "v1.0.0",  
)
```

Once defined, you can reference the dependency in your `BUILD` files using its name:

```
deps = ["@example_dependency//..."],
```

## Strategies for Version Management

- **Centralized Versioning:** Use a single `WORKSPACE` file at the root of your monorepo to define all external dependencies. This involves defining all dependencies and their versions in a centralized location, which all projects within the monorepo reference. This strategy has the following advantages:
  - **Unified Versions:** All projects use the same version of dependencies.
  - **Simplified Upgrades:** Upgrading a dependency version in one location updates it across all projects.
- **Dependency Locking:** Similar to `yarn.lock` or `package-lock.json` in the JavaScript ecosystem, consider using tools like `bazel-deps` to generate a locked list of transitive dependencies. This ensures reproducibility across builds. Dependency locking involves creating a lock file that pins dependencies to specific versions, ensuring that every build uses the exact same versions of dependencies. Implementation with Bazel:
  - Utilize Bazel's `http_archive` and `git_repository` rules to fetch dependencies.

- Pin dependencies in a `WORKSPACE` file or a centralized dependencies file.

## Automated Dependency Updates

Implementing automated tools and bots, such as Dependabot or Renovate, to automatically submit pull requests when dependency updates are available, ensures that your dependencies are consistently up-to-date.

In addition, there are some other important considerations as follows:

- **Automated Testing:** Ensure that automated updates are tested to validate that they do not introduce breaking changes.
- **Security Patches:** Prioritize automated updates for security vulnerabilities.

## Handling Transitive Dependencies

Transitive dependencies can introduce version conflicts. Utilizing Bazel's dependency resolution and specifying direct dependencies helps in managing and resolving such issues.

The following are some strategies to consider:

- **Explicit Dependency Declaration:** Ensure that all direct dependencies are explicitly declared.
- **Conflict Resolution:** Define strategies for resolving version conflicts, such as preferring the direct dependency version.

## Conflict Resolution

Conflicts can arise when multiple parts of a project depend on different versions of the same external library. Before diving into solutions, it's essential to understand the root of the problem:

- **Multiple Versions:** Different teams or modules might depend on different versions of the same library. This can



lead to inconsistencies and unexpected behaviors.

- **Transitive Dependencies:** A module might depend on Library A, which in turn depends on Library B. If another module depends on a different version of Library B directly, conflicts can arise.
- **Global Context:** In a monorepo, all projects share the same build context. This means that conflicting dependencies can't be isolated as they might be in separate repositories.

Following are some strategic approaches to effectively handle and resolve these dependency clashes:

- **Centralized Version Management:** One approach is to have a centralized team or process to manage the versions of external dependencies. This team would be responsible for updating versions and ensuring compatibility.
- **Use Dependency Mediation:** Instead of using multiple versions, choose a single version that works for all modules. This might involve some compromise and coordination among teams.
- **Shading:** "Shading" is a process where you include a dependency in your project under a different namespace. This allows multiple versions of the same library to coexist without conflict. However, this can increase the binary size and might lead to other unforeseen issues.
- **Avoiding the Conflict:** Sometimes, it might be easier to avoid the conflict altogether. This can be done by:
  - Using a different library that provides similar functionality.
  - Writing custom code to replace the functionality provided by the conflicting library.
- **Custom Bazel Rules:** Advanced users can write custom Bazel rules to handle specific scenarios. For instance, a

rule that allows multiple versions of a library but ensures they are not used in the same binary.

To navigate this intricate landscape, here are some best practices that teams should consider:

- **Regularly Update Dependencies:** Regularly updating dependencies can help in catching and resolving conflicts early.
- **Automate Checks:** Use automated tools to check for dependency updates and potential conflicts.
- **Documentation:** Document the decisions made regarding dependency versions and conflict resolutions. This can be invaluable for new team members and for future references.
- **Communication:** Foster communication among teams. When teams are aware of what others are doing, they can coordinate better and avoid potential conflicts.

Managing dependencies in a monorepo can be challenging, but with the right tools and practices, it becomes a strength rather than a liability. Bazel, with its fine-grained dependency controls and efficient build system, is an invaluable tool for any team working in a monorepo environment. By following best practices and leveraging Bazel's capabilities, teams can ensure consistent, efficient, and reliable builds across all projects in their monorepo.

## **Bazel MODULES: A Modern Way for Handling External Dependencies**

Bazel **MODULES** are a set of conventions and tools designed to manage external dependencies in a Bazel workspace. They provide a standardized way to declare, fetch, and use external sources, ensuring that all dependencies are consistent and reproducible across different environments.

One of the primary advantages of using Bazel **MODULES** is the guarantee of reproducibility. By explicitly defining dependencies, Bazel ensures that builds remain consistent and reproducible across a variety of machines and environments.

Furthermore, Bazel **MODULES** provide robust version management capabilities. Developers have the flexibility to specify the exact versions of their dependencies. This precision not only ensures consistency throughout the project but also aids in avoiding the notorious “*dependency hell*” scenario.

On the security front, Bazel takes extra precautions. It fetches dependencies through secure channels and rigorously verifies their integrity. This meticulous approach significantly diminishes the risk of inadvertently introducing malicious code into the project.

Lastly, efficiency is a hallmark of Bazel **MODULES**. Bazel is designed to cache downloaded dependencies. This feature results in accelerated build times since there’s no need to repeatedly fetch the same dependencies.

## **Declaring Dependencies with MODULES**

To declare an external dependency in a Bazel workspace, you can use the module rule in a **MODULE.bazel** file. Here’s a basic example:

```
module(  
  name = "my_project",  
  version = "1.0",  
  deps = [  
    {  
      "name": "example_dependency",  
      "version": "2.4.1",  
      "repo": {  
        "type": "git",  
        "url": "https://github.com/example/example_dependency.git",  
        "strip_prefix": "example_dependency-2.4.1",
```

```
    },  
  },  
],  
)
```

In this example, `my_project` declares a dependency on `example_dependency` version 2.4.1.

Bazel is a versatile build tool that can integrate with various ecosystems, including the Java ecosystem. One of the challenges in the Java world is managing dependencies, which are often fetched from Maven repositories. The Bazel extension for Maven, provided by `rules_jvm_external`, allows Bazel to fetch and manage Java dependencies from Maven repositories seamlessly.

Let's break down this `MODULES.bazel` file:

```
bazel_dep(name = "rules_jvm_external", version = "5.3")  
# To update maven dependencies, update the lines below and then  
run:  
# bazel run @unpinned_maven//:pin  
maven = use_extension("@rules_jvm_external//:extensions.bzl",  
"maven")  
maven.install(  
  artifacts = [  
    "junit:junit:4.13.2",  
    "org.openjdk.jmh:jmh-core:1.23",  
    "org.openjdk.jmh:jmh-generator-annprocess:1.23",  
  ],  
  lock_file = "://maven_install.json",  
  repositories = [  
    "https://maven.google.com",  
    "https://repo1.maven.org/maven2",  
  ],  
)  
use_repo(maven, "maven", "unpinned_maven")
```

## **Bazel Dependency Declaration**

```
bazel_dep(name = "rules_jvm_external", version = "5.3")
```

This line declares a dependency on the ``rules_jvm_external`` Bazel extension, which provides rules and tools for fetching and managing Java dependencies from Maven repositories. The version ``5.3`` is specified, ensuring a consistent version is used across builds.

## **Maven Extension Initialization**

```
maven = use_extension("@rules_jvm_external//:extensions.bzl",  
"maven")
```

Here, the Maven extension from ``rules_jvm_external`` is initialized and assigned to the ``maven`` variable.

```
# To update maven dependencies, update the lines below and then  
run: bazel run @unpinned_maven//:pin
```

The preceding comment provides guidance on how to update Maven dependencies: by modifying the artifact list and then running the pinning command.

## **Maven Artifacts Installation**

```
maven.install(  
  artifacts = [  
    "junit:junit:4.13.2",  
    "org.openjdk.jmh:jmh-core:1.23",  
    "org.openjdk.jmh:jmh-generator-annprocess:1.23",  
  ],  
  lock_file = "//:maven_install.json",  
  repositories = [  
    "https://maven.google.com",  
    "https://repo1.maven.org/maven2",  
  ],  
)
```

This block instructs Bazel to fetch and install the specified Maven artifacts (`junit`, `jmh-core`, and `jmh-generator-annprocess`).

The ``lock_file`` attribute points to ``maven_install.json``, which is a generated file that locks the versions of all fetched Maven artifacts, ensuring reproducibility across builds.

The ``repositories`` attribute lists the Maven repositories from which the artifacts should be fetched. In this case, artifacts will be fetched from Google's Maven repository and Maven Central.

## Repository Usage

```
use_repo(maven, "maven", "unpinned_maven")
```

This line instructs Bazel to use the fetched Maven artifacts in the Bazel workspace. The ``unpinned_maven`` repository is a special repository that contains the fetched Maven artifacts without version pinning. By running the pinning command mentioned in the comment (``bazel run @unpinned_maven//:pin``), the versions of the artifacts are locked, and the ``maven_install.json`` lock file is generated.

The ``MODULES.bazel`` file provided showcases how Bazel integrates with the Maven ecosystem using the ``rules_jvm_external`` extension. By declaring Maven artifacts in the ``MODULES.bazel`` file, Bazel can fetch, manage, and use these artifacts in the build process, ensuring consistency, reproducibility, and seamless integration with the Java ecosystem.

## [Configuring an Air-Gapped Bazel Build](#)

In an era where security and reliability are paramount, air-gapped builds have emerged as a crucial concept in continuous integration and deployment pipelines. An air-gapped build environment is isolated from the internet, ensuring that the build process is secure, reproducible, and free from external influences, such as sudden changes in dependencies or potential security vulnerabilities from third-party sources.

Bazel has native support for creating reproducible builds, and it can be configured to operate in an air-gapped environment. This involves pre-fetching all dependencies and storing them in a local repository, ensuring that the build process is entirely self-contained and not reliant on external networks.

Consider a scenario where you are building a Java application with Bazel in an air-gapped environment. First, you would need to fetch all necessary dependencies and store them in a local directory. You can use a tool like `bazel-deps` to generate a comprehensive list of dependencies, and then manually download them to a designated directory.

Once all dependencies are locally available, you can configure Bazel to use these local resources during the build process. In your `WORKSPACE` file, you would specify the local paths to the dependencies, ensuring that Bazel does not attempt to fetch anything from the internet. For instance, you might configure a Maven repository as follows:

```
maven_repository(  
  name = "local_maven",  
  urls = ["file:///path_to_your_local_directory"],  
)
```

In your `BUILD` files, you would then reference the dependencies as usual, and Bazel would resolve them using the local paths specified in the `WORKSPACE` file. For example:

```
java_library(  
  name = "my_app",  
  srcs = glob(["src/main/java/com/example/**/*.java"]),  
  deps = [  
    "@local_maven//com/example:dependency1",  
    "@local_maven//com/example:dependency2",  
  ],  
)
```

Also, air gapping involves specifying the `HTTP_ARCHIVE` rule to fetch required files locally instead of from external sources. These dependencies need to be pre-downloaded and made available within the local filesystem. You might want to structure the directory to mimic the URL paths of the original external sources for clarity.

In your `WORKSPACE` file, you will use the `http_archive` rule, but instead of providing a URL to fetch the dependency, you will

point it to the local file using the `URLs` attribute. The `URLs` attribute will contain a `file://` URL that points to the location of the downloaded dependency in your filesystem.

Let's consider an example where you have a dependency archived as `example_dependency.tar.gz` and stored in a directory named `local_deps`. In your `WORKSPACE` file, you would configure the `http_archive` rule as follows:

```
load("@bazel_tools//tools/build_defs/repo:http.bzl",
     "http_archive")
http_archive(
    name = "example_dependency",
    urls =
        ["file:///path/to/local_deps/example_dependency.tar.gz"],
    build_file = "@//path/to:BUILD.example_dependency",
)
```

In this configuration, the `URLs` attribute is pointing to the local filesystem path where `example_dependency.tar.gz` is stored. The `file://` prefix is used to specify that it's a local file path. The `build_file` attribute is used to specify the `BUILD` file for the dependency.

Another strategy involves the use of the `--distdir` argument, which allows Bazel to fetch dependencies from a predefined local directory instead of downloading them from the internet.

In a typical setup, you would first prepare a directory that contains all the necessary dependencies required for the build. This directory, often referred to as a distribution directory, should be populated with the appropriate files and artifacts that your project needs. Ensure that the directory is accessible from the air-gapped environment where the build process will occur.

When initiating a build using Bazel, you can specify the `--distdir` argument followed by the path to the distribution directory. Bazel will then utilize the specified directory as a source to retrieve the necessary dependencies for the build process.



For instance, consider you have a project that depends on several external libraries, and you have prepared a distribution directory at `/path/to/distdir`. When executing a build, you would run a command similar to the following:

```
bazel build //my:target --distdir=/path/to/distdir
```

By doing this, Bazel is instructed to fetch the required dependencies from the local directory at `/path/to/distdir`, ensuring that the build process can proceed seamlessly even in the absence of internet connectivity. This approach is particularly beneficial in secure or isolated environments, allowing for consistent and reliable builds while ensuring that all dependencies are correctly managed and accessible locally.

By configuring Bazel to use local dependencies, you ensure that the build process is isolated from external factors, enhancing its security and reliability. This approach aligns with best practices for creating reproducible builds, ensuring that the application can be consistently built and deployed across various environments without unexpected variations or issues.

## **Enforcing Versioning and Compatibility in a Monorepo**

In a Monorepo, where multiple projects coexist in a single codebase, enforcing internal and external versioning and compatibility is crucial to maintain a consistent and reliable build. With Bazel, you can define rules and constraints that help in managing dependencies, ensuring that the versions of the libraries and tools used across projects are compatible.

Imagine a situation where a Monorepo hosts several projects, each relying on a shared external library. The projects are synchronized to operate on a unified version of this library. When there's a need to upgrade the version, the maintainer initiates the process by creating a new git branch. Within this branch, modifications are made to the version definition in either the `MODULES` or `WORKSPACE` files as necessary.

Following this adjustment, a comprehensive testing phase ensues, involving all projects to ensure they function correctly with the updated library version. Any discovered incompatibilities or issues are meticulously addressed and resolved during this phase. After confirming the seamless operation of all projects with the upgraded library, the changes are merged into the main branch.

This structured approach, facilitated by Bazel, ensures a coordinated upgrade of all projects within the monorepo simultaneously. It effectively mitigates potential conflicts and disruptive alterations, maintaining the integrity and reliability of each component within the monorepo. Thus, Bazel plays a pivotal role in streamlining version upgrades, fostering a stable and consistent development environment across all projects.

## **Querying Dependencies and Getting Graphs**

In a monorepo, managing and understanding dependencies is crucial for efficient development and build optimization. Bazel shines brightly in this aspect, offering robust capabilities to query dependencies and generate insightful graphs to visualize the architecture of your projects. This section will delve into the art of querying dependencies and extracting graphical representations of dependency graphs in a monorepo managed by Bazel.

Bazel's query command is a potent tool that allows developers to explore the dependency graph of their projects. It provides a way to ask various questions about the build, such as finding all dependencies of a target, all targets that depend on a specific target, and even more complex queries to analyze the build structure. The query language used by Bazel is expressive, allowing for a wide range of queries to suit different needs.

Consider a scenario where you have a monorepo with multiple projects, and you want to understand the dependencies of a specific target. You could use the Bazel query command as follows:

```
bazel query 'deps(//project/path:target_name)'
```

This command will list all the dependencies of the specified target, providing a clear view of what components the target relies on. It's a simple yet powerful way to explore the immediate and transitive dependencies of a target.

For a more visual representation, Bazel allows you to generate graphs from the query results. Graphs are instrumental in providing a visual overview of the dependencies, making it easier to analyze and understand the build structure. You can use tools like graphviz to visualize these graphs effectively.

To generate a graph of the dependencies, you could execute a command as follows:

```
bazel query 'deps(//project/path:target_name)' --output graph >
output_graph.dot
```

After executing this command, you can use a tool like graphviz to visualize the .dot file:

```
dot -Tpng output_graph.dot -o output_graph.png
```

This series of commands will produce a PNG image representing the dependency graph of the specified target, allowing for a more intuitive understanding of the dependencies and their relationships.

Diving deeper, Bazel's querying prowess allows for the filtering of results, enabling the extraction of more refined and relevant information. Utilizing the `kind()` function, you can filter the results to display only specific types of targets. For example, if you are solely interested in the Python libraries that a target depends on, you could craft a query like:

```
bazel query 'kind(py_library, deps(//app:my_app))'
```

This query meticulously sifts through the dependencies, presenting only those that are Python libraries, thus allowing

for a more focused and manageable set of results.

In the realm of advanced querying, the `--output` flag unveils further possibilities, allowing for the customization of the output format. By default, the query results are displayed as a list of labels, but with the `--output` flag, you can modify this to receive the output in various formats such as a graph or XML. For example:

```
bazel query 'deps(//app:my_app)' --output=graph
```

would render the dependencies as a graph, providing a visual representation that could be instrumental in understanding the architecture and flow of dependencies.

From the previous chapter's benchmark project, we could run the following command:

```
bazel query 'deps(//app/hellotest:benchmark)' --output graph > output_graph.dot
```

And get something like this:

A terminal window with a dark background and light text. The prompt is `~/building-large-scale-apps-with-monorepo-and-bazel/chapter-5/bazel_modules (9.206s)`. The command entered is `bazel query 'deps(//app/hellotest:benchmark)' --output graph > output_graph.dot`. The output shows the Bazel server starting and connecting, followed by several "checking cached actions" messages. It then reports "Loading: 0 packages loaded", "Loading: 35 packages loaded", "currently loading: @bazel\_tools//third\_party/zlib", and finally "Loading: 37 packages loaded".

```
~/building-large-scale-apps-with-monorepo-and-bazel/chapter-5/bazel_modules (9.206s)
bazel query 'deps(//app/hellotest:benchmark)' --output graph > output_graph.dot
Starting local Bazel server and connecting to it...
checking cached actions
checking cached actions
checking cached actions
checking cached actions
checking cached actions
checking cached actions
Loading: 0 packages loaded
Loading: 35 packages loaded
  currently loading: @bazel_tools//third_party/zlib
Loading: 37 packages loaded
```

**Figure 5.3:** Query execution for `hellotest:benchmark` project and target

Then, executing the following command, we transform the graph into a png image:

```
dot -Tpng output_graph.dot -o output_graph.png
```

And the image shows an impossible-to-see number of elements:



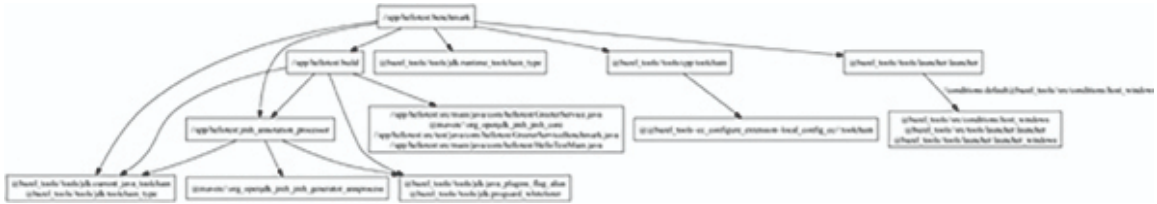
**Figure 5.4:** Generated image for the queried dependency graph

Here are a few ways to refine your query to reduce the number of elements returned:

### 1. Limiting the Depth of Dependencies

Limit the depth of the dependencies retrieved by specifying a number with `--depth`. For example, to get dependencies up to 2 levels deep:

```
bazel query 'deps(//app/hellotest:benchmark, 2)' --output graph > output_graph.dot
```

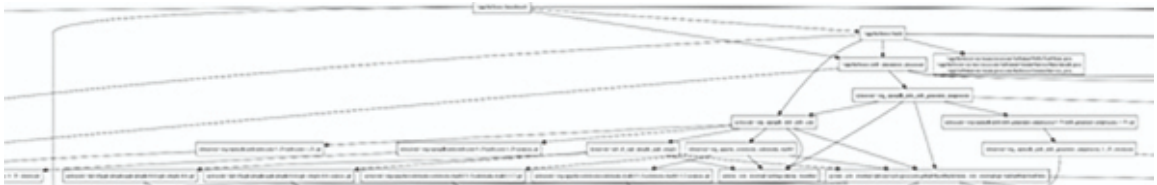


**Figure 5.5:** Generated image when limiting the depth of dependencies

### 2. Filtering Specific Dependencies

You can filter out specific dependencies, such as tests, by using the `except` keyword:

```
bazel query 'deps(//app/hellotest:benchmark) except tests(//app/hellotest:benchmark)' --output graph > output_graph.dot
```

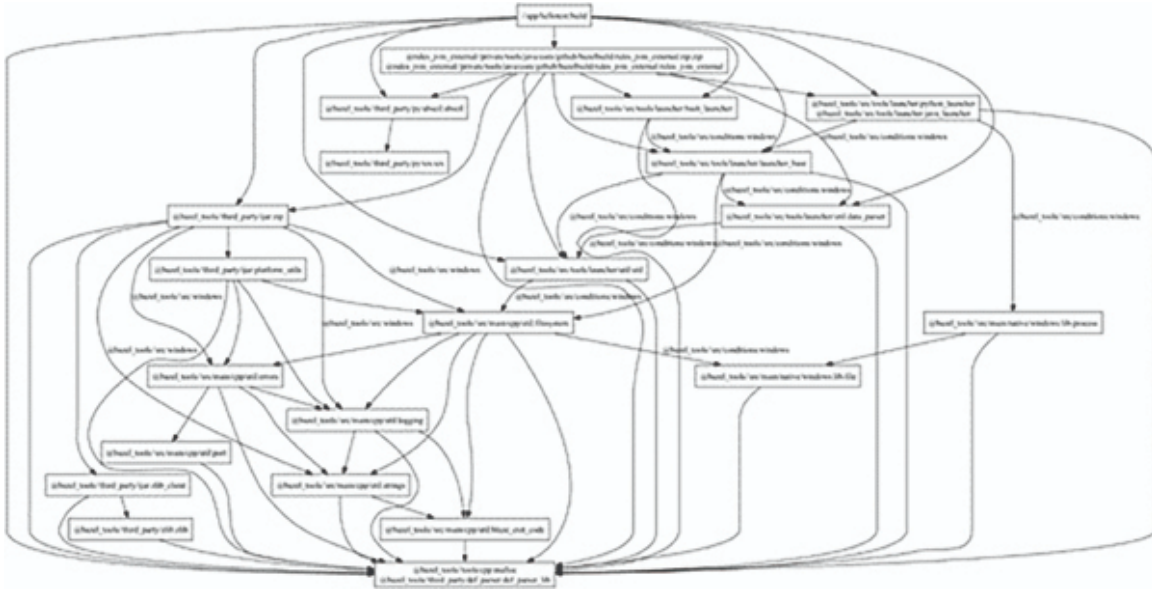


**Figure 5.6:** Generated image when filtering specific dependencies

### 3. Including Specific Types of Dependencies

Include only specific types of dependencies, such as libraries:

```
bazel query 'kind(library, deps(//app/hellotest:benchmark))'
--output graph > output_graph.dot
```



**Figure 5.7:** Generated image when including specific types of dependencies

#### 4. Excluding Specific Paths or Targets

Exclude dependencies from specific paths or targets:

```
bazel query 'deps(//app/hellotest:benchmark, 2) except
//app/hellotest:build' --output graph > output_graph.dot
```

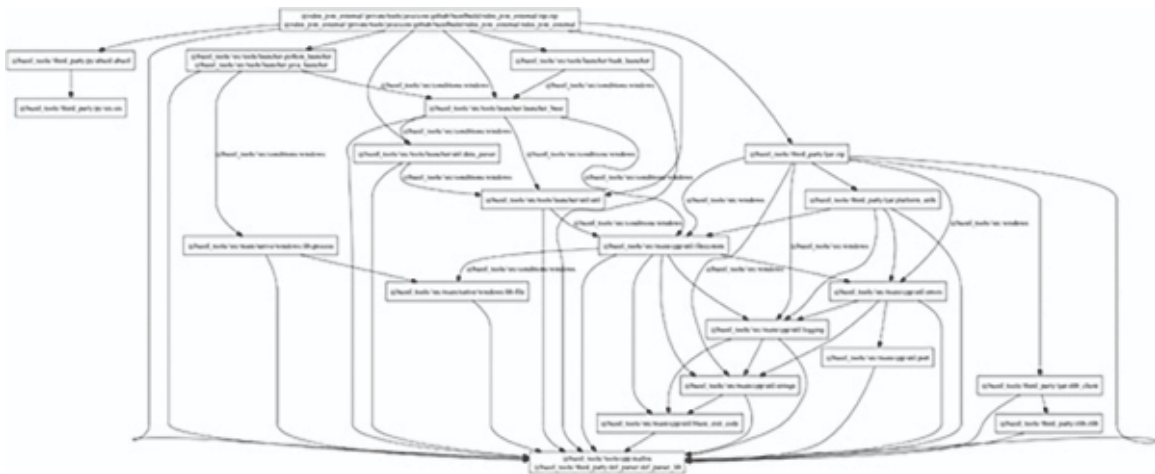


**Figure 5.8:** Generated image when excluding specific paths or targets

#### 5. Combining Multiple Filters

Combine multiple filters to fine-tune the output:

```
bazel query 'kind(library, deps(//app/hellotest:benchmark))
except //app/hellotest:build --output graph >
output_graph.dot
```



**Figure 5.9:** Generated image when combining multiple filters

Adjusting your query by applying filters, limiting depth, or focusing on specific dependency types will help reduce the number of elements returned, making the output more manageable and focused on your needs.

In conclusion, querying dependencies and generating graphs in a monorepo with Bazel is a powerful approach to manage and optimize builds. Through expressive queries and visual representations, developers can gain deep insights into the build structure, enabling informed decisions for build optimization and refactoring.

## [Integrating Bazel Within an IDE](#)

Integrating Bazel within an IDE can supercharge the development workflow, allowing for a seamless and productive development experience.

The official Bazel plugin is available for IntelliJ, Android Studio, Visual Studio Code, and CLion and can be accessed at <https://ij.bazel.build>. This plugin is open-source and mirrors the version utilized internally at Google.

### **Features:**

- **Language-Specific Interoperability:** The plugin is compatible with various language-specific plugins, including those for Java, Scala, and Python.

- **Semantic Awareness:** It allows the importation of BUILD files into the IDE, recognizing the semantics of Bazel targets.
- **Starlark Recognition:** The plugin enables the IDE to recognize Starlark, the language used in Bazel's BUILD and .bzl files.
- **Direct Execution:** It facilitates the building, testing, and execution of binaries directly within the IDE.
- **Configurable:** Users can create configurations tailored for debugging and running binaries.

### Installation:

- **Automatic Installation:** For automatic installation, navigate to the plugin browser in your IDE and search for the Bazel plugin.
- **Manual Installation:** For installing older versions manually, download the necessary zip files from JetBrains' Plugin Repository. Subsequently, these zip files can be installed directly through the IDE's plugin browser.

For more information, please visit the Bazel IDE installation page at <https://bazel.build/install/ide>.

## Conclusion

This chapter provided a comprehensive exploration into the realm of dependency management and versioning within Bazel. We delved deeply into managing both internal and external dependencies, emphasizing the significance of each in a robust build environment. The chapter elucidated the mechanisms of conflict resolution, ensuring that dependencies coexist harmoniously within a project. A pivotal aspect covered was Bazel **MODULES**, a contemporary approach that revolutionizes the management of external dependencies. Through **MODULES**, we learned the intricacies of declaring and managing dependencies, configuring air-gapped Bazel builds,



and enforcing stringent versioning and compatibility within a monorepo.

In the latter part of the chapter, we transitioned into practical insights, demonstrating how to query dependencies and generate insightful graphs to visualize dependency relationships. A crucial section was dedicated to integrating Bazel within an Integrated Development Environment (IDE), which is instrumental in enhancing developer productivity and streamlining the build and test processes. This chapter, rich with practical examples and strategic insights, aims to be a valuable resource for mastering dependency management in Bazel, ensuring that your projects are not only robust and reliable but also optimized for efficiency and productivity.

In the next chapter, we will discuss how to set up, organize, build, and run projects, as well as the best practices for using Bazel on the following development platforms:

- Android/Kotlin
- Python
- NodeJS/TypeScript
- Golang
- iOS

## **Further Readings**

- Build programs with Bazel: <https://bazel.build/run/build#distribution-directory>
- Query guide | Bazel: <https://bazel.build/query/guide>
- Bazel: Managing Dependencies: <https://docs.bazel.build/versions/main/be/general.html#dependencies>
- Dependency Management with Bazel: <https://medium.com/@Jakeherringbone/dependency-management-with-bazel-3e3b0ce6f4ed>

- Bazel Modules: Dependencies for Bazel projects:  
<https://blog.bazel.build/2021/08/10/bazel-deps.html>
- Building software in a restricted environment with Bazel:  
<https://docs.bazel.build/versions/main/restricted-environment.html>
- Managing Dependencies in your Bazel Build:  
<https://medium.com/swlh/managing-dependencies-in-your-bazel-build-9c0b3e552f6e>
- Visualizing Bazel Build Graphs:  
<https://docs.bazel.build/versions/main/skylark/depgraph.html>
- Integrating Bazel with IDEs:  
<https://docs.bazel.build/versions/main/ide.html>
- Dependency Management for Bazel and the Monorepo:  
<https://github.com/jflex-de/bazel-deps>

## **CHAPTER 6**

# **Hello-World Using Other Languages and Platforms**

## **Introduction**

The journey of a software developer often commences with the iconic initiation: crafting a “Hello World” program. This humble beginning is far more than a trivial ritual; it is an introduction to the fundamental principles of various coding languages and platforms. “Hello World” serves as a mirror, reflecting the syntax, structure, and style inherent to each environment. In this chapter, we are set to navigate through the nuances of this classic exercise as it translates across different technological landscapes.

Our exploration will span several robust and widely-used languages and platforms. We will start with Android/Kotlin, which is mainstream in Android app development, before moving on to the versatile and accessible Python. Following that, we will dive into NodeJS, a cornerstone of modern server-side development, and then to Go, with its streamlined approach to systems programming. Finally, we will enter the realm of iOS, to see how “Hello World” manifests in Apple’s sophisticated ecosystem. This chapter is not just about displaying a greeting on the screen; it is about grasping the foundations and idiosyncrasies of each platform, offering both beginners and experts a panoramic view of programming’s diverse ecosystem through the lens of the simplest possible program.

## **Structure**

In this chapter, we will explore how to set up, organize, build, run, and apply best practices when using Bazel across the following development platforms:

- Android/Kotlin
- Python
- NodeJS/TypeScript
- Golang
- iOS

## **Android/Kotlin**

In this section, we will dive into the world of Android app development using Bazel as our build tool and Kotlin as our programming language. This assumes that you are already familiar with both Bazel and Android/Kotlin development and both are already installed. We will explore how to set up a Bazel-based Android project, define build targets, and manage dependencies effectively.

In this section, we will delve into a specific set of code examples. These are conveniently accessible through our GitHub repository at

<https://github.com/OrangeAVA/Building->

[Large-Scale-Apps-with-Monorepo-and-Bazel](https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel), located within the `/chapter-6/bazel_android_kotlin` directory. This repository serves as a practical reference throughout our discussion.

## **Setting up Your Android/Kotlin Bazel Project**

Before we begin, ensure that you have Bazel and Android development tools installed on your system. If you have not already, follow the official installation instructions for both.

Let us check our development environment is prepared for Bazel, Android and Kotlin development by running the following commands:

```
echo $ANDROID_HOME  
/Users/jrai/Library/Android/sdk
```

This shows the path where the android SDK is installed. Then, let us check the emulator availability by running the following commands:

```
emulator -version  
INFO    | Android emulator version 32.1.14.0 (build_id  
10330179)  
INFO    | Storing crashdata in: /tmp/android-jrai/emu-  
crash.db,  
INFO    | Duplicate loglines will be removed, if you wish to  
see  
Android emulator version 32.1.14.0 (build_id 10330179)  
(CL:N/A)  
...
```

And finally, let us check we also have the Android Debug Bridge:

```
adb --version  
Android Debug Bridge version 1.0.41  
Version 34.0.3-10161052  
Installed as /Users/jrai/Library/Android/sdk/platform-  
tools/adb  
Running on Darwin 22.6.0 (arm64)
```

If you have retrieved the previous logs, we are all set to proceed with our journey.

## [Organizing Your Android App](#)

By structuring your Android project into several Bazel projects, you enhance modularity, facilitate code reuse, and boost build efficiency. These projects ought to be arranged

in a manner that separates applications, shared functional code, and shared non-functional code, such as architectural elements. In the upcoming chapters, we will explore this subject in depth.

For this very simple project, we are also going to simplify the structure like this:

```
.
├── MODULE.bazel
├── WORKSPACE
├── app
│   ├── BUILD.bazel
│   └── src
│       └── main
│           ├── AndroidManifest.xml
│           ├── java
│           ├── manifest
│           └── res
└── common
    ├── dashboard
    │   ├── BUILD.bazel
    │   └── src
    │       └── main
    ├── home
    │   ├── BUILD.bazel
    │   └── src
    │       └── main
    └── notifications
        ├── BUILD.bazel
        └── src
            └── main
```

We will explore key files and directories, including:

- **WORKSPACE:** The unique rule in this file is to configure the Android SDK as explained later. All dependencies are defined in **MODULE.bazel**.

- **MODULE.bazel**: This file defines the required external dependencies as was explained in previous chapters.
- **app** directory: contains build rules, internal and external dependencies, and source code to build the Android application.
- **common** Directory: contains shared or common modules/components like dashboard, home, and notifications. Each module may represent a feature or a shared library.

In your **WORKSPACE** file, include the necessary Android and Kotlin rules:

```
android_sdk_repository(  
    name = "androidsdk",  
)
```

The **android\_sdk\_repository** rule is invoked with a specified name **"androidsdk"**. This name acts as an identifier for referencing the Android SDK within **BUILD** files throughout the workspace. No explicit path or version is provided, implying that Bazel might use predefined environment variables or default configurations to locate and utilize the appropriate Android SDK version for building Android targets. In this configuration, Bazel is informed about the Android SDK's existence and location, enabling the build and compilation of Android application modules and libraries within the workspace using the specified or automatically detected Android SDK.

The **MODULE.bazel** file defines the external dependencies by using the modern Bazel dependency management approach.

```
module(name = "android-example")  
  
bazel_dep(name = "rules_android", version = "0.1.1")  
bazel_dep(name = "rules_kotlin", version = "1.9.0")  
bazel_dep(name = "bazel_skylib", version = "1.2.1")
```

```
bazel_dep(name = "platforms", version = "0.0.7")
bazel_dep(name = "rules_jvm_external", version = "5.3")
maven = use_extension("@rules_jvm_external//:extensions.bzl",
"maven")
maven.install(
  artifacts = [
    "androidx.appcompat:appcompat:1.0.2",
    "androidx.core:core-ktx:1.3.0",
    "com.google.android.material:material:1.0.0",
    "androidx.constraintlayout:constraintlayout:1.1.3",
    "androidx.lifecycle:lifecycle-extensions:2.0.0",
    "androidx.navigation:navigation-fragment:2.0.0",
    "androidx.navigation:navigation-ui:2.0.0",
    "androidx.navigation:navigation-fragment-ktx:2.0.0",
    "androidx.navigation:navigation-ui-ktx:2.0.0",
    "androidx.navigation:navigation-runtime-ktx:2.0.0"
  ],
  repositories = [
    "https://maven.google.com",
    "https://jcenter.bintray.com/",
    "https://repo1.maven.org/maven2",
  ],
)
use_repo(maven, "maven")
```

Here, we are including:

- **rules\_android**: Provides Bazel rules for building and managing Android applications and libraries, making Android development more efficient and reproducible.
- **rules\_kotlin**: Offers Bazel rules for building Kotlin projects, enabling seamless integration of Kotlin code into the Bazel build system.
- **bazel\_skylib**: A library that provides fundamental building blocks and utilities for writing Bazel build rules



and extensions, enhancing the flexibility and extensibility of Bazel.

- **platforms**: Allows the specification of platform-specific configurations, enabling consistent and controlled dependency management for various platforms within a Bazel project.
- **rules\_jvm\_external**: Offers Bazel rules to fetch and manage external dependencies for Java and JVM-based projects, simplifying the integration of third-party libraries into your Bazel builds.

Then, this `MODULE.bazel` file uses the maven support to include the external dependencies required for the Android app.

The `BUILD` file for the `app` project should contain the Android rules and dependencies:

```
load("@rules_kotlin//kotlin:android.bzl",
      "kt_android_library")
android_binary(
    name = "app",
    manifest = "src/main/AndroidManifest.xml",
    manifest_values = {
        "versionCode": "1",
        "versionName": "1.0",
        "minSdkVersion": "21",
        "targetSdkVersion": "30",
    },
    deps = [
        ":res",
        ":build_kt",
    ],
)
android_library(
    name = "res",
    manifest = "src/main/manifest/AndroidManifest.xml",
```

```

custom_package = "com.example.myapplication",
resource_files = glob(["src/main/res/**"]),
visibility = ["//visibility:public"],
deps = [
    "@maven//:com_google_android_material_material",
    "@maven//:androidx_constraintlayout_constraintlayout",
    "@maven//:androidx_navigation_navigation_ui_ktx",
    "@maven//:androidx_navigation_navigation_fragment_ktx",
]
)
kt_android_library(
    name = "build_kt",
    srcs = glob(["src/main/java/**/*.*kt"]),
    deps = [
        "//app:res",
        "//common/dashboard",
        "//common/home",
        "//common/notifications",
        "@maven//:androidx_appcompat_appcompat",
        "@maven//:androidx_navigation_navigation_fragment_ktx",
        "@maven//:androidx_navigation_navigation_ui_ktx",
    ],
    visibility = ["//app:__subpackages__"],
)

```

- **android\_binary**: Represents the Android application itself, relying on Kotlin sources and resources defined in other targets.
- **android\_library**: Holds the Android resources and some dependencies, acting as a shared resource library.
- **kt\_android\_library**: Represents the Kotlin source files necessary for the application, along with their dependencies.

The **BUILD** file for the `common/dashboard` project should contain the Android rules and dependencies:

```

load("@rules_kotlin//kotlin:android.bzl",
     "kt_android_library")
kt_android_library(
    name = "dashboard",
    srcs = glob(["src/main/java/**/*.*kt"]),
    deps = [
        "//app:res",
        "@maven//:androidx_lifecycle_lifecycle_extensions"
    ],
    visibility = ["//app:__subpackages__"],
)

```

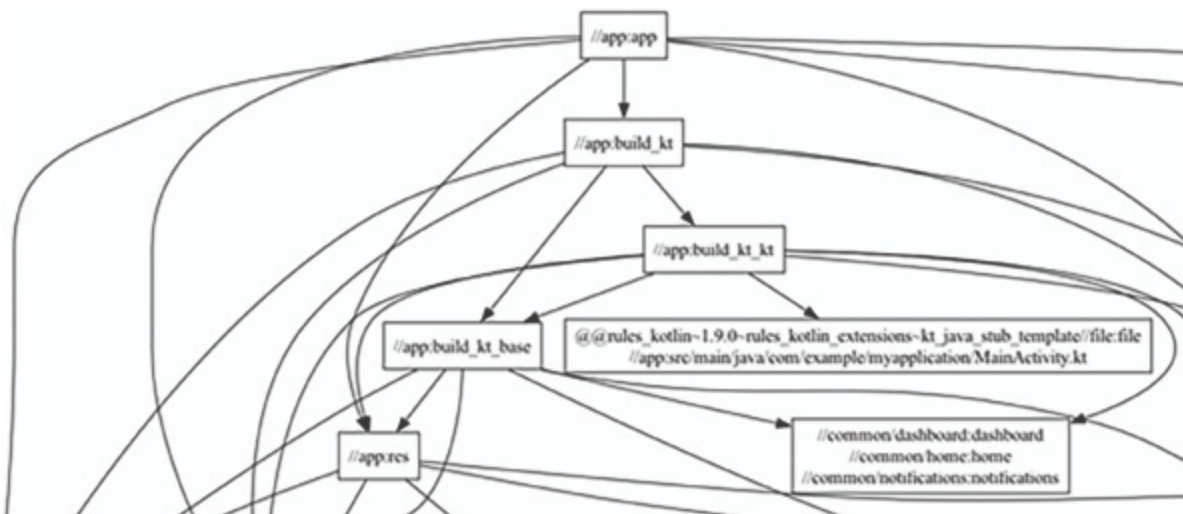
In summary, this `BUILD.bazel` file defines a Kotlin Android library named `dashboard`, includes Kotlin source files from a specified path, declares dependencies, and sets visibility for other packages within the workspace to use this library.

The same approach is used for `home` and `notifications` libraries. The three of them are then used by the `app` project. Let us get a dependency diagram by running the following commands:

```

bazel query 'deps("//app", 3) except @bazel_tools//...:* except
@rules_kotlin//...:* except @maven//...:*' --output graph >
output_graph.dot && dot -Tpng output_graph.dot -o
output_graph.png && open output_graph.png

```



**Figure 6.1:** *Generated dependency diagram showing app project and its libraries. The original diagram is too large, so this was intentionally cropped*

In the application we are going to host a simple **MainActivity** at `app/src/main/java/com/example/myapplication/MainActivity.kt` like this:

```
package com.example.myapplication
import ...
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val navView: BottomNavigationView =
            findViewById(R.id.nav_view)
        val navController =
            findNavController(R.id.nav_host_fragment)
        // Passing each menu ID as a set of Ids because each
        // menu should be considered as top level destinations.
        val appBarConfiguration = AppBarConfiguration(setOf(
            R.id.navigation_home, R.id.navigation_dashboard,
            R.id.navigation_notifications)
        )
        setupActionBarWithNavController(navController,
            appBarConfiguration)
        navView.setupWithNavController(navController)
    }
}
```

This application project's **BUILD.bazel** file also includes the ones for the dashboard, home, and notifications so its references are resolved using these internal libraries.

The home project located at `//common/home` is going to host the **Fragment** and the **ViewModel** classes. **HomeFragment** creates the view and links the **ViewModel** class.

```
package com.example.myapplication.ui.home
```

```

import ...
class HomeFragment : Fragment() {
    private lateinit var homeViewModel: HomeViewModel

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        homeViewModel =
            ViewModelProviders.of(this).get(HomeViewModel::class.java
            )
        val root = inflater.inflate(R.layout.fragment_home,
            container, false)
        val textView: TextView = root.findViewById(R.id.text_home)
        homeViewModel.text.observe(viewLifecycleOwner, Observer {
            textView.text = it
        })
        return root
    }
}

```

The **ViewModel** class handles UI-related data management tasks, such as configuring the content to be displayed in labels.

```

package com.example.myapplication.ui.home
import ...
class HomeViewModel : ViewModel() {
    private val _text = MutableLiveData<String>().apply {
        value = "This is home Fragment"
    }
    val text: LiveData<String> = _text
}

```

Similarly, the dashboard and notifications components follow the same approach when implementing their respective classes.

## **Building and Running Your Android App**

Let us initiate the Android emulator. First, identify the available virtual devices by running the command:

```
emulator -list-avds
```

For example, you might see:  
Pixel\_3a\_API\_34\_extension\_level\_7\_arm64-v8a.

To start the emulator with this device, use the command:

```
emulator -avd Pixel_3a_API_34_extension_level_7_arm64-v8a &
```

The appended ampersand (&) allows the emulator to operate in the background, freeing up the console for subsequent commands. Next, build the app, including all necessary internal dependencies, using:

```
bazel build //app
```

The initial build process might be a bit slow as everything gets built.

```

ip.class:
Methods with invalid locals information:
  void androidx.viewpager.widget.PagerTitleStrip.updateTextPositions(int, float, boolean)
  Info: Information in locals-table is invalid with respect to the stack map table. Local refers to
Info: Some warnings are typically a sign of using an outdated Java toolchain. To fix, recompi
INFO: From Desugaring common/dashboard/dashboard_kt.jar for Android:
Info: Unexpected error while reading com.example.myapplication.ui.dashboard.DashboardFragment
Info: Unexpected error while reading com.example.myapplication.ui.dashboard.DashboardFragment
Info: Unexpected error while reading com.example.myapplication.ui.dashboard.DashboardViewMode
INFO: From Dexing external/rules_jvm_external~5.3~maven~maven/_dx/androidx_navigation_navigat
Info: Stripped invalid locals information from 1 method.
Info in bazel-out/android-armeabi-v7a-fastbuild/bin/external/rules_jvm_external~5.3~maven~mav
tinationBuilder.class:
Methods with invalid locals information:
  androidx.navigation.NavDestination androidx.navigation.NavDestinationBuilder.build()
  Type information in locals-table is inconsistent. Cannot constrain type: INT for value: v23
Info: Some warnings are typically a sign of using an outdated Java toolchain. To fix, recompi
Target //app:app up-to-date:
  bazel-bin/app/app_deploy.jar
  bazel-bin/app/app_unsigned.apk
  bazel-bin/app/app.apk
INFO: Elapsed time: 136.621s, Critical Path: 27.81s
INFO: 713 processes: 64 internal, 578 darwin-sandbox, 71 worker.
INFO: Build completed successfully, 713 total actions

```

**Figure 6.2:** Console output when executing “*bazel build //app*”

After this book section, try by your own to make modifications to one of the libraries and rebuild. You will notice Bazel’s efficiency in updating binaries — this becomes especially noticeable with larger Android applications.

After the build, the Android app binary will be assembled and ready for installation via Bazel:

```
bazel mobile-install //app
```

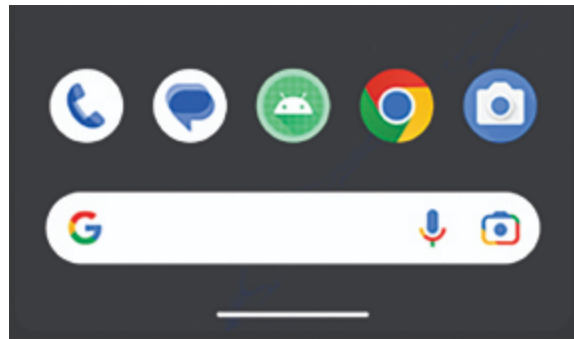
```

bazel mobile-install //app
WARNING: For repository 'bazel_skylib', the root module requires module version bazel_
INFO: Analyzed target //app:app (0 packages loaded, 0 targets configured).
INFO: Found 1 target...
INFO: From Installing //app:app:
I1101 17:52:26.013684 8396824704 incremental_install.py:445] Updating 1 dex...
I1101 17:52:26.084872 8396824704 incremental_install.py:506] Application resources up-
I1101 17:52:26.120445 8396824704 incremental_install.py:604] Updating 0 native libs...
Target //app:app up-to-date:
  bazel-bin/app/app_files/full_deploy_marker
  bazel-bin/app/app_files/deploy_info_incremental.deployinfo.pb
INFO: Elapsed time: 6.564s, Critical Path: 6.20s
INFO: 33 processes: 19 internal, 10 darwin-sandbox, 1 local, 3 worker.
INFO: Build completed successfully, 33 total actions

```

**Figure 6.3:** Console output when executing “*bazel mobile-install //app*”

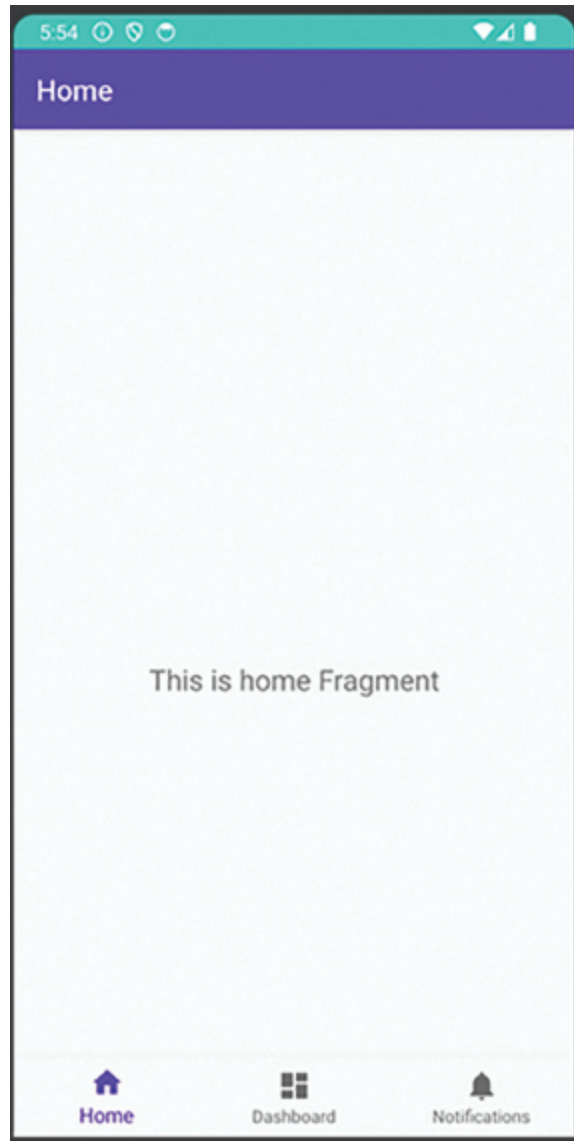
You can then observe the installed app on the emulator's main screen, as illustrated in the subsequent image.



**Figure 6.4:** App icon appears in the middle

Upon launching the app, you will see its home screen.





**Figure 6.5:** Home screen

To remove the application, the following command can be executed:

```
adb uninstall com.example.myapplication
```

In this section, we have explored the process of developing Android apps using Kotlin and Bazel. You have learned how to set up your development environment, create a Bazel workspace, define Android targets, build your app, and run it on an emulator or device. With this knowledge, you are

well-equipped to streamline your Android development workflow with the power of Bazel and Kotlin.

## **Best Practices Using Android/Kotlin in Bazel**

This section focuses on leveraging Bazel's capabilities for Android and Kotlin development, particularly utilizing the `MODULE.bazel` features. Bazel's efficient build system can significantly improve your Android/Kotlin project's build times and reliability when configured correctly.

- **Structuring Your Project**

- Modularization: Break down your Android/Kotlin project into multiple modules. Each module should represent a logical unit of your application.
- `MODULE.bazel` per Module: Place a `MODULE.bazel` file in each module's root. This file should declare the module's dependencies and other configurations specific to that module.

- **Declaring Dependencies**

- Explicit Declaration: Use `MODULE.bazel` to explicitly declare dependencies for each module. This practice improves build performance and readability.
- Version Management: Centralize dependency versions in a single location to ensure consistency across modules.

- **Optimizing Kotlin Builds**

- Kotlin Rules: Use the latest Kotlin rules for Bazel. Ensure they are compatible with your Kotlin version for optimal performance.

- Incremental Compilation: Leverage Bazel's incremental compilation features for Kotlin. This reduces build times significantly by only rebuilding what is necessary.
- **Android Resource Management**
  - Resource Modularization: Divide resources into respective modules. This method makes managing and referencing resources more straightforward.
  - Resource Processing: Use Bazel's resource processing tools to optimize and package resources efficiently.
- **Testing Strategies**
  - Unit Testing: Set up unit tests in each module. Use `MODULE.bazel` to declare test dependencies.
  - Instrumentation Testing: Configure instrumentation tests for Android. Ensure Bazel can run these tests in a consistent environment.
- **Continuous Integration**
  - CI Configuration: Integrate Bazel builds into your CI pipeline. Bazel's reproducibility and speed offer significant advantages in CI environments.
  - Cache Utilization: Utilize Bazel's caching mechanisms to reduce build times in CI.
- **Cross-Module Refactoring**
  - Refactor Safely: When refactoring across modules, use Bazel's query language to understand dependency graphs. This guarantees that changes do not break dependent modules.
- **Performance Monitoring**

- Build Profiling: Regularly profile your builds to identify bottlenecks. Bazel provides tools for detailed build profiling.
- Optimization: Continuously optimize build scripts and configurations based on profiling data.

Adopting these best practices in your Android/Kotlin projects using Bazel, particularly with `MODULE.bazel` features, can significantly enhance your development experience. Bazel's powerful build optimization, along with structured module management, can lead to more efficient, maintainable, and scalable projects. Remember, the key is in understanding and leveraging Bazel's unique features to their full potential.

## Python

Within this section, we will delve into the process of creating a Python application that leverages multiple Bazel projects. Bazel, a potent build tool, offers the capability to oversee intricate projects and their dependencies. By structuring your Python application across several Bazel projects, you can enhance its modularity, maintainability, and build effectiveness.

In this section, we will delve into code examples that can be found in the GitHub repository at <https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel>, specifically within the `/chapter-6/bazel_python` directory. These examples will guide our discussion and exploration throughout this part of the book.

## Setting up Your Python Environment

Before you start working on your Python application using Bazel, it is essential to set up your Python environment correctly. Follow these steps to ensure a smooth development experience.

Make sure you have Python installed on your system. Bazel supports Python 2.7 and Python 3.5 or later. It is recommended to use Python 3 for new projects as Python 2 is no longer supported.

You can check your Python version by running:

```
python --version
```

If Python is not installed, download and install it from the official Python website.

## Organizing Your Python Project

Let us explore how to effectively organize a Python project using Bazel, a powerful tool for automating software builds and testing. The project structure we will be focusing on is as follows:

```
.
├── WORKSPACE.bazel
├── MODULE.bazel
├── BUILD.bazel
├── app
│   ├── BUILD.bazel
│   └── main.py
├── common
│   ├── BUILD.bazel
│   ├── calculator.py
│   └── calculator_test.py
└── requirements.txt
```

In this structure, we found:

- **WORKSPACE.bazel**: For a Python project, this file typically is empty and dependencies rules are often defined in **MODULE.bazel** meanwhile specific libraries are specified in **requirements.txt**.
- **MODULE.bazel**: The **module.bazel** file defines the module named “python-example,” sets up Bazel dependencies

(specifically `rules_python` version “0.26.0”), configures Python toolchain with coverage tool and Python version “3.11”, and establishes Python and pip extensions for handling dependencies specified in `requirements.txt`.

- `/BUILD.bazel`: Contains general build rules or configurations applicable to the entire project but in this case is empty as anyone is required.
- `app/BUILD.bazel`: Defines the build targets for your application, for example, `py_binary` for `main.py`.
- `common/BUILD.bazel`: Contains build targets related to your common library, including `py_library` for `calculator.py` and `py_test` for `calculator_test.py`.
- `app/main.py`: The main Python file for your application.
- `common/calculator.py`: A Python file for common functionalities used across the project.
- `common/calculator_test.py`: Test file for `calculator.py`.
- `requirements.txt`: A standard Python requirements file used to specify third-party dependencies.

## **MODULE.bazel in depth**

The `MODULE.bazel` file in question is configuring a Bazel module for a Python project. Let us break down its components and understand what each part is doing:

### **1. Module Declaration:**

```
module(name = “python-example”)
```

This line declares a Bazel module with the name `python-example`. Bazel modules are a way to organize and encapsulate a set of Bazel build configurations and dependencies.

### **2. Bazel Dependency on `rules_python`:**

```
bazel_dep(name = “rules_python”, version = “0.26.0”)
```

Here, `bazel_dep` specifies a dependency on `rules_python` at version 0.26.0. `rules_python` is a collection of Bazel rules for Python, providing essential tools and libraries for building and testing Python applications with Bazel.

### 3. Python Toolchain Configuration:

```
python =
use_extension("@rules_python//python/extensions:python.bzl", "python")
python.toolchain(
    configure_coverage_tool = True,
    is_default = True,
    python_version = "3.11",
)
use_repo(python, "python_versions")
```

This section uses the `use_extension` function to load Python-related rules from `rules_python`. It then configures the Python toolchain, specifying:

- `configure_coverage_tool = True`: Enables the configuration of a coverage tool for Python code.
- `is_default = True`: Sets this toolchain as the default Python toolchain.
- `python_version = "3.11"`: Specifies that Python version 3.11 should be used.

The `use_repo` function is then used to associate this Python configuration with a repository named `python_versions`, which likely contains different Python versions.

### 4. Pip Dependency Parsing:

```
pip =
use_extension("@rules_python//python/extensions:pip.bzl", "pip")
pip.parse(
    hub_name = "bazel_pip_deps",
```

```
python_version = "3.11",
requirements_lock = "://:requirements.txt",
)
use_repo(pip, "bazel_pip_deps")
use_repo(python, "python_3_11")
```

This part loads the pip extension from `rules_python` to manage Python pip dependencies. The `pip.parse` function is called to configure the pip dependencies with the following parameters:

- `hub_name = "bazel_pip_deps"`: Defines a name for the dependency hub, presumably where the parsed pip dependencies will be stored.
- `python_version = "3.11"`: Ensures compatibility with Python 3.11.
- `requirements_lock = "://:requirements.txt"`: Points to a `requirements.txt` file at the root of the Bazel workspace, which lists the pip dependencies.
- The `use_repo` calls associate the pip configuration with the repositories `bazel_pip_deps` and `python_3_11`, linking them with the pip dependencies and the Python version, respectively.

In summary, this `module.bazel` file sets up a Bazel module for a Python project, configures the Python toolchain for version 3.11, and sets up pip dependency management using a `requirements.txt` file, all within the context of the Bazel build system.

### **`/app/BUILD.bazel` in depth**

This `app/BUILD.bazel` file is configuring a Bazel build rule to create an executable Python program named `main`, which consists of `main.py` and depends on both a local Bazel target (`//common:calculator`) and an external Python package (Flask).

```
load("@bazel_pip_deps//:requirements.bzl", "requirement")
```



```
py_binary(  
    name = "main",  
    srcs = ["main.py"],  
    deps = [("//common:calculator",  
            requirement("Flask"))  
    ]  
)
```

**/common/BUILD.bazel**

This file is used to define a Python library named "calculator" consisting of the calculator.py file and a test suite named "calculator\_test" for testing the functionality in calculator.py, with a dependency linking the test to the library.

```
load("@rules_python//python:defs.bzl", "py_library",  
      "py_test")  
py_library(  
    name = "calculator",  
    srcs = ["calculator.py"],  
    visibility = [("//visibility:public")  
    ]  
)  
py_test(  
    name = "calculator_test",  
    srcs = ["calculator_test.py"],  
    deps = [  
        ":calculator"  
    ],  
)
```

**main.py**

This code sets up a Flask web server that, when accessed on the root path (/), displays the sum of two randomly generated numbers using a Calculator class.

```
from common.calculator import Calculator  
from flask import Flask
```

```
from random import randint
app = Flask(__name__)
calculator = Calculator()
@app.route('/')
def randomNumberCalculator():
    randomInt1 = randint(0, 250)
    randomInt2 = randint(0, 250)
    calculator = Calculator()
    return "{} + {} = {}?".format(randomInt1, randomInt2, \
    calculator.add(randomInt1, randomInt2))
if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

### **calculator.py**

This code defines a class named `calculator` with a single method `add`. This method takes two parameters, `x` and `y`, and returns their sum. Essentially, it implements a basic addition operation.

```
class Calculator:
    def add(self, x, y): return x + y;
```

### **calculator\_test.py**

This Python code is a unit test script using the `unittest` framework. It tests a method from the `Calculator` class, which is assumed to be defined in the `common.calculator` module. Specifically, the script contains a test case class `TestSum` that inherits from `unittest.TestCase`.

In the `TestSum` class, there is a method `test_sum` that:

- Instantiates an object of the `calculator` class.
- Calls the `add` method of the `calculator` object with arguments `1` and `2`.
- Asserts that the result of this `add` method call is equal to `3` using `assertEqual`.

Finally, the script checks if it is the main module being run (`if __name__ == "__main__":`). If it is, it executes the tests by calling `unittest.main()`. This is a typical pattern for running unit tests in Python, allowing the script to be run standalone to execute the tests.

```
import unittest
from common.calculator import Calculator
class TestSum(unittest.TestCase):
    def test_sum(self):
        calculator = Calculator()
        self.assertEqual(calculator.add(1, 2), 3)
if __name__ == "__main__":
    unittest.main()
```

## [Building and Running Your Python App](#)

The commands for building and running this Python applications are as usual:

- `bazel build //app` to build your application.
- `bazel test //common:calculator_test` to run tests.
- `bazel run //app` to run the application.

Once running, you are able to access this app by opening a browser and pointing to `http://127.0.0.1:5000`.

## [Best Practices Using Python in Bazel](#)

The recent introduction of `Modules.bazel` has further enhanced its capabilities, especially for Python projects.

### **Understanding Modules.bazel**

`Modules.bazel` is an extension to Bazel that allows for more modular and manageable build configurations. It simplifies

the management of dependencies and enhances the reusability of code.

### **Key features:**

- **Modularity:** Facilitates dividing the project into smaller, more manageable modules.
- **Dependency Management:** Simplifies dependency declaration and resolution.
- **Versioning:** Supports specifying versions for dependencies, ensuring consistency across builds.

### **Python-Specific Practices in Bazel**

When working with Python in Bazel, consider the following best practices:

- **Organize Python Code into Modules:** Structure your Python code into modules and submodules. This organization aligns well with `Modules.bazel` and helps in maintaining a clean build structure.
- **Use `py_library` and `py_binary` Rules:**
  - **`py_library`:** Define reusable Python libraries. Each library should correspond to a Python module.
  - **`py_binary`:** Use this rule for Python executables. Ensure that the main function is well-defined in the specified source file.
- **Manage Dependencies with `Modules.bazel`:** Declare Python dependencies within `Modules.bazel` files. Use the `module()` function to define module properties and dependencies.
- **Version Control:** Utilize the versioning capabilities of `Modules.bazel` to lock down dependency versions. This practice ensures reproducible builds and avoids the “*it works on my machine*” syndrome.

- **Integration with External Dependencies:** For Python projects that rely on external packages (for example, from PyPI), integrate these using the `pip_import` rule or similar mechanisms. Define these dependencies in `Modules.bazel` for better management.
- **Testing:** Leverage Bazel's testing rules like `py_test` to define and run Python tests. Group tests according to functionality and use `Modules.bazel` to handle test-specific dependencies.
- **Cross-language Interoperability:** If your project involves multiple languages (for example, Python and C++), use Bazel's capabilities to manage inter-language dependencies and build processes. `Modules.bazel` can be particularly effective in handling such complexities.
- **Continuous Integration (CI):** Integrate Bazel builds with your CI pipeline. Utilize `Modules.bazel` to handle environment-specific configurations and dependencies.

### **Advanced Tips**

- **Performance Optimization:** Utilize Bazel's caching and parallel execution features. Structure your Python modules and Bazel targets to maximize cache hits and parallelizable builds.
- **Custom Rules and Macros:** For complex or repetitive tasks, consider writing custom Bazel rules or macros. This strategy can encapsulate common patterns and enhance build maintainability.

## **[NodeJS/Typescript](#)**

In this section, we will dive into the world of NodeJS app development using Bazel as our build tool, Typescript and Express framework. This assumes that you are already familiar with Bazel, NodeJS, Typescript and Express development and all are already installed. We will explore

how to set up a Bazel-based NodeJS project, define build targets, and manage dependencies effectively.

In this section, we will delve into a specific set of code that can be found in our GitHub repository at <https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel>. The relevant code is located in the `/chapter-6/bazel_node_typescript` folder, which we will thoroughly explore and discuss.

## [Aspect Build](#)

Aspect Build is a software company that specializes in utilizing Bazel to streamline and enhance the development and build processes of various software projects. Aspect Build's projects involve the creation and implementation of custom build aspects, rules, and extensions tailored to specific programming languages and frameworks. These aspects are designed to improve code quality, dependency management, testing, and deployment workflows across a wide range of software development scenarios. By leveraging the power of Bazel's extensibility, Aspect Build aims to provide optimized and efficient build systems that enable developers to build, test, and deploy their applications more reliably and with better performance.

Using rules from Aspect Build instead of the official Bazel rules offers several key benefits. First, “`aspect.build`” rules are designed to be highly composable and extensible, allowing you to create custom build and analysis processes tailored to your project's specific needs. Second, they often provide a more user-friendly and declarative syntax, making it easier to define and understand complex build configurations. Finally, “`aspect.build`” rules are frequently more up-to-date and actively maintained than some of the official Bazel rules, ensuring that you have access to the latest features and improvements in the Bazel ecosystem.

These advantages make `aspect.build` rules a valuable choice for developers looking to streamline their Bazel-based build and automation workflows.

A key approach and rule from Aspect Build is `npm_link_all_packages`. The purpose of the `npm_link_all_packages` rule is to ensure that all npm packages listed in your project's `package.json` file are linked or made available in the `node_modules` directory within your Bazel workspace. This is essential because Bazel needs access to these dependencies during the build and test processes.

The rule `npm_link_all_packages(name = "node_modules")` is used to manage and link all the npm packages specified in the `package.json` file into the `node_modules` directory of your Bazel project. The name attribute ("`node_modules`" in this case) is used to define a target's name within the Bazel project. It provides a unique identifier for this rule that can be used to reference it in other parts of your Bazel configuration.

By using this rule, you can leverage Bazel's ability to create a hermetic and reproducible build environment. It ensures that all the necessary npm packages are available within your Bazel project, enabling consistent and predictable builds regardless of the external development environment. This is particularly useful in larger projects or when working in teams, as it helps eliminate potential discrepancies between development and production environments.

## [Setting up your NodeJS/Typescript Environment](#)

Before we dive into the setup, make sure you have the following tools installed on your system: Node.js and npm (Node Package Manager). You will need Node.js to run JavaScript and npm to manage Node.js packages. You can download and install them from the official Node.js website

(<https://nodejs.org/>). Let us check their availability by running:

```
node -v
v18.18.2

npm -v
9.8.1
```

## Organizing Your NodeJS/Typescript Project

By structuring your Android project into several Bazel projects, you enhance modularity, facilitate code reuse, and boost build efficiency. These projects should be organized in a way that distinguishes between applications, shared functional code, and shared non-functional code, like architectural components. We will go deeper into this topic in the forthcoming chapters.

Given the straightforward nature of this project, we will streamline the structure in the following manner:

```
.
├── BUILD.bazel
├── MODULE.bazel
├── WORKSPACE
├── app
│   ├── BUILD.bazel
│   ├── app.ts
│   ├── main.test.ts
│   └── main.ts
├── jest.config.js
├── common
│   ├── BUILD.bazel
│   ├── index.test.ts
│   └── index.ts
└── package.json
```



```
|— pnpm-lock.yaml
└— tsconfig.json
```

As in previous language examples, we have `/app` hosting the application code and `/common` the non-functional or shared one. The app itself is a silly REST API service that returns a number when called `GET /`. The number is stored in the common library.

The following files hold some key insights:

```
.bazelrc
```

We are going to configure a `.bazelrc` file as:

```
common --enable_bzlmod
build --@aspect_rules_ts//ts:skipLibCheck=honor_tsconfig
fetch --@aspect_rules_ts//ts:skipLibCheck=honor_tsconfig
query --@aspect_rules_ts//ts:skipLibCheck=honor_tsconfig
```

The `build --@aspect_rules_ts//ts:skipLibCheck=honor_tsconfig` configures Bazel's behavior when building targets. It specifies an aspect from the `@aspect_rules_ts` repository. Aspects in Bazel are used to modify or analyze the build process for specific targets. In this case, the aspect is named `ts:skipLibCheck`, and it is instructed to `honor_tsconfig`.

- `@aspect_rules_ts//ts:skipLibCheck` is a reference to an aspect defined in the `@aspect_rules_ts` repository.
- `honor_tsconfig` is an option for the `ts:skipLibCheck` aspect. This configuration tells Bazel to honor the `skipLibCheck` setting from the TypeScript configuration file (`tsconfig.json`) when building TypeScript targets. If `skipLibCheck` is set to `true` in your `tsconfig.json`, Bazel will skip type checking of library declaration files during the build.

The `fetch --@aspect_rules_ts//ts:skipLibCheck=honor_tsconfig` configures Bazel's behavior when fetching external dependencies. It also references the same `ts:skipLibCheck` aspect from the `@aspect_rules_ts` repository and tells Bazel to

**honor\_tsconfig.** This means that when Bazel fetches external dependencies, it will consider the `skipLibCheck` setting in the `tsconfig.json` file of those dependencies. If the setting is set to `true`, Bazel will skip type checking for those dependencies during the fetch process.

Finally, `query` `--@aspect_rules_ts//ts:skipLibCheck=honor_tsconfig` configures Bazel's behavior when querying the build graph. Similar to the previous lines, it references the `ts:skipLibCheck` aspect from the `@aspect_rules_ts` repository and specifies `honor_tsconfig`. When you use Bazel's query functionality to inspect the build graph, it will take into account the `skipLibCheck` setting in the `tsconfig.json` files of targets and dependencies.

### **tsconfig.json**

We are going to provide a `tsconfig.json` to configure typescript transpilation as follows:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "declaration": true,
    "strict": true,
    "baseUrl": ".",
    "paths": {
      "@org/common": ["/common/"]
    }
  },
  "exclude": ["bazel-*"]
}
```

It specifies various options that control how TypeScript compiles and checks your TypeScript code. Let us break down each section of this `tsconfig.json` file:

### **Compiler Options**

- **“target”**: **“ES2022”**: This option sets the target ECMAScript version for the TypeScript compiler. In this case, it is set to ES2022, which means TypeScript will allow you to use features from ECMAScript 2022 when writing your code. It also indicates the JavaScript version that TypeScript will transpile your code to if your code uses newer ECMAScript features.
- **“module”**: **“commonjs”**: This option specifies the module system to use when generating code. Here, **“commonjs”** is chosen, which is the module system commonly used in Node.js environments. This setting is essential to ensure compatibility with Node.js’s `require` and `module.exports` system.
- **“declaration”**: **true**: When this option is set to true, TypeScript generates declaration files (**.d.ts** files) for your code. Declaration files are used for type checking in your project and can also be published alongside your JavaScript code to provide type information to consumers of your library.
- **“strict”**: **true**: Enabling **“strict”** sets several strict type-checking options. This includes options like **“noImplicitAny”**, **“strictNullChecks”**, and others, which help catch type-related errors at compile-time and enforce better code quality.
- **“baseUrl”**: **“.”**: This option specifies the base directory for module resolution. When using TypeScript modules, you can use module aliases to simplify imports.
- **“baseUrl”**: **“.”** indicates that the root directory of your project should be considered as the base for module resolution.
- **“paths”**: This section allows you to define module resolution aliases. In your case, you have defined an alias for **“@org/common”** that points to **“./common/”**. This means that you can import modules using **“@org/common”**

in your TypeScript code, and TypeScript will resolve them to the `./common/` directory.

### **exclude**

- `"exclude": ["bazel-*"]`: This section specifies patterns for files and directories that TypeScript should exclude from compilation. In this case, it is excluding any files or directories with names starting with `"bazel-"`. This is a common practice when using Bazel for building and managing projects, as Bazel often handles the compilation of TypeScript files on its own.

### **.npmrc**

The `.npmrc` file contains a configuration setting `hoist=false` that controls the behavior of npm regarding package installation and dependency resolution. When you set `hoist` to `false`, you are essentially telling npm not to use the package hoisting feature. Package hoisting is a mechanism in npm that helps manage and optimize the installation of dependencies in a project.

By setting `hoist` to `false`, you disable this feature, which means that npm will not attempt to move common dependencies up to a higher level in the directory tree to reduce duplication. Instead, each package within your project will maintain its own copy of dependencies, potentially increasing the disk space used but providing more isolation between packages. This can be useful in situations where strict isolation between dependencies is required, but it might also result in larger disk usage if multiple packages use the same dependencies.

However, it is important to note that disabling hoisting (`hoist=false`) can lead to longer installation times, especially for projects with many dependencies, as npm will need to install and maintain separate copies of shared dependencies for each package. Therefore, you should carefully consider

your project's requirements and constraints before making a decision on whether to enable or disable package hoisting in your `.npmrc` configuration.

#### `.swcrc`

The `.swcrc` file, also known as a “**SWC configuration**” file, is used to configure the behavior of the SWC (Speedy Web Compiler) tool, which is a JavaScript/TypeScript transpiler. This file allows developers to customize how their code is transformed and compiled by SWC.

```
{
  "jsc": {
    "loose": true,
    "target": "es2020",
    "parser": {
      "syntax": "typescript",
      "decorators": true
    },
    "transform": {
      "legacyDecorator": true,
      "decoratorMetadata": true
    }
  },
  "module": {
    "type": "commonjs",
    "strict": true,
    "strictMode": true,
    "lazy": true,
    "noInterop": true
  }
}
```

In the `.swcrc` file, the configuration is organized into two main sections: “`jsc`” and “`module`”. The “`jsc`” section deals with the settings related to the JavaScript compiler (`jsc`)

itself, while the `“module”` section specifies how modules should be handled.

In the `“jsc”` section, several options are set. The `“loose”` option is set to true, indicating that SWC should apply loose transformations, which can result in more concise code but may not strictly adhere to certain JavaScript/TypeScript rules. The `“target”` is specified as `“es2020”`, which sets the target ECMAScript version for the transpiled code. Additionally, the `“parser”` section configures the parser used by SWC for TypeScript, enabling decorators. The `“transform”` section includes options like `“legacyDecorator”` and `“decoratorMetadata”`, which control how decorators in the code are handled during transformation.

In the `“module”` section, module-related settings are defined. `“type”` is set to `“commonjs”`, indicating that the code should be compiled using the CommonJS module system. `“strict”` and `“strictMode”` are set to true, enforcing strict mode in the generated code. `“lazy”` is set to true, suggesting that SWC should use lazy evaluation when handling modules. Finally, `“noInterop”` is set to true, which disables module interop, potentially resulting in smaller bundle sizes.

#### **package.json**

This file defines the configuration and dependencies for a Node.js project named `“aspect-typescript.”` In the `“devDependencies”` section, it lists packages necessary for development purposes, such as type definitions for **Express**, **Jest**, **Node.js**, and **Supertest**.

```
{
  “name”: “aspect-typescript”,
  “devDependencies”: {
    “@types/express”: “4.17.17”,
    “@types/jest”: “29.4.0”,
    “@types/node”: “18.14.0”,
    “@types/supertest”: “2.0.12”,
```

```

    "prettier": "2.8.4",
    "supertest": "6.3.3",
    "typescript": "4.9.5"
  },
  "dependencies": {
    "jest-cli": "^29.7.0",
    "express": "4.18.2",
    "jest-junit": "15.0.0"
  },
  "prettier": {
    "printWidth": 100,
    "tabWidth": 2,
    "singleQuote": true,
    "trailingComma": "all",
    "bracketSpacing": false,
    "arrowParens": "avoid",
    "semi": false
  }
}

```

Furthermore, it includes development tools like Prettier for code formatting and TypeScript for type-checking and compiling TypeScript code. The `"dependencies"` section contains the essential runtime dependency, which is Express, a popular Node.js web application framework. Lastly, there is a `"prettier"` configuration object that specifies code formatting settings, such as line width, tab width, and whether to use single quotes or trailing commas.

### **BUILD.bazel**

The first few lines contain load statements that import external rules and functions used in this build file. These statements bring in rules and functions from different Bazel repositories, such as npm dependencies management, TypeScript configuration, npm package linking, copying files, and Kubernetes object generation.

The package block defines the default visibility for targets within the package. In this case, it sets the default visibility to public, meaning that the targets defined in this package are visible to other packages.

```
load("@npm//:defs.bzl", "npm_link_all_packages")
load("@aspect_rules_ts//ts:defs.bzl", "ts_config")
load("@aspect_rules_js//npm:defs.bzl", "npm_link_package")
load("@aspect_bazel_lib//lib:copy_to_bin.bzl", "copy_to_bin")
package(default_visibility = ["//visibility:public"])
npm_link_package(
  name = "node_modules/@org/common",
  src = "//common",
)
ts_config(
  name = "tsconfig",
  src = ":tsconfig.json",
)
copy_to_bin(
  name = "swcrc",
  srcs = [".swcrc"],
)
copy_to_bin(
  name = "jest.config",
  srcs = ["jest.config.js"],
  visibility = ["//:__subpackages__"],
)
npm_link_all_packages(name = "node_modules")
```

The subsequent rules include:

- **npm\_link\_package:** This rule links a package named `@org/common` from the `node_modules` directory to a target named `//common`. It helps manage npm dependencies in the project.



- **ts\_config**: This rule defines a target named “**tsconfig**” and specifies that its source is the **tsconfig.json** file in the current directory. It is used to configure TypeScript settings for the project.
- **copy\_to\_bin**: Two instances of this rule are defined. One for copying the **.swcrc** file and another for copying the **jest.config.js** file into the binary output directory.
- **npm\_link\_all\_packages**: This rule ensures that all npm packages listed in the package.json file are linked into the **node\_modules** directory.

In summary, this **build.bazel** file sets up rules and dependencies for managing npm packages, TypeScript configuration, copying specific files, and generating Kubernetes objects within a Bazel project. These rules and settings are essential for building and deploying a Node.js/TypeScript application using Bazel.

#### **app/BUILD.bazel**

This is a Bazel build configuration file for a typescript project, and it defines various rules and targets to build and deploy a **Node.js/TypeScript** application.

```
load("@aspect_rules_swc//swc:defs.bzl", "swc")
load("@aspect_rules_ts//ts:defs.bzl", "ts_project")
load("@bazel_skylib//lib:partial.bzl", "partial")
load("@aspect_rules_jest//jest:defs.bzl", "jest_test")

ts_project(
  name = "app",
  srcs = glob(
    include = ["*.ts"],
    exclude = ["*.test.ts"],
  ),
  declaration = True,
  transpiler = partial.make(
    swc,
    swcrc = "://:swcrc",
```

```
),
tsconfig = "://:tsconfig",
deps = [
  "://:node_modules/@org/common",
  "://:node_modules/@types/express",
  "://:node_modules/@types/node",
  "://:node_modules/express",
],
)
ts_project(
  name = "app_test",
  srcs = glob(["*.test.ts"]),
  declaration = True,
  transpiler = partial.make(
    swc,
    swcrc = "://:swcrc",
  ),
  tsconfig = "://:tsconfig",
  deps = [
    ":app",
    "://:node_modules/@types/jest",
    "://:node_modules/@types/supertest",
    "://:node_modules/supertest",
  ],
)
jest_test(
  name = "test",
  config = "://:jest.config",
  data = [":app_test"],
  node_modules = "://:node_modules",
)
js_binary(
  name = "main",
  data = [":app"],
  entry_point = "main.js",
```

)

Let us break down each part of the file to understand its purpose:

- **Loading Rules and Definitions:** The file starts by loading various Bazel rules and definitions from different Bazel repositories. These rules and definitions are essential for building and deploying the project. Here are some of the loaded rules:
  - `@aspect_rules_swc//swc:defs.bzl`: Rules related to the SWC (Super-fast JavaScript to JavaScript Compiler).
  - `@aspect_rules_ts//ts:defs.bzl`: Rules related to TypeScript projects.
  - `@aspect_rules_js//js:defs.bzl`: Rules related to JavaScript projects, including binary and image layer rules.
  - `@bazel_skylib//lib:partial.bzl`: Partial rule used to define a transpiler (SWC) with specific configurations.
  - `@aspect_rules_jest//jest:defs.bzl`: Rules related to Jest tests.
- **ts\_project Rules:** This section defines two TypeScript projects:
  - **app**: The main TypeScript application. It specifies TypeScript source files, transpiler options (using SWC), TypeScript configuration (`tsconfig`), and dependencies on various Node.js and TypeScript packages.
  - **app\_test**: The TypeScript project for running tests. It also specifies TypeScript source files, transpiler options, TypeScript configuration, and dependencies. It depends on the `app` target and includes testing-related dependencies.

- **jest\_test** Rule: This rule defines a Jest test target named “test”. It specifies the Jest configuration, test data (the **app\_test** target), and the location of Node.js modules.
- **js\_binary** Rule: This rule defines a JavaScript binary target named “main”. It specifies the entry point for the JavaScript binary as “main.js” and depends on the **app** target.

This **BUILD.bazel** file is comprehensive and orchestrates the build, testing, and deployment of a Node.js/TypeScript application. It leverages various Bazel rules and dependencies to achieve this automation and reproducibility in the development and deployment process.

#### **common/BUILD.bazel**

This Bazel build file defines how the common library is going to be built:

```
load("@aspect_rules_swc//swc:defs.bzl", "swc")
load("@aspect_rules_ts//ts:defs.bzl", "ts_project")
load("@aspect_rules_js//npm:defs.bzl", "npm_package")
load("@bazel_skylib//lib:partial.bzl", "partial")
load("@aspect_rules_jest//jest:defs.bzl", "jest_test")

ts_project(
  name = "common_project",
  srcs = glob(
    include = ["*.ts"],
    exclude = ["*.test.ts"],
  ),
  declaration = True,
  transpiler = partial.make(
    swc,
    swcrc = "://:swcrc",
  ),
  tsconfig = "://:tsconfig",
)
```

```

ts_project(
  name = "common_project_test",
  srcs = glob(["*.test.ts"]),
  declaration = True,
  transpiler = partial.make(
    swc,
    swcrc = "://:swcrc",
  ),
  tsconfig = "://:tsconfig",
  deps = [
    ":common_project",
    "://:node_modules/@types/jest",
  ],
)

jest_test(
  name = "test",
  config = "://:jest.config",
  data = [":common_project_test"],
  node_modules = "://:node_modules",
)

npm_package(
  name = "common",
  srcs = [":common_project"],
  package = "@org/common",
  visibility = [":visibility:public"],
)

```

Let us break down what each section of the **BUILD** file does:

- Loading Rules: The load statements at the beginning of the file import external rules and macros from various Bazel repositories. These rules define how to build and process different aspects of your project. Here is what each load statement does:
  - `@aspect_rules_swc//swc:defs.bzl` loads rules related to the **SWC** (Super-fast, Babel-compatible

JavaScript/TypeScript compiler).

- `@aspect_rules_ts//ts:defs.bzl` loads rules for TypeScript projects.
  - `@aspect_rules_js//npm:defs.bzl` loads rules for handling npm packages.
  - `@bazel_skylib//lib:partial.bzl` loads a rule called `partial` from the Bazel Skylib library. This rule is used for partial execution of certain actions.
  - `@aspect_rules_jest//jest:defs.bzl` loads rules for running Jest tests.
- `ts_project` Rule - `common_project`: This rule defines a TypeScript project named `"common_project"`. It specifies the source files for this project using the `glob` function, which includes all `.ts` files in the current directory except those with a `.test.ts` extension. The `declaration = True` indicates that TypeScript declaration files (`.d.ts`) should be generated during the build. The `transpiler` attribute specifies how TypeScript files should be transpiled. It uses the SWC transpiler, and the configuration for SWC is defined in the `//:swcrc` file. The `tsconfig` attribute specifies the TypeScript configuration file to use, which is `//:tsconfig` in this case.
  - `ts_project` Rule - `common_project_test`: This rule defines another TypeScript project named `"common_project_test"`, specifically for test files (`.test.ts`). It includes all `.test.ts` files in the current directory. Like the previous rule, it generates TypeScript declaration files (`declaration = True`) and uses the SWC transpiler with the same SWC configuration. It has dependencies on two targets: `:"common_project"` (the non-test TypeScript code) and `//:node_modules/@types/jest` (TypeScript typings for Jest).
  - `jest_test` Rule - `test`: This rule defines a Jest test suite named `"test"`. It specifies the Jest configuration file

using `config = "//:jest.config"`. The data attribute lists the targets that should be tested. In this case, it tests `":common_project_test"`. The `node_modules` attribute specifies the location of npm dependencies, which is `//:node_modules` in this case.

- `npm_package` Rule - `common`: This rule defines an npm package target named `"common"`. It includes the `":common_project"` as its source, which means that the TypeScript code from `"common_project"` will be included in the npm package. The `package = "@org/common"` specifies the npm package name that will be published to the npm registry under the `@org` scope. The `visibility` attribute makes this target publicly visible.

In summary, this BUILD file configures Bazel to build and test a Node.js/TypeScript project. It uses the SWC `transpiler` for TypeScript compilation, Jest for testing, and defines an npm package target for publishing the project as an npm package. The dependencies and configurations are set up to ensure that TypeScript code and tests are processed correctly.

## [Building and Running Your NodeJS/Typescript App](#)

Before you can build your TypeScript code, you need to `install` dependencies. Run the following command to install the required npm packages:

```
npm install
```

This will read the `package.json` file generated by Bazel and install the necessary dependencies.

You are now ready to build your TypeScript code using Bazel. Run the following command to build the `"my_library"` target:

```
bazel build //app:main
```

This will compile your TypeScript code and generate output in the `dist` directory.

Now, you can run your code using Bazel:

```
bazel run //src:main
```

Then, open a web browser at `http://localhost:3000`.

Congratulations! You have successfully set up your Node.js/TypeScript environment with Bazel. You can now start building and testing your Node.js applications efficiently using the power of Bazel.

## **Best Practices Using NodeJS/Typescript in Bazel**

These practices will help you efficiently manage your Node.js projects, streamline your builds, and enhance your development experience.

### **Project Structure and Workspace Setup**

One of the first steps in setting up a Node.js and TypeScript project in Bazel is defining a well-organized project structure. It is essential to create a clear separation between your source code and build configuration. Consider segregating visual components, functional behaviors, and non-functional code by using different projects. Keep your TypeScript source code under the “`src`” directory and include your `BUILD.bazel` file at the root. Ensure that your `MODULES.bazel` file contains all necessary Bazel dependencies and external repository rules.

### **Fine-Grained Build Targets**

When defining build targets in Bazel for your Node.js and TypeScript project, it is advisable to be as fine-grained as possible. Instead of creating a single large build target for your entire project, break it down into smaller, more focused targets. This allows Bazel to optimize builds by only



rebuilding the parts of your project that have changed, reducing build times.

### **Using Bazel Rules for Node.js and TypeScript**

Leverage Bazel's Node.js and TypeScript rules to define your build targets. The `ts_library`, `ts_binary`, and `nodejs_binary` rules are powerful tools for managing TypeScript and Node.js code in Bazel. These rules ensure that your TypeScript code is transpiled correctly and that dependencies are managed efficiently.

### **Dependency Management with npm**

Node.js projects often rely on npm or Yarn for dependency management. It is crucial to integrate these package managers with Bazel effectively. Use the `npm_install` or `yarn_install` rules to fetch and manage your project's Node.js dependencies. This assures that Bazel is aware of your project's external dependencies and can cache them appropriately.

### **Bazel-Managed Development Server**

For local development, consider setting up a Bazel-managed development server using the `ts_devserver` rule. This rule allows you to run your Node.js server locally while still benefiting from Bazel's caching and build optimizations. It is an excellent way to streamline your development workflow and maintain consistency between local development and production builds.

### **Continuous Integration with Bazel**

Integrate Bazel into your continuous integration (CI) pipeline to ensure consistent and reproducible builds across different environments. Use tools like `Bazelisk` to make it easy to install and use the correct version of Bazel in your CI jobs. This confirms that your Node.js and TypeScript projects are built consistently across development, testing, and production environments.

## **Testing and Code Quality Checks**

Integrate testing and code quality checks into your Bazel build process. Use Bazel rules like `jest_test` or `tslint` to run your tests and perform static code analysis. By incorporating these checks into your Bazel build, you can catch issues early in your development cycle and maintain high code quality standards.

## **Monitoring and Optimization**

Monitor and optimize your Bazel builds regularly. Use tools like Bazel's `query` command to inspect build dependencies and identify potential bottlenecks. By fine-tuning your Bazel configuration and optimizing your build targets, you can significantly improve build performance for your Node.js and TypeScript projects.

Incorporating these best practices into your Node.js and TypeScript projects with Bazel will help you achieve more efficient builds, maintain code quality, and streamline your development workflow. By following these guidelines, you can harness the power of Bazel to enhance your Node.js development experience.

## **Golang**

We will explore the process of setting up Bazel for Go development, creating and configuring Go targets, managing dependencies, and optimizing your build process. By the end of this chapter, you will have a solid understanding of how to harness the power of Bazel to supercharge your Go projects and streamline your development workflow.

So, without further ado, let us dive into the world of building Go projects with Bazel and discover how this dynamic duo can revolutionize your development experience.

## Setting up Your Golang Environment

Before we get started, ensure that you have a working Go installation. Download and install Go from the official Go website. Make sure to set up your GOPATH and GOBIN environment variables as per the Go installation instructions.

## Organizing Your Golang Project

Bazel offers great flexibility when it comes to structuring your project, but adhering to certain conventions can simplify the development process. We will use a sample project structure to illustrate how you can effectively organize your Go project with Bazel.

```
.
├── BUILD.bazel
├── MODULE.bazel
├── WORKSPACE
├── common
│   └── greetings
├── deps.bzl
├── go.mod
├── go.sum
└── src
    ├── app
    └── handlers
```

Now, let us break down each component of this structure and discuss how to effectively use it in your Go project.

### **/BUILD.bazel**

The **BUILD.bazel** file is used to define rules and targets for building Go code within the directory it resides.

```
load("@gazelle//:def.bzl", "gazelle")
gazelle(name = "gazelle")
```

This `BUILD.bazel` file uses Gazelle, a Bazel tool, to generate and manage Bazel build files for a Go project. Gazelle automatically generates Bazel build files for Go projects by scanning the project's source code directories, identifying Go packages and their dependencies, and then creating corresponding `BUILD.bazel` files with rules that define how to build and test the Go packages. This automation helps ensure that your Go project is properly integrated with Bazel, making it easier to build, test, and manage dependencies within the Bazel ecosystem.

```
/src/app/BUILD.bazel
```

This `BUILD.bazel` file defines how to build an app that exposes two REST endpoints.

```
load("@rules_go//go:def.bzl", "go_binary", "go_library")
go_library(
  name = "app_lib",
  srcs = ["main.go"],
  importpath = "github.com/OrangeAVA/Building-Large-Scale-Apps-
with-Monorepo-and-Bazel/chapter-6/bazel_go/src/app",
  visibility = ["//visibility:private"],
  deps = [
    "//src/handlers",
    "@com_github_gorilla_mux//:mux",
  ],
)
go_binary(
  name = "app",
  embed = [":app_lib"],
  visibility = ["//visibility:public"],
)
```

This `BUILD.bazel` file sets up a Go library target (`app_lib`) and a Go binary target (`app`) for a Go project. The library target compiles the `main.go` source file and depends on both internal and external dependencies, while the binary

target embeds the library and is marked as publicly visible. This configuration allows you to build and run the Go application using Bazel.

**/src/handlers/BUILD.bazel**

This **BUILD.bazel** file defines how to build a library for the endpoint's handlers.

```
load("@rules_go//go:def.bzl", "go_library")
go_library(
    name = "handlers",
    srcs = ["handler.go"],
    importpath = "github.com/OrangeAVA/Building-Large-Scale-Apps-
with-Monorepo-and-Bazel/chapter-6/bazel_go/src/handlers",
    deps = [ "//common/greetings" ],
    visibility = [ "//visibility:public" ],
)
```

This **BUILD.bazel** file defines a Go library target called **"handlers"** that includes the **"handler.go"** source file, has a specified import path, depends on the **"greetings"** package from the **"//common/greetings"** target, and is publicly visible to other parts of the project. This target can be used by other Bazel targets or Go applications within the same Bazel workspace.

**/common/greetings/BUILD.bazel**

This **BUILD.bazel** file defines how to build a library for a shared function including tests.

```
load("@rules_go//go:def.bzl", "go_library", "go_test")
go_library(
    name = "greetings",
    srcs = ["greetings.go"],
    importpath = "github.com/OrangeAVA/Building-Large-Scale-Apps-
with-Monorepo-and-Bazel/chapter-6/bazel_go/common/greetings",
    visibility = [ "//visibility:public" ],
)
```

```
go_test(  
  name = "greetings_test",  
  srcs = ["greetings_test.go"],  
  embed = [":greetings"],  
)
```

This **BUILD.bazel** file defines a Go package named "greetings" with its source code in "greetings.go" and a corresponding test target named "greetings\_test" that tests the "greetings" package. The test is able to access and test the code in the "greetings" package because it is embedded as a dependency using the embed attribute. This structure is commonly used in Bazel to organize and test Go code within a project.

### **MODULE.bazel**

The **MODULE.bazel** file is often used to specify dependencies and other module-level configurations.

```
bazel_dep(name = "rules_go", version = "0.43.0")  
bazel_dep(name = "gazelle", version = "0.34.0")  
go_deps = use_extension("@gazelle//:extensions.bzl",  
  "go_deps")  
go_deps.from_file(go_mod = "///:go.mod")  
use_repo(  
  go_deps,  
  "com_github_gorilla_mux"  
)
```

The **MODULE.bazel** file is configuring dependencies for a Bazel project that uses Go as its programming language. It specifies dependencies on Bazel extensions like **rules\_go** and **gazelle**, configures the **go\_deps** extension to read the project's **go.mod** file, and then declares a repository dependency on gorilla/mux using the **use\_repo** function. These dependencies and configurations are essential for managing the Go dependencies and building the project with Bazel.

## **WORKSPACE**

The **WORKSPACE** file is essential in any Bazel project, but in this case is empty as all dependencies are defined in **MODULE.bazel** file.

### **common**

The **common** directory can be used to store shared code that multiple parts of your project may utilize. In the sample structure, we have a subdirectory called **greetings**, which could contain shared Go code related to greeting messages, for example,

### **deps.bzl**

The **deps.bzl** file is a Bazel-specific file that can be used to define custom build rules or functions. It can be handy if your project requires complex build logic that is not easily expressed in the standard Bazel **BUILD** files.

### **go.mod** and **go.sum**

These files are standard Go module files, used for managing your project's dependencies. You should keep them in the root of your project to ensure proper dependency management when using Go modules.

### **src**

The **src** directory is where your project's Go source code resides. In this structure, we have two subdirectories:

- **app**: This directory can contain the main application code. Typically, this is where your entry point to the application is located.
- **handlers**: This directory can contain HTTP request handlers or other types of handlers specific to your project.

By structuring your Go project following the sample structure and incorporating Bazel's **BUILD** files, you can efficiently manage your project's dependencies and build

process. Bazel's flexibility allows you to adapt this structure to fit the specific needs of your project while maintaining a clean and organized codebase. In the upcoming sections, we will dive deeper into how to define **BUILD** files, declare dependencies, and build your Go project effectively with Bazel.

## [Building and Running Your Golang App](#)

First, run the command `bazel run @rules_go//go -- mod tidy -v`. This command uses Bazel to execute a Go module management command. It begins with `bazel run`, which instructs Bazel to run a specific target. The target, `@rules_go//go`, is a reference to a Bazel rule for Go, likely indicating that this command is executing within a Go-related Bazel workspace. Following `--`, the actual Go command `mod tidy -v` is provided. This command instructs Go's module system to tidy the project's dependencies (`mod tidy`) while displaying verbose output (`-v`). Bazel facilitates the execution of this Go command, ensuring that the correct environment and dependencies are in place within the Bazel workspace, making it a part of a reproducible and consistent build process.

To build your GoLang application with Bazel, run the `build` command:

```
bazel build ...
```

This command will use Bazel to compile your GoLang code into an executable binary. The output binary will be located in the Bazel `bazel-bin` directory.

Run the compiled binary:

```
./bazel-bin/src/app/app_/app
```

You should see the output `"server started at port :5000"` displayed in your terminal. If you open



**http://localhost:5000/greet** in a browser you will get:

```
Hi, Javier. Welcome!
```

Run the tests using Bazel:

```
bazel test ...
```

Bazel will compile and run all the tests, and you will see the test results in your terminal.

```
bazel test ...
```

```
INFO: Analyzed 5 targets (81 packages loaded, 11193 targets configured).
```

```
INFO: Found 4 targets and 1 test target...
```

```
INFO: Elapsed time: 91.753s, Critical Path: 72.68s
```

```
INFO: 29 processes: 11 internal, 18 local.
```

```
INFO: Build completed successfully, 29 total actions
```

```
//common/greetings:greetings_test PASSED in 0.2s
```

```
Executed 1 out of 1 test: 1 test passes.
```

```
There were tests whose specified size is too big. Use the --test_verbos_timeout_warnings command line option to see which ones these are.
```

We have covered the basics of building, testing, and running a GoLang application with Bazel. Bazel provides a robust and reproducible build environment for GoLang projects, making it a valuable tool for managing your development workflow. You can extend these concepts to more complex GoLang projects and take advantage of Bazel's scalability and performance optimization features. Happy coding!

## **Best Practices Using Golang in Bazel**

### **Use Gazelle**

Firstly, it streamlines the integration of Go projects with Bazel, a powerful build and test tool, by automatically generating the necessary Bazel build files (**BUILD.bazel** or **BUILD** files) for Go packages. This reduces the manual effort involved in writing and updating these files, especially for

large projects with many dependencies. Gazelle understands the Go project structure and resolves external dependencies by fetching them from their sources, ensuring that the build files are always up-to-date with the project's dependencies and structure. This automation not only enhances productivity but also minimizes the chances of human error in build configurations. Furthermore, Gazelle supports a workflow that is familiar to Go developers, aligning well with Go's own tooling and conventions, thus making the adoption of Bazel within the Go ecosystem smoother and more intuitive.

### **Modularize Your Codebase**

Divide your GoLang application into smaller, reusable modules or packages. Each module should have a clear responsibility and well-defined API. Bazel works best with fine-grained dependencies, so keeping your code modular will improve build performance and maintainability.

### **Use Bazel's Go Rules**

Leverage Bazel's built-in Go rules, such as `go_binary`, `go_library`, and `go_test`, which simplify the definition of Go targets and their dependencies. These rules ensure that Bazel can efficiently manage your GoLang code and its dependencies during the build process.

### **Vendor Dependencies**

Consider using the Go module system to manage your project's dependencies. Bazel can integrate with Go modules to fetch and cache dependencies, reducing the need to commit them to your repository. This keeps your repository clean and reduces storage requirements.

### **Module.bazel and Workspace Configuration**

Configure your Bazel Module and Workspace files with `BUILD` files that describe your project's structure. Make sure your GoLang packages are properly defined in these `BUILD` files,

and specify dependencies explicitly. This ensures that Bazel can accurately determine the build order and manage dependencies efficiently.

### **Use Bazelisk for Version Management**

To ensure consistency across development environments, encourage your team to use Bazelisk to manage Bazel versions. This tool helps developers use the correct Bazel version specified in your project's configuration, preventing version-related build issues.

### **Selective Testing**

When running tests, use Bazel's selective test execution feature. This allows you to run only the tests affected by code changes, saving time and resources during development and continuous integration builds.

### **Leverage Remote Caching and Execution**

Take advantage of Bazel's remote caching and execution capabilities. Configure a remote cache and execution server to store build artifacts and execute builds remotely. This speeds up builds and allows for faster iteration.

### **CI/CD Integration**

Integrate Bazel into your CI/CD pipeline to ensure consistent and reproducible builds. Configure your CI system to use the same Bazel version as your development environment, and use Bazel's built-in support for various CI platforms.

### **Monitoring and Profiling**

Use Bazel's built-in profiling and monitoring tools to analyze build performance. Identify bottlenecks and optimize your build configurations to achieve faster build times.

### **Documentation and Onboarding**

Document your project's Bazel setup and best practices for new team members. Create an onboarding guide that explains how to work with Bazel in your GoLang project,

including common tasks like building, testing, and dependency management.

By following these best practices, you can streamline your GoLang development process with Bazel, improve build performance, and ensure a consistent and reproducible development environment for your team.

## iOS

Developing native iOS applications goes beyond merely compiling code in a different programming language; it encompasses an entire development ecosystem, including an integrated development environment (IDE), dependency management, and an execution platform unique to iOS. Spotify's experience, as shared in October 2023 (refer to <https://engineering.atspotify.com/2023/10/switching-build-systems-seamlessly/>), highlights their successful transition to Bazel. This shift allowed them to dramatically cut down full build times, reducing them from a lengthy 80 minutes to less than 20 minutes in most cases, with a remarkable 75% of builds completing in less than 30 seconds.

## Setting up Your iOS Environment

Before you start setting up your iOS environment with Bazel, make sure you have the following prerequisites in place:

- **macOS:** Bazel requires macOS as the development environment for iOS development. Ensure you have a Mac running a compatible version of macOS.
- **Xcode:** Install Xcode, which includes the necessary iOS SDKs and tools for iOS development. You can download Xcode from the Mac App Store.
- **iOS Simulator:** Make sure you have the iOS Simulator installed on your macOS machine. This is typically included when you install Xcode. Ensure you have

iPhone/iPad os installed following instructions at <https://developer.apple.com/documentation/safari-developer-tools/adding-additional-simulators>

## Organizing Your iOS Project

Every programming language comes with its own traditions when it comes to project structure and directory naming, and developers tend to hold onto these conventions regardless of the build tool they employ. In the realm of iOS development, it is a common practice to store source code in a ``/Sources`` directory and other related assets in a ``/Resources`` directory, and we will adhere to this convention.

Due to the project's simplicity, we will adopt a more simplified structure as outlined here:

```
.
├─ MODULE.bazel
├─ WORKSPACE
├─ app
│   └─ App.xcodeproj
│   └─ BUILD.bazel
│   └─ Resources
│   └─ Sources
```

### **MODULE.bazel**

```
bazel_dep(name = "rules_apple", version = "3.1.1")
bazel_dep(name = "rules_swift", version = "1.13.0")
bazel_dep(name = "rules_xcodeproj", version = "1.13.0")
bazel_dep(name = "bazel_features", version = "1.1.1")
```

These external rules enhance your project's capabilities by providing specialized features for iOS application management, Swift language code compilation, and Xcode project management.

### **WORKSPACE**

The **WORKSPACE** file is essential in any Bazel project, but in this case is empty as all dependencies are defined in **MODULE.bazel** file.

```
.bazelrc
```

We are going to configure a **.bazelrc** file as:

```
common --enable_bzlmod  
build --verbose_failures
```

In the provided **.bazelrc** file, there are two configuration options specified:

- **common --enable\_bzlmod**: This line configures Bazel to enable experimental support for Bazel modules (**bzlmod**). Bazel modules are a feature that allows you to organize and share build rules and dependencies in a more modular and reusable way. Enabling this feature indicates that you want to use and experiment with Bazel modules in your project, which can help improve build organization and reusability.
- **build --verbose\_failures**: This line configures Bazel to provide more detailed information when a build failure occurs. When this option is enabled, Bazel will display additional diagnostic information in the build output, making it easier to identify and understand the reasons behind build failures. This can be particularly helpful for debugging and troubleshooting issues in your project's build process.

**app/BUILD.bazel**

This **BUILD.bazel** file defines build targets for an iOS application using Bazel.

```
load("@rules_apple//apple:ios.bzl", "ios_application")  
load("@rules_swift//swift:swift.bzl", "swift_library")  
load( "@rules_xcodeproj//xcodeproj:defs.bzl",  
"top_level_target", "xcodeproj")
```

```
swift_library(  
  name = "lib",  
  srcs = glob(["Sources/*.swift"]),  
)  
  
ios_application(  
  name = "app",  
  bundle_id = "build.bazel.rules-apple-example",  
  families = [  
    "iphone",  
    "ipad",  
  ],  
  infoplists = ["Resources/Info.plist"],  
  minimum_os_version = "17.0",  
  visibility = ["//visibility:public"],  
  deps = [":lib"],  
)  
  
xcodproj(  
  name = "xcodproj",  
  build_mode = "bazel",  
  project_name = "App",  
  tags = ["manual"],  
  top_level_targets = [  
    ":app", target_environments = ["device", "simulator"]  
  ],  
)
```

Let us break it down:

**Loading Rules:** The first three lines of the file load specific Bazel rules using the ``load`` function. These rules come from external Bazel repositories and are essential for iOS development with Bazel. They include:

- ``ios_application`` defines an iOS application target.
- ``swift_library`` defines a Swift library target.
- ``xcodproj`` defines an Xcode project target.

**Swift Library Target:** The ``swift_library`` target named `"lib"` is defined to build a Swift library. It specifies the source files using the ``srcs`` attribute, which uses a glob pattern to include all Swift source files within the `"Sources"` directory.

**iOS Application Target:** The ``ios_application`` target named `"app"` is defined to build an iOS application. It has various attributes, including:

- ``bundle_id``: Specifies the bundle identifier for the app.
- ``families``: Lists the device families that the app supports (iPhone and iPad).
- ``infoplists``: Points to the `Info.plist` file that contains configuration information for the app.
- ``minimum_os_version``: Sets the minimum iOS version required to run the app.
- ``visibility``: Defines the visibility of the target as public, making it accessible to other parts of the project.
- ``deps``: Specifies dependencies, in this case, the `"lib"` target is a dependency, meaning the app depends on the Swift library.

**Xcode Project Target:** The ``xcodeproj`` target named `"xcodeproj"` is defined to generate an Xcode project for this Bazel project. It includes attributes like:

- ``build_mode``: Indicates the build mode as `"bazel,"` which means that Xcode will be configured to use Bazel for building.
- ``project_name``: Specifies the name of the generated Xcode project.
- ``tags``: Assigns tags to the target. In this case, it is tagged as `"manual."`
- ``top_level_targets``: Lists the top-level targets in the generated Xcode project, including the `"app"` target. It



also specifies target environments, which include “device” and “simulator.”

In summary, this `BUILD.bazel` file sets up a Bazel project for building an iOS application written in Swift. It defines targets for a Swift library, an iOS application, and an Xcode project. The iOS application depends on the Swift library, and the Xcode project is configured to use Bazel for building, including support for both device and simulator target environments.

```
/app/App.xcodeproj
```

This folder represents the Xcode project directory, and all its contents are automatically generated by the ‘xcodeproj’ target.

## **Building and Running Your iOS app**

Before you can start building and running your iOS app with Bazel, you need an Xcode project. Bazel can generate and maintain Xcode projects for you with just one simple command:

```
bazel run //app:xcodeproj
```

This command tells Bazel to create (and update, if necessary) an Xcode project for your iOS app. The `//app:xcodeproj` target specifies which part of your project should be used for generating the Xcode project. Once this command is executed, you will have a fully functional Xcode project that you can open in Xcode and work with as you normally would.

## **Building Your iOS App**

With the Xcode project in place, you can use Bazel to build your iOS app. Bazel’s build process is highly efficient and incremental, which means it only rebuilds what is necessary. To build your app, execute the following command in your project directory:

```
bazel build //app
```

The `//app` target specifies the part of your project you want to build. Bazel will analyze your project's dependencies, compile the necessary source code, and generate the app binary. Any subsequent builds will be faster because Bazel caches build artifacts intelligently.

You could find the built app in the following path:

```
bazel-bin/app/app.ipa
```

## **Launching the iOS Simulator**

Once you have built your iOS app, you will want to run it on the iOS Simulator for testing and debugging. Bazel simplifies this process as well:

```
bazel run //app
```

This command tells Bazel to launch the iOS Simulator with your app. Bazel handles the necessary setup and configuration, so you can focus on testing and debugging your application.

## **Build the App for a Device**

To enable the distribution of your app or its installation on a physical iOS device, you must correctly configure provisioning profiles and distribution certificates. If this topic seems complex, you can either skip it for now or revisit it later.

To prepare your app for installation and execution on an iOS device, Bazel requires the appropriate provisioning profile for the specific device model. Follow these steps:

1. Access your Apple Developer Account and obtain the necessary provisioning profile for your device. For detailed guidance, consult Apple's official documentation.
2. Relocate the downloaded provisioning profile to your project's workspace directory, typically referred to as

**\$WORKSPACE.**

3. Optionally, you can exclude the provisioning profile from version control by adding it to your `.gitignore` file.
4. Update your `BUILD` file by adding the following line to the `ios_application` target:

```
provisioning_profile =  
    "<your_profile_name>.mobileprovision"
```

Ensure that the `<your_profile_name>` matches the actual profile name required for successful installation on your target device.

5. Now, you can build the app for your iOS device using the following command:

```
bazel build //:iOSApp --ios_multi_cpus=arm64
```

This command generates the app as a universal binary. If you wish to build for a specific device architecture, you can specify it within the build options.

6. To build for a particular Xcode version, employ the `--xcode_version` option. For a specific SDK version, you can utilize the `--ios_sdk_version` option. Typically, using `--xcode_version` should suffice for most scenarios.
7. To establish a minimum required iOS version, introduce the `minimum_os_version` parameter into the `ios_application` build rule within your `BUILD` file.
8. Moreover, make sure to update the previously defined `xcodeproj` rule to indicate support for building for a device:

```
xcodeproj(  
    name = "xcodeproj",  
    build_mode = "bazel",  
    project_name = "iOSApp",  
    tags = ["manual"],  
    top_level_targets = [  
        ios_application(  
            name = "iOSApp",  
            srcs = glob(["**/*.swift", "**/*.m", "**/*.h", "**/*.plist", "**/*.xib", "**/*.xcassets"]),  
            ios_sdk_version = "11.0",  
            xcode_version = "10.2",  
            minimum_os_version = "11.0",  
            provisioning_profile = "  
                <your_profile_name>.mobileprovision"  
            )  
        ]  
    )  
)
```

```
    top_level_target(":iOSApp", target_environments =
      ["device", "simulator"]),
  ],
)
```

Note: For a more advanced provisioning profile integration, you can explore options like `provisioning_profile_repository` and `local_provisioning_profile` rules.

Now, to install your app on an iOS device, follow these steps:

1. The simplest approach is to launch Xcode and navigate to **Windows > Devices**. Then, select your connected device from the list on the left.
2. Add your app by clicking the “Add” (plus sign) button under the “**Installed Apps**” section. Select the `.ipa` file that you previously built.

If you encounter issues during the installation process, ensure that you have correctly specified the provisioning profile in your `BUILD` file (as described in step 4 of the previous section).

If your app fails to launch on your device, confirm that your device is included in the provisioning profile. Also, you can check the “**View Device Logs**” button on the Devices screen in Xcode for potential error information.

## [External Dependencies](#)

The iOS ecosystem also encompasses its own set of dependency management tools, which play a crucial role in handling project configurations.

One of the pioneering and widely adopted dependency managers in the iOS world is `Cocoapods`. It facilitates the integration of third-party libraries (referred to as “`Pods`”) as well as local pods. The specifications for these

dependencies are meticulously defined in a `podspec` file, typically written in Ruby. CocoaPods fetches source files from remote dependencies, but it is worth noting that this approach can substantially increase compilation times.

Another option is `Carthage`, which offers a cleaner approach to dependency management. Carthage generates a comprehensive framework that needs to be linked to your project. In most cases, these projects employ dynamic linking, necessitating the copying of frameworks into the final binary, potentially affecting launch times. Dependencies in Carthage are resolved using git tags, which prevents the creation of Monorepo projects.

For those seeking a streamlined project configuration process, `XcodeGen` comes into play. This tool automatically generates an Xcode project based on a single YAML or JSON file. It can be combined with Carthage and/or CocoaPods, offering flexibility. However, for large-scale projects, maintaining it can be challenging, as it relies on a single configuration file.

A similar approach to `XcodeGen` is offered by `Tuist`, which uses Swift files to configure the project. One notable feature is the ability to split the configuration into multiple files within the project folders, making maintenance significantly easier. Tuist provides an elegant Swift interface and offers numerous integrations with other dependency managers. It serves as a secure, native way to scale Xcode projects, while still relying on Xcode as the build system.

Bazel provides varying degrees of support for all these tools through rules. While a detailed exploration of these tools is beyond the scope of this book, you can readily discover examples and find support by conducting online searches.

## **[Using iOS Best Practices in Bazel](#)**

Using Bazel with iOS projects requires a nuanced understanding of both the build system and the iOS development ecosystem. This section aims to provide best practices for integrating Bazel into your iOS development workflow.

- **Structuring Your Workspace**

- **Modularize Your Code:** Divide your iOS project into smaller, reusable modules. Bazel excels at caching and parallelizing builds, so the more granular your modules, the more you benefit from these features.
- **Use a Consistent Directory Structure:** Maintain a clear and consistent directory structure. This helps in defining more streamlined BUILD files and makes it easier to manage dependencies.

- **Dependency Management**

- **Manage External Dependencies:** Utilize Bazel's workspace rules to handle external dependencies. For CocoaPods, consider using `rules_pods` to integrate them into your Bazel build.
- **Version Control for Dependencies:** Keep your dependencies pinned to specific versions. This ensures reproducibility of your builds.

- **Efficient Build Files**

- **Optimize BUILD Files:** Keep your BUILD files lean and focused. Define only what is necessary for each module, avoiding redundancy.
- **Use Macros Wisely:** Define macros for common build patterns. This reduces boilerplate and helps maintain consistency across your BUILD files.

- **Build Configurations**

- Leverage Configurable Builds: Make use of Bazel's ability to handle multiple build configurations. This is particularly useful for managing different environments like development, staging, and production.
- Use Apple's Build Settings Thoughtfully: Integrate Apple-specific build settings carefully. Bazel's sandboxed environment might handle some settings differently than Xcode.
- **Testing**
  - Integrate with XCTest: Leverage Bazel's rules for XCTest to ensure that your unit and UI tests are incorporated into your build process.
  - Test Isolation: Ensure that your tests are isolated and reproducible. Bazel's sandboxing can be very beneficial here.
- **Continuous Integration**
  - CI/CD Integration: Bazel is well-suited for CI/CD pipelines. Its ability to cache and parallelize builds can significantly reduce build times on CI servers.
  - Remote Caching: Utilize remote caching to share build artifacts among team members and CI systems, further reducing build times.
- **Stay Updated**
  - Keep Up with Bazel's Evolution: Bazel is actively developed. Stay updated with the latest releases and changes, especially those related to Apple's ecosystem.

## **Conclusion**

As we conclude this chapter on using Bazel with a variety of languages and platforms, it is important to reflect on the versatility and power that Bazel brings to the table. Throughout this chapter, we have explored how Bazel can be effectively used to build and test software across multiple languages and environments, including Android with Kotlin, Python, NodeJS with TypeScript, Golang, and iOS. This diversity showcases Bazel's adaptability and its ability to handle projects of varying complexity and size. Whether you are working on a multi-language web application, a complex mobile app, or a straightforward script, Bazel provides a consistent and efficient way to manage your builds.

Moreover, the examples provided throughout these pages illustrate the fundamental concepts and practical applications of Bazel in different contexts. By applying Bazel to a "Hello World" program in each of these languages and platforms, we have demonstrated how Bazel's core principles of reproducibility, speed, and scalability are maintained across different development environments. This not only enhances your workflow but also ensures that your projects remain manageable and efficient as they grow. As you continue your journey with Bazel, keep these examples in mind as a foundation upon which you can build more complex and sophisticated projects, leveraging Bazel's robust features to streamline your development process regardless of the language or platform you choose.

In the next chapter, we will delve into modern software development practices, emphasizing the enhancement of collaboration, efficiency, and quality in code contribution.

## **[Recommended Readings](#)**

As you read through this chapter, we explore the use of Bazel with a variety of languages and platforms, including



Android/Kotlin, Python, NodeJS/TypeScript, Golang, and iOS. The following readings are recommended to deepen your understanding and skill in these areas.

## **Android/Kotlin**

- “Bazel: The Revolutionary Build System for Android App Development” — This ProAndroidDev article provides a comprehensive look into using Bazel for Android development, particularly focusing on Kotlin. <https://proandroiddev.com/bazel-the-revolutionary-build-system-for-android-app-development-77d6ea340c51>
- “Building an Android App with Bazel” — Bazel’s official guide to starting with Android app development offers a solid foundation for beginners. <https://bazel.build/versions/6.1.0/start/android-app>

## **Python**

- “Build and Deploy PyApp with Bazel” — Earthly’s blog post delves into the specifics of building and deploying a Python application using Bazel, making it a must-read for Python developers. <https://earthly.dev/blog/build-and-deploy-pyapp-with-bazel/>
- Aspect Build: Rules for Python — This documentation from Aspect Build covers rules and best practices for using Bazel with Python projects. [https://docs.aspect.build/rulesets/rules\\_python/](https://docs.aspect.build/rulesets/rules_python/)

## **NodeJS/TypeScript**

- “Using Bazel with TypeScript” — An insightful article from Earthly that focuses on integrating TypeScript with Bazel. <https://earthly.dev/blog/using-bazel-with-typescript/>
- Aspect Rules Jest: Documentation — For those using Jest with Bazel, this documentation provides essential

information and guidelines.  
[https://docs.aspect.build/rulesets/aspect\\_rules\\_jest/docs/jest\\_test/](https://docs.aspect.build/rulesets/aspect_rules_jest/docs/jest_test/)

## Golang

- “Build Golang with Bazel and Gazelle” — This Earthly blog post offers a practical guide to using Bazel and Gazelle for Go projects. <https://earthly.dev/blog/build-golang-bazel-gazelle/>
- Bazel Go Rules Documentation — Directly from the Bazel Go GitHub repository, this documentation is a comprehensive resource for Go developers. [https://github.com/bazelbuild/rules\\_go/blob/master/docs/go/core/bzlmod.md](https://github.com/bazelbuild/rules_go/blob/master/docs/go/core/bzlmod.md)

## iOS

- “Building Faster in iOS with Bazel” — Sergio Fernandez’s Medium article provides insights into accelerating iOS development with Bazel. <https://medium.com/@fdzsergio/building-faster-in-ios-with-bazel-448a3074e73>
- “Using Bazel with Your iOS Projects” — This DEV.to article by PeterTech is a great resource for iOS developers beginning with Bazel. <https://dev.to/petertech/using-bazel-with-your-ios-projects-g7e>
- “iOS Framework Development with Bazel” — Baracoda’s blog post focuses on developing iOS frameworks using Bazel, offering valuable insights for advanced users. <https://baracoda.com/blog/ios-framework-bazel>
- Bazel Rules Apple: iOS App Tutorial — This tutorial from the Bazel GitHub repository is essential for anyone looking to build iOS apps with Bazel. [https://github.com/bazelbuild/rules\\_apple/blob/master/doc/tutorials/ios-app.md](https://github.com/bazelbuild/rules_apple/blob/master/doc/tutorials/ios-app.md)

Each of these readings will provide you with specialized knowledge and practical insights, helping you to effectively implement Bazel in your projects across different languages and platforms.

# CHAPTER 7

## Streamlining Development Workflow

### Introduction

In the dynamic world of software development, adapting and refining workflows is key to success. “*Streamlining Development Workflow*” delves into modern software development practices, emphasizing the enhancement of collaboration, efficiency, and quality in code contribution. It offers a thorough examination of tools and methods that have revolutionized team dynamics in shared codebases, from the impact of version control systems like Git to platforms like GitHub and GitLab. This chapter illuminates the technical progress and cultural shift towards more collaborative and transparent development, underscoring the significant role of Bazel in Monorepos for managing large-scale projects. It delves into code contribution traceability, the importance of effective code reviews, and the strategic use of CI and CD pipelines, showing how Bazel streamlines these aspects for more consistent and efficient integration of code contributions. This comprehensive chapter navigates the complexities of modern software projects, providing practical insights for enhancing development workflows with these advanced methodologies.

### Structure

In this chapter, the following topics will be covered:

- Code Contribution Workflows
- Setting up Continuous Integration with Bazel
- Enabling a Sort of Local CI with Bazel
- CI Worker Set up Models
- Managing Code Formatting, Linting, and Static Code Analysis

## **Code Contribution Workflows**

Code contribution refers to the process of adding or modifying code within a software project. This process is a cornerstone of collaborative software development, enabling multiple developers to work together on a shared codebase. Code contributions can range from small bug fixes to the addition of new features or significant architectural changes. The key aspect of code contribution is that it is a collaborative effort, often involving code reviews, version control, and adherence to project guidelines to ensure the quality and compatibility of the new code.

Over the years, the process of code contribution has evolved significantly, primarily due to advancements in tools and methodologies. In the early days of software development, code changes were often managed manually, leading to challenges in collaboration and version control. The introduction of centralized version control systems (VCS) like CVS and Subversion marked a significant improvement. However, the real transformation came with distributed version control systems, such as Git, which facilitated much more flexible and collaborative workflows. Additionally, the rise of online platforms like GitHub and GitLab revolutionized code contribution by making it easier for developers to collaborate, track issues, and review code.

In today's fast-paced software development environment, efficient and robust code contribution workflows are essential. These workflows are designed to streamline the process of integrating individual code contributions into a larger codebase, ensuring consistency, quality, and efficiency. Key elements of these workflows include version control systems, code review practices, continuous integration (CI) and continuous deployment (CD) pipelines, and, importantly, the use of build tools like Bazel.

### **Code Contribution Traceability**

The cornerstone of modern code contribution is a version control system like Git. Developers create branches to work on new features, bug fixes, or other improvements. Effective branching strategies, such as GitFlow or Trunk-Based Development, play a crucial role in managing the contributions of multiple developers and ensuring that the main codebase remains stable and deployable at all times.

During code contribution, traceability is the ability to trace changes back to their origin, including who made the change, why it was made, and how it impacts the codebase. This is crucial for maintaining the integrity of the software project and is typically facilitated through version control systems. By keeping a detailed history of changes, teams can better understand the evolution of their project, identify the causes of issues, and manage contributions from multiple developers more effectively.

In the realm of software development, the traceability of code contributions is a critical aspect of maintaining a coherent and efficient workflow. Commit comments play a pivotal role in this process, especially when contributions involve changes across multiple files. These comments serve as a key to understanding the motivations and rationale behind each code change, providing valuable

context not just for current team members, but also for future contributors who may be revisiting the code.

While documenting individual code contributions within a feature branch is beneficial in the short term, it is the comments made during merge requests that carry long-term value. Investing effort in detailed and informative merge request comments is crucial. These comments not only facilitate a smoother code review process but also become an integral part of the project's historical record. A well-documented merge request offers clarity about what changes were introduced and why, aiding in both current understanding and future maintenance.

The strategic use of merge request comments opens up opportunities for automating aspects of project management, such as generating release changelogs. Tools like Release Please (<https://github.com/googleapis/release-please>), Auto Changelog (<https://github.com/cookpete/auto-changelog>), and the GitHub Changelog Generator (<https://github.com/github-changelog-generator/github-changelog-generator>) can be employed to create comprehensive and informative changelogs. These tools operate by extracting information from commit messages and merge request comments, underlining the importance of consistency and clarity in these communications.

To maximize the effectiveness of these tools, adopting a standardized approach to commit messages is recommended. The Conventional Commits specification offers a lightweight, yet structured convention for crafting commit messages. This specification aligns with Semantic Versioning (SemVer), categorizing changes as features, fixes, or breaking changes. A typical commit message following this convention might look like: `feat(lang): add Polish language``. This format not only simplifies the process of generating automated tools like changelogs but also ensures a clear and explicit commit history, making it easier

for team members to navigate and understand the evolution of the project.

For more detailed guidelines on Conventional Commits, developers and teams can refer to the official documentation at Conventional Commits (<https://www.conventionalcommits.org/en/v1.0.0/>). By adhering to these guidelines, teams can significantly enhance the traceability and manageability of their code contributions, streamlining both the development process and long-term maintenance of the software project.

## **Code Reviews**

Code reviews are a critical part of the contribution process. They provide an opportunity for team members to collaborate, share knowledge, and ensure code quality. Tools like GitHub, GitLab, or Bitbucket facilitate code reviews by integrating pull requests (PRs) or merge requests (MRs), where changes can be discussed, reviewed, and eventually merged into the main branch. Effective code reviews not only catch bugs but also foster a culture of learning and mentorship.

## **At a Monorepo**

Within a Monorepo environment, code contribution requires careful management to maintain the stability and consistency of the codebase. Bazel ensures that changes in one part of the Monorepo do not inadvertently break other parts.

Code contribution in the context of a monorepository requires developers to be more aware of the broader impact of their changes. They must understand the dependencies within the repo and ensure that their changes are compatible with the entire system. Bazel's ability to manage dependencies and build targets efficiently makes it an essential tool in this environment.

## **CI/CD Pipelines**



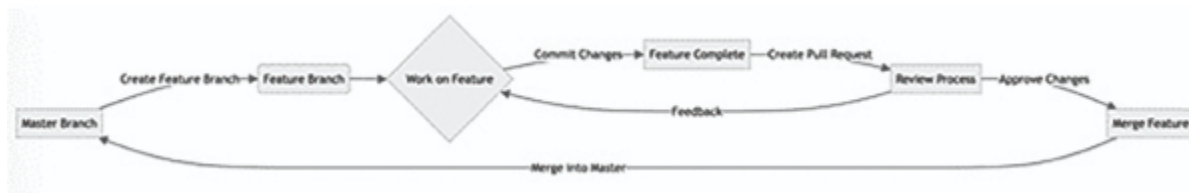
With Bazel, the CI pipeline can be optimized to only rebuild and test affected targets, saving time and resources. Continuous integration ensures that code contributions are automatically tested, making it easier to maintain a high-quality codebase. Continuous deployment, on the other hand, allows for automated deployment of code changes to production, ensuring that new features and fixes are delivered quickly to users.

Modern workflows heavily rely on automation to streamline processes. Tools for static code analysis, automated testing, and deployment are integrated into the workflow. Bazel complements these tools by providing a consistent build environment. Automation not only speeds up the process but also minimizes human errors, leading to more reliable and maintainable codebases.

Feature branching, GitFlow, and trunk-based development are three distinct approaches to version control and collaboration in software development. Each method has its own merits and ideal use cases.

## Feature Branching

Feature branching involves creating a new branch in the version control system for each new feature or bug fix. This allows developers to work independently on different features without impacting the main codebase. The main advantage of feature branching is that it isolates new development from the main code. This isolation reduces the risk of introducing bugs to the main codebase. Here is a basic outline for such a diagram:



**Figure 7.1:** Feature Branching Workflow

This diagram represents the following steps in the feature branching workflow:

1. **Master Branch (in other words, main or trunk ):**  
The workflow starts with the master branch, which is the main branch in the repository hosting a stable version of code always.
2. **Create Feature Branch:** A new feature branch is created off the master branch for each new feature.
3. **Work on Feature:** Development work is done on the feature branch.
4. **Commit Changes:** Once the feature is complete, the changes are committed to the feature branch.
5. **Create Pull Request:** A pull request (PR) is created to merge the feature branch into the master branch.
6. **Review Process:** The PR goes through a review process where other team members can provide feedback.
7. **Feedback Loop:** If there is feedback, changes may be made on the feature branch and the process repeats from the commit step.
8. **Approve Changes:** Once the changes are approved in the PR.
9. **Merge Feature:** The feature branch is merged into the master branch. The feature is now part of the master branch, completing the workflow.

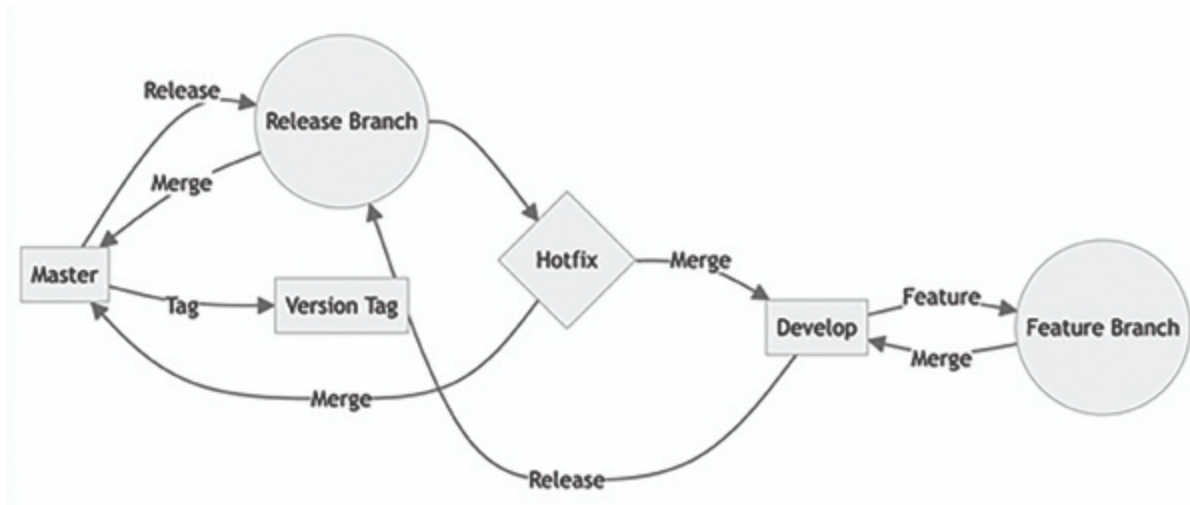
This is a simplified view of the feature branching workflow, but it covers the essential steps involved in this process.

Nevertheless, this approach can lead to integration challenges when merging these branches back into the main branch, especially if the branches have diverged significantly over time.

## GitFlow

GitFlow is a specific branching model for Git. It defines a strict branching structure centered around two main branches: master for production releases and develop for the next release development. Feature branches, release branches, and hotfix branches branch off and merge back into these two main branches.

The following diagram represents a typical GitFlow workflow, showing the relationship and interactions between different branches used in this model.



**Figure 7.2:** Gitflow Workflow

In this diagram:

- **Master:** The main branch where the source code always reflects a production-ready state.
- **Develop:** The branch where all the development happens. It contains the latest delivered development changes for the next release.
- **Feature Branch:** Branches off from 'Develop'. These are used to develop new features.
- **Release Branch:** Branches off from 'Develop' when the team decides to freeze new features for the upcoming

release. It is used for last-minute dotting of i's and crossing of t's.

- **Hotfix:** Branches off from 'Master'. It is used to quickly patch production releases.
- **Version Tag:** Marks a specific point in the 'Master' branch's history that is considered an important version (for example, v1.0, v2.0).

GitFlow provides a robust framework for managing larger, more complex projects and is excellent for teams that need a clear structure for release management. Conversely, it can be overly complex for smaller projects or teams, leading to unnecessary overhead.

## Trunk-based Development

Trunk-based development, on the other hand, encourages developers to commit changes directly to a single branch, often called the **trunk** or **main** branch. Short-lived feature branches may be used but are merged back into the trunk frequently, usually within a day or two. This method emphasizes continuous integration and minimizes the divergence between branches.

This flow emphasizes continuous integration and frequent merging of feature branches into the trunk, minimizing the divergence between branches and ensuring a more stable and up-to-date main codebase.



**Figure 7.3:** Trunk-based Development Workflow

In this diagram:

1. **Developer Starts Work:** A developer begins working on a new feature or bug fix.

2. **Create Short-Lived Feature Branch:** The developer creates a new branch from the trunk or main branch. This branch is intended to be short-lived.
3. **Work on Feature:** The developer works on the feature, committing changes regularly.
4. **Push to Remote Feature Branch:** Once a set of changes is ready, the developer pushes them to the remote repository.
5. **Code Review:** The changes undergo a code review process, where other team members review the code.
6. **If Changes Requested:** If the review process identifies necessary changes, the developer goes back to working on the feature.
7. **If Approved:** If the code review is successful, the feature branch is merged into the trunk or main branch.
8. **Continuous Integration Testing:** Automated tests run to ensure that the new changes integrate well with the existing codebase.
9. **Test Pass:** If the tests pass, the feature is ready for deployment. If not, the developer needs to address the issues.
10. **Deploy to Production:** Upon successful testing, the feature is deployed to production.
11. **Feature Live in Production:** The feature is now live and accessible to users.

The primary benefits of trunk-based development include:

- **Improved Integration:** Continuous integration of changes reduces the integration challenges commonly found in feature branching. This leads to fewer merge conflicts and integration issues.
- **Faster Feedback Cycle:** Developers get immediate feedback on their changes as they are integrated into

the main codebase quickly. This allows for early detection and resolution of issues.

- **Enhanced Collaboration:** With everyone working on the same codebase, there is more collaboration and knowledge sharing among team members. This can lead to a more cohesive and robust product.
- **Simplified Process:** Trunk-based development simplifies the development process by eliminating the need for complex branching strategies, making it easier for new team members to understand and contribute to the project.
- **Continuous Delivery:** This method aligns well with continuous delivery and deployment practices, as changes are always ready to be deployed to production, enhancing the overall agility of the team.

In summary, while feature branching and GitFlow have their places in certain project contexts, trunk-based development offers significant advantages in terms of integration, feedback, collaboration, simplicity, and alignment with continuous delivery practices, making it a compelling choice for many software development teams.

## [Setting Up Continuous Integration with Bazel](#)

Setting up continuous integration (CI) operating within a Monorepo environment, especially one dealing with modern container-based architectures, requires a thoughtful and systematic approach. In this context, the primary role of Bazel, a powerful build tool, becomes crucial in producing binaries like Docker images and updating continuous deployment (CD) infrastructure as code definitions for various products and environments hosted within the Monorepo.

## Container Image Management in Monorepos

When working with a monorepository structure, you often manage numerous products, each potentially requiring multiple container images for different layers, domains, and services. Each docker container image comprises a name and a label, where the name identifies the content of the image, and the label signifies its version. Bazel's task is to generate one or more containers based on the code contributions, which could vary depending on the specific trigger within the CI process, such as a commit in a feature branch, the main branch, or a release branch.

### Labeling Strategy for Docker Images

A crucial aspect of this set up is implementing a labeling strategy that reflects the stage of code contribution and ensures uniqueness and traceability of each container. Labels typically include a prefix indicating the stage of code contribution and the commit hash code. For example:

- A frontend container for 'productX' might be labeled `productX-frontend:feat-JIRA1234-b1c7bc91`, indicating it originated from a feature branch addressing the JIRA1234 ticket, with a specific git hash.
- A backend container from the main branch may be labeled `productX-backend:dev-b1c7bc91`, reflecting its development stage and associated git hash.
- For release branches, a label like `productX-backend:release-241130-b1c7bc91` would indicate the release date and the corresponding commit.

This naming convention, following a namespace approach, enhances organization and clarity, especially in the framework of a monorepository system hosting multiple products and services.

### Integration with Infrastructure as Code

Beyond container creation, Bazel's role extends to updating the infrastructure as code (IaC) components, such as Helm charts or Terraform configurations, for the affected products and environments. This IaC could be implemented by using plain and easy templating systems. This update is vital to ensure that the infrastructure aligns with the latest container deployments.

## Triggering Continuous Deployment Pipelines or Jobs

The CI pipeline's integration with CD is another critical element. Based on the updated code and container images, the CI process can trigger CD pipelines for the respective products and environments. This automation ensures that any changes, whether they are new features, bug fixes, or updates, are promptly and efficiently deployed across the necessary environments.

## CI Pipeline Jobs

The flexibility of CI pipelines allows them to be tailored according to the specific requirements of a product. This section outlines a typical pipeline, explaining each job's role and how they can evolve over time. Keep in mind that yours could be similar or totally different to this one.



**Figure 7.4:** Prototypical CI Pipeline

From the diagram, each job has these responsibilities:

- **Dependency Installation:** Ensures all necessary dependencies are available for the build process. Initially, this might involve just fetching libraries, but over time it may include more complex tasks like conditional dependency management based on different build environments.



- **Secrets Gathering:** Securely obtains necessary credentials and secrets required for the build process. Starts with basic secret retrieval, and can evolve to include dynamic secret generation or integration with advanced secret management systems.
- **Build:** Compiles the source code into executable binaries or libraries. Initial setups might focus on a simple build process, which later evolves to incorporate incremental builds, multi-platform support, and optimization flags.
- **Test:** Runs automated tests to validate the correctness of the code. Early stages involve basic unit tests, expanding over time to include integration, performance, and stress tests, with improved test reporting and flakiness detection.
- **Package Components:** Bundles the built software into distributable formats. May start with simple packaging and evolve to support multiple package formats, conditional packaging based on features, and digital signing.
- **Coverage:** Measures how much of the code is covered by tests. Begins with basic coverage metrics, progressing to more detailed coverage analysis, including branch, path, and integration test coverage.
- **Code QA Checks:** Ensures code quality through static analysis, linting, and code style checks. Initially focuses on basic style checks, growing to include more comprehensive analysis, custom rules, and automatic code formatting.
- **Publish Components:** Distributes the packaged software to repositories or storage systems. It can evolve from simple publication processes to include version management, multi-environment releases, and rollback capabilities.

- **Trigger Deployment:** Initiates the deployment of the software into a staging or production environment. It might start with manual triggers, evolving to automated, environment-specific deployments with canary releases and feature flagging.

Each job in the CI pipeline plays a critical role in the software development lifecycle, and their definitions are subject to change as the project grows and the team gains more experience. It is important for teams to regularly review and refine their CI processes to ensure they align with the evolving needs of the project and the organization.

### **Additional Recommendations from Learned Lessons**

Here is a detailed look at how to effectively implement CI with Bazel:

- **Leverage Bazel's Build Caching:** One of Bazel's key features is its build caching mechanism. By configuring your CI system to cache Bazel's output directories, you can significantly reduce build times for subsequent runs. This caching ensures that only the parts of your project that have changed will be rebuilt, saving time and computational resources.
- **Utilize Incremental Builds:** Bazel excels at performing incremental builds, where only targets dependent on changed files are rebuilt. To maximize the effectiveness of this feature, it is important to manage the granularity of your build targets. The goal is to strike the right balance between reusability and minimizing unnecessary rebuilds, which contributes to faster build processes and more efficient resource usage.
- **Parallel Test Execution:** Bazel has the capability to run tests in parallel, which can drastically speed up test execution. To fully benefit from this feature, ensure that your tests are designed to be run concurrently. This not

only improves the speed of your testing phase but also contributes to a more robust testing environment.

- **Manage Dependencies Carefully:** With Bazel's strict dependency checking, it is crucial to keep a well-organized and up-to-date dependency graph. This vigilance helps catch issues early and ensures that your builds are as efficient as possible.
- **Implement Sandboxed Environments:** Bazel's sandboxed environments are pivotal for ensuring reproducible builds. These environments help catch environment-specific bugs early in the development cycle, reducing the risk of "*works on my machine*" problems and enhancing the reliability of your builds.
- **Integrate with Version Control:** Seamless integration between your CI system and version control (like Git) is vital. This integration allows for efficient tracking of changes and more effective management of builds, ensuring that each change is accounted for and built accordingly.
- **Focus on Documentation and Training:** A well-informed team is crucial for the effectiveness of your CI system. Ensure that your team members are properly trained in both the CI process and Bazel. Comprehensive documentation and regular training sessions can greatly improve the efficiency and adoption of the CI system across your team.
- **Preventing Accidental Discards in Bazel's Analysis Cache:** Large repositories lead Bazel to spend significant time on the "**Analysis Phase**". The results, stored in Bazel's in-memory server cache, can be accidentally discarded by running a Bazel command with different flags. Performance degradation often goes unnoticed, indicated by messages like "**options have changed, discarding analysis cache**". Commonly, changes

in CI scripts can trigger this issue. To avoid this, a layer in the CI design should wrap Bazel calls, ensuring consistent flag usage. Relying solely on ``.bazelrc`` can be insufficient, and certain Bazel bugs, like those in ``bazel coverage``, can also cause cache discards.

- **Persistent Runners:** With the rise of Kubernetes, ephemeral CI instances have become common, benefiting most build systems by isolating builds. Nonetheless, Bazel, with its built-in correctness guarantee, suffers in such setups. Initial attempts to use CI system caching, like ``--repository_cache``, partially mitigate issues but do not eliminate the need for Bazel to re-execute repository rules. Optimally, CI runners should be responsive and scalable, balancing cost and peak load demands. Various solutions like Buildkite, CircleCI, and GitHub Actions offer scalable runner options.
- **Warm Persistent Runners:** Persistent runners face challenges, especially during the day's ramp-up period, resembling the inefficiency of fresh machines. The Bazel server and output tree are affected by the last workload, causing cache and output invalidations when syncing through commit history. Time and strategy are required to enhance performance for the slowest 95th percentile builds.
- **Runner Health Checking:** Shifting from ephemeral to persistent runners introduces the risk of resource leaks, like unreleased docker containers. Implementing health checks and cleanup mechanisms is crucial to manage poorly behaved workloads and maintain runner efficiency.
- **Choosing and Deploying a Remote Cache:** Selecting a remote cache involves understanding various trade-offs, like network bandwidth limitations and the need for replication. It is essential to prioritize addressing non-

determinism in build inputs to ensure high cache hit rates.

- **Mirroring Internet Files:** External service dependencies can impact uptime. Setting up Bazel's downloader with `--experimental_downloader_config` for a read-through mirror is advisable. Restricting new internet dependencies and firewalling agents during build and test phases can enhance security and stability.`
- **Defining SLAs and Monitoring:** Providing metrics on CI performance, like queue wait times and notification speed for failing tests, is crucial. Monitoring external repository and analysis cache invalidation rates aids in diagnosing and addressing performance issues.
- **Maintaining a Green Build:** Defining "green" status and implementing quick fixes for red master branches are vital. Strategies include identifying breakages, alerting responsible parties, and conducting post-mortems to prevent future issues.
- **Remote Execution:** In large-scale organizations, Remote Build Execution (RBE) allows Bazel builds to be executed in a distributed fashion across remote servers, significantly speeding up the build process by parallelizing the work. RBE becomes relevant for efficiently handling extensive, parallelizable workloads. Options include SaaS offerings and systems like BuildBarn that integrate RBE with remote caching.
- **Ensuring Continuous Delivery and Deployments:** CI pipelines should handle release artifact creation, avoiding local machine builds for reproducibility. Balancing security and performance requires a separate pipeline for building release artifacts.
- **Remote Build Execution with Simple CI Runners:** Implementing RBE requires understanding Bazel's

functioning, particularly the work needed before remote execution. Without careful setup, RBE might increase costs without fully leveraging Bazel's incremental model.

- **Enforce Security Practices:** Last but not least, implement and strictly enforce security practices in your CI pipeline. This includes managing access controls, securely storing secrets, and regularly scanning for vulnerabilities to protect your codebase and CI environment from security threats.

In summary, setting up CI with Bazel, particularly with the Monorepo approach handling container-based deployments, requires a multifaceted approach. This includes leveraging Bazel's features like caching and incremental builds, efficiently managing container images, adopting strategic naming and labeling conventions, and integrating seamlessly with infrastructure as code (IaC) tools and continuous deployment (CD) pipelines. By maintaining a well-managed and secure development environment and ensuring your team is adept at utilizing these systems, this comprehensive strategy not only streamlines the development process but also guarantees the efficient, traceable, and reliable delivery of high-quality, reliable software in complex, multi-product environments.

## **Enabling a Sort of Local CI with Bazel**

Local Continuous Integration (CI) is a powerful concept in software development, where changes made by developers are continuously integrated and tested within their development environment. This strategy offers immediate feedback on code changes, enhancing productivity and code quality. Bazel, a build and test tool, plays a crucial role in enabling this local CI process, especially when paired with a tool like Bazel Watcher.

Frontend developers are familiar with the benefits of local hot reload development environments, where changes are instantly compiled and executed as soon as they are saved to disk. This immediate feedback loop significantly speeds up development. Now, with Bazel and a tool like Bazel Watcher, developers across all disciplines can experience similar benefits.

## **What is a Bazel Watcher?**

Bazel Watcher is an innovative tool that extends the functionality of Bazel to create a more dynamic and responsive development environment. It monitors the file system for changes in source files and automatically triggers Bazel builds and tests in response. This means that every time a developer saves a change, Bazel Watcher ensures those changes are immediately reflected in the build and test outputs, mimicking a CI pipeline but on a local scale.

## **How Bazel Watcher Works**

Bazel Watcher works by watching for file changes in the source tree. When a change is detected, it intelligently triggers Bazel to rebuild or retest only the affected parts of the codebase. This selective building and testing make it highly efficient, as it does not require rebuilding the entire project for every small change. The result is a rapid feedback loop, allowing developers to quickly identify and fix issues as they code.

## **Installing Bazel Watcher**

Bazel Watcher can be installed via package managers like npm for Node.js projects, from Homebrew in MacOS (`brew install ibazel`) or directly from its source repository. The installation process typically involves running a simple command in the terminal, such as `npm install bazel-watcher` or a similar command based on your project's ecosystem.

More info at <https://github.com/bazelbuild/bazel-watcher#installation>.

## Examples Using Bazel Watcher

Here are some example scenarios demonstrating how Bazel Watcher can be used:

- **Instant Feedback for Code Changes:** As a developer works on a new feature or bug fix, they can rely on Bazel Watcher to instantly compile and test their changes, providing immediate feedback.
- **Streamlining Frontend Development:** Frontend developers, often accustomed to hot-reloading tools, can use Bazel Watcher to instantly see the impact of their styling or UI changes.
- **Optimizing Backend Development:** Backend developers can use Bazel Watcher to ensure that any changes to the codebase do not break existing functionality, with tests running automatically upon each save.
- **Integrating with Large Codebases:** In large projects, Bazel Watcher can be particularly useful, as it selectively tests and builds only affected areas, saving time and resources.

In summary, Bazel Watcher enhances the development workflow by bringing the principles of CI into the local development environment. It not only streamlines the process of building and testing but also aligns closely with the practices of modern, agile software development. By providing instant feedback on changes, it helps developers to quickly iterate and improve their code, leading to higher quality software and more efficient development cycles.

## [CI Worker Set up Models](#)



The configuration and management of CI workers are critical aspects that significantly impact the efficiency and scalability of your build process. The setup of CI workers involves navigating a balance between two key sets of trade-offs— isolation versus reusability, and operational complexity versus scalability. Each approach to CI worker setup offers a unique combination of these factors, and understanding these nuances is crucial for optimizing your CI pipeline.

In this section, we will delve into various models of CI worker setups, each tailored to address specific needs and challenges in the CI process. Many of these models were already defined by Son Luong Ngoc in his post “*Bazel in CI (Part 2): Worker Set up*” at <https://sluongng.hashnode.dev/bazel-in-ci-part-2-worker-setup>. Here is a brief overview of the models that we will explore:

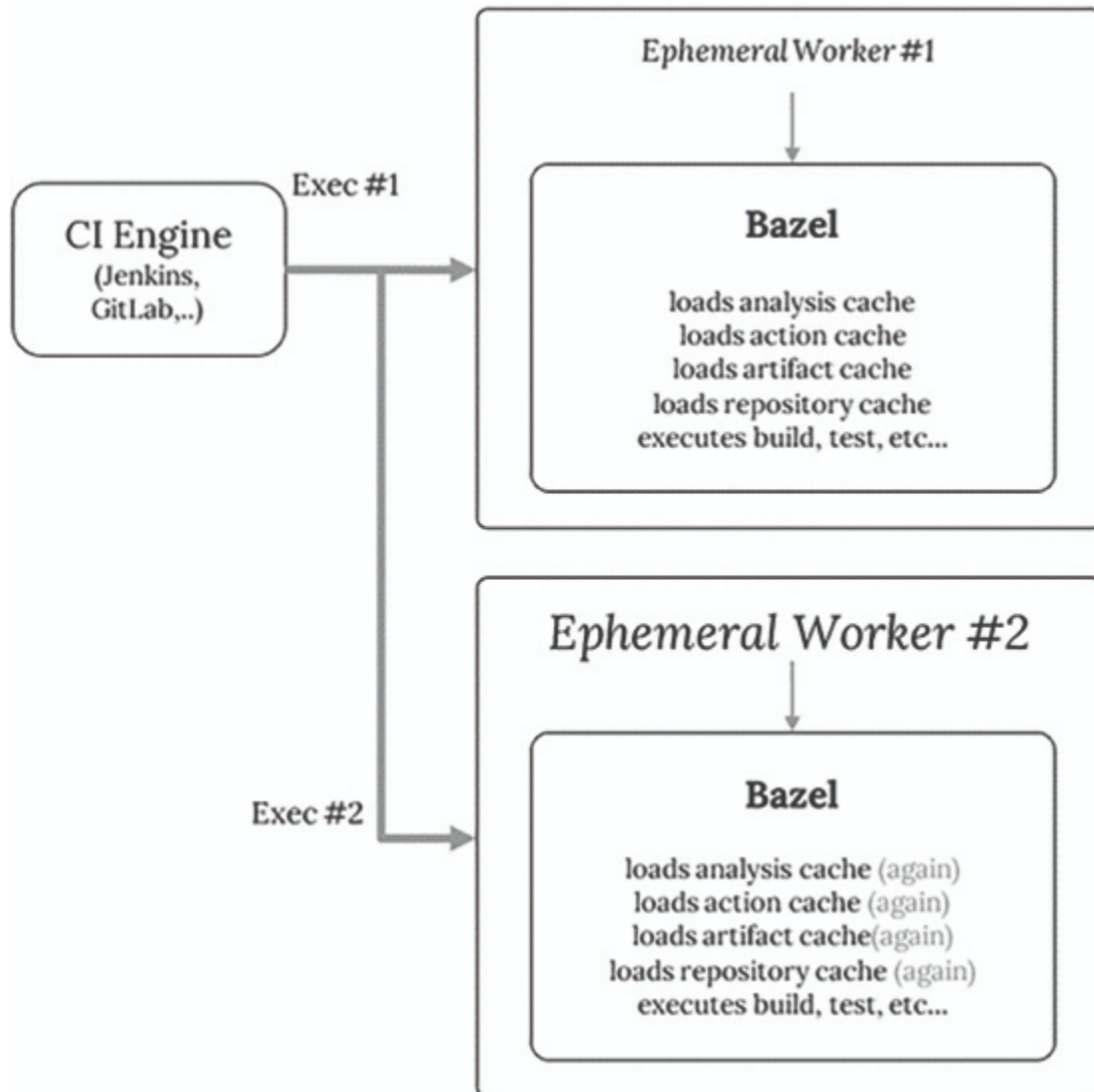
- **Ephemeral Workers:** These workers are temporary and are created anew for each build, offering high isolation but often at the cost of increased set up time and resource utilization.
- **Single Stateful Worker:** This model employs a persistent worker that maintains its state over time, enhancing reusability and reducing set up time, but potentially compromising isolation.
- **Multiple Stateful Workers:** An extension of the single stateful worker model, this mode of operation utilizes several persistent workers, improving scalability and parallel processing capabilities.
- **Hot-Pool of workers:** A dynamic model where workers are provisioned on-demand based on predefined configurations, balancing scalability with controlled environment set up.

- **Sharded Worker Sets:** This model involves dividing workers into specialized groups or 'shards,' each optimized for specific tasks or build environments, thereby enhancing efficiency and reducing build times.
- **Remote Build Execution:** A sophisticated model where build tasks are offloaded to remote servers or cloud environments, significantly improving scalability and potentially reducing local resource constraints.

It is important to note that adopting these models does not require a rigid sequence. Instead, the choice should be based on the specific requirements of your Monorepo and the dynamics of your team. Each model addresses distinct challenges and can be implemented incrementally. This flexibility allows you to tailor your CI setup to progressively resolve the unique build and operational challenges you encounter.

## **Ephemeral Workers**

Initially, many organizations tend to opt for ephemeral CI Workers due to their simplicity in setup.



**Figure 7.5:** Diagram Illustrating the Concept of Ephemeral CI Workers

Ephemeral CI setups are commonly supported by most CI service providers, often allowing execution of shell commands within a Docker Container or VM. In such setups, it is typical to include a command like ``bazel test //...`` to initiate a Bazel build.

On the other hand, the transient nature of ephemeral workers means that the Bazel JVM server must be recreated for each build, resulting in the loss of Bazel's valuable in-memory cache.

Moreover, these temporary environments also lead to the loss of local persistent cache, necessitating that Bazel builds start anew each time.

For more experienced users, a workaround involves directing `--disk_cache` to a reusable container volume mount. This tactic can help minimize action executions by caching the Content Addressable Storage (CAS) and Action Cache (AC) locally on the disk.

Similarly, configuring `--repository_cache` to use a reusable volume can reduce the need to download external dependencies repeatedly. We have discussed the Bazel repository cache in more detail in another section.

Despite these optimizations, builds might still experience slowness due to the necessity of running the analysis phase from the beginning in each build. This is because many repository rule actions must still be executed, even with the downloaded archives in the repository cache.

## **Single Stateful Worker**

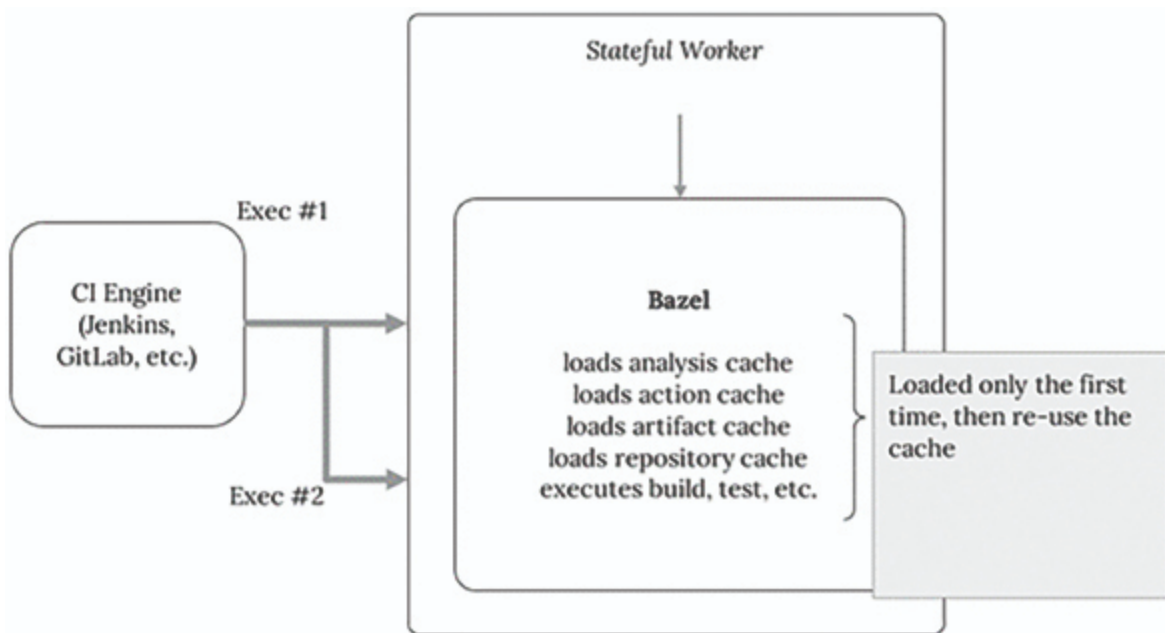
As your Monorepo expands, the limitations of an ephemeral worker CI setup become apparent.

With the growth of the Monorepo, the analysis phase in builds and tests becomes increasingly sluggish. This is due to the expansion of the build graph with more actions, artifacts, nodes, and edges, all of which require time to traverse.

To mitigate this in ephemeral workers, we previously focused on reducing their ephemeral nature by implementing isolated local caching via disk cache and repository cache.

To further enhance the efficiency of Bazel's Analysis Phase, it is beneficial to utilize Bazel's in-memory cache.

Maintaining the same Bazel JVM across CI jobs can significantly reduce the time spent in the analysis phase.



**Figure 7.6:** Illustration of Leveraging Bazel's In-memory Cache in CI Jobs

An effective strategy here is to activate `--watchfs`, allowing the Bazel JVM to track project changes using `fsnotify` rather than repeatedly executing `stat()` and hashing all project files.

Still, transitioning to stateful workers necessitates some additional considerations:

- **CI Sandbox Technology:** The shift away from containers means maintaining the JVM process across jobs. This requires using the same container or VM, making a shell session the new unit of isolation.
- **Reduced Isolation:** This technique significantly diminishes the isolation of CI environments, introducing new risk factors like disk space shortages, lingering Git lock files, and the use of dirty files in builds.
- **Additional Measures:** To address these issues, a wrapper for Bazel execution is needed to manage

potential problems before and after each build, adding complexity to the CI setup.

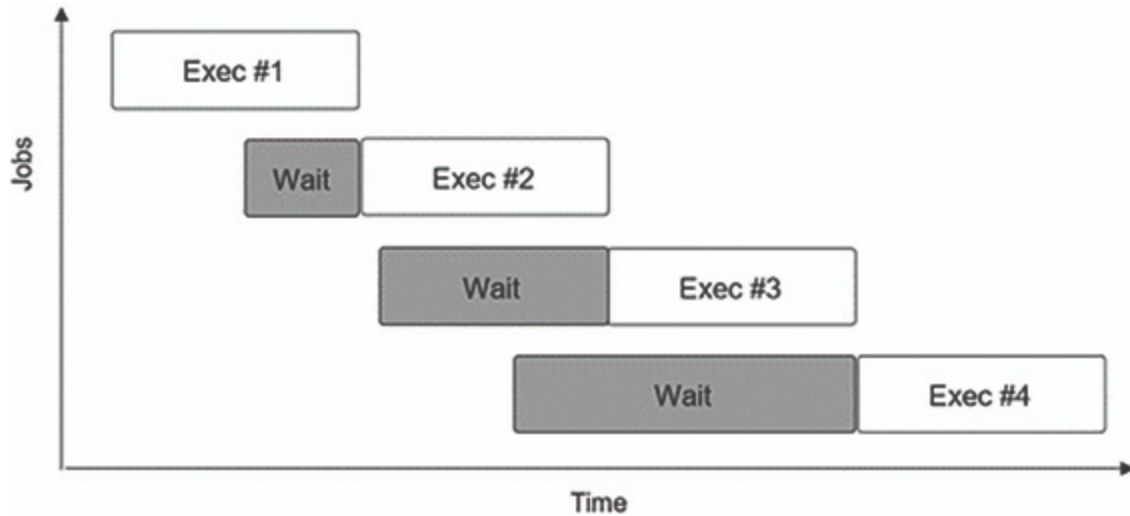
Using a single stateful worker involves a process for seeding fresh workers with a pre-warmed local cache. This might involve a new worker provisioning process, like using terraform for VM instance creation, with embedded cache seeding logic. An example could be performing a Bazel build from the latest default branch to warm up the cache.

Contrary to what might be expected, using a stateful worker can actually reduce infrastructure costs. Faster builds and tests allow for scaling down the total compute resources needed. This not only lowers infrastructure expenses but also enhances overall engineering productivity through quicker feedback cycles.

Although, the increased complexity of the CI setup does raise the total cost of ownership. This trade-off is becoming more prevalent — by accepting higher complexity and investing in the right solutions, we can reduce infrastructure costs and boost productivity across engineering teams.

## **Multiple Stateful Workers**

As your team expands and the rate of changes escalates, relying on a single stateful worker becomes insufficient. Indicators such as the median time builds spend in queue (P50 of build-in-queue) and queue duration are crucial metrics signaling this shift.



**Figure 7.7:** Visual Representation of the Need for Multiple Stateful Workers

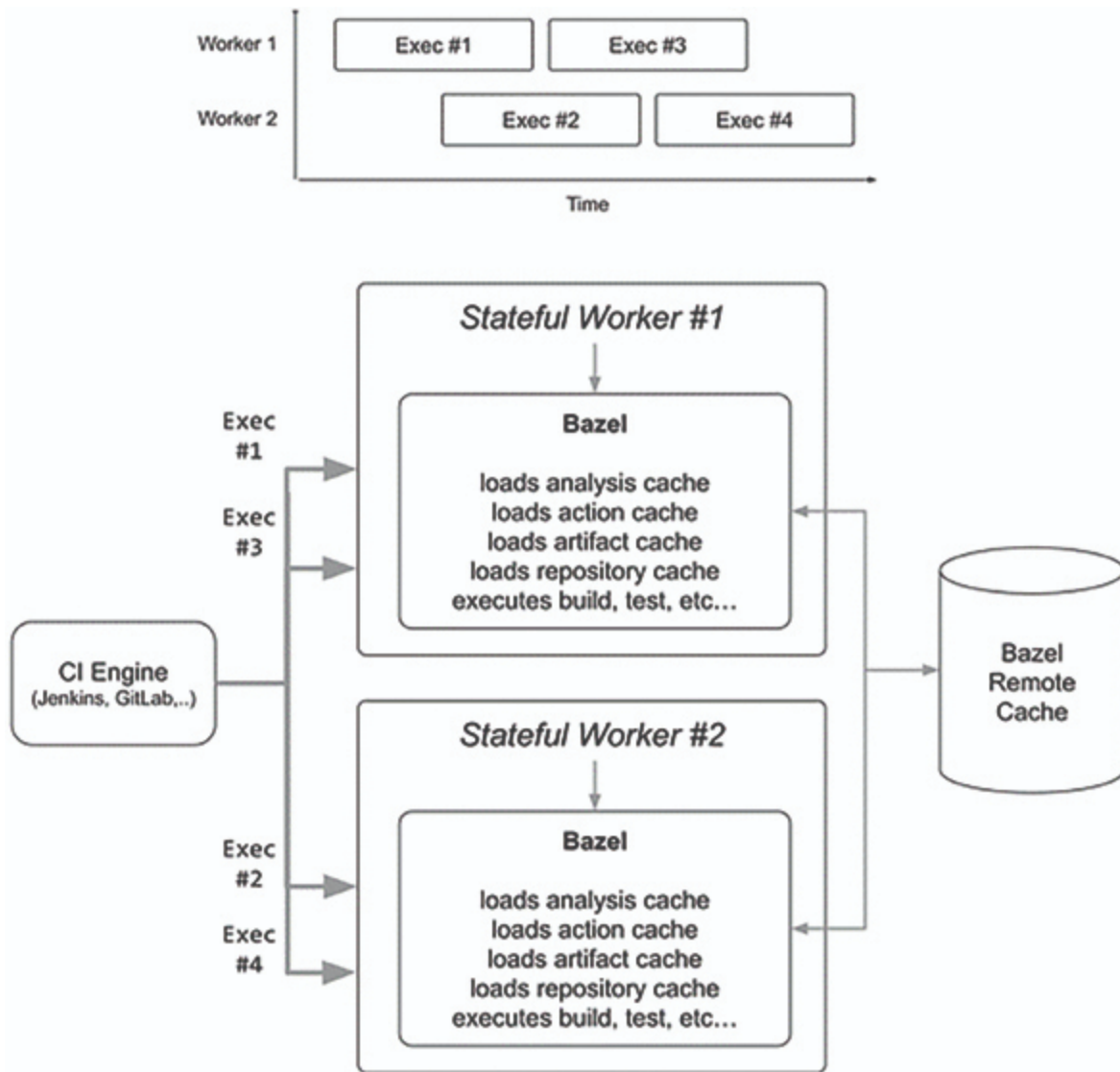
This scenario necessitates the deployment of multiple stateful workers.

Even so, transitioning to a setup with multiple stateful workers introduces a set of new challenges, particularly those related to the CAP theorem:

- **Distribution of Workload:** With more workers, the workload is distributed, reducing overall queue time and increasing availability for new jobs. Yet, this leads to inconsistencies in local cache content across workers. The greater the number of workers, the more pronounced these inconsistencies become.
- **Potential for Increased Build/Test Time:** As a result of cache inconsistencies, tests and builds may need to be rerun on different workers, potentially leading to an overall increase in build and test times.

To tackle the consistency issue, integrating `--remote_cache`` is advisable. This allows for centralizing all Content Addressable Storage (CAS) and Action Cache (AC) results in a shared storage accessible to all workers. Workers can then reference existing action entries in the action cache to avoid

redundant compiling and testing when the same job is rerun on a different worker.



**Figure 7.8:** Illustration of Leveraging a Remote Cache in a Multi-worker Environment

Additionally, implementing merged result CI builds can enhance cache consistency. This version control strategy aligns local caches across different workers with the project's default branch. As CI jobs use a common base, workers can utilize the CAS and AC from this base to maintain a consistent local cache over time.



A common mistake with multiple stateful workers is over-provisioning. An excess of workers can lead to a more sparse distribution of the local cache, decreasing local cache hit rates. Moreover, relying heavily on a centralized remote cache can be costly at scale.

To address this, it is recommended to maintain just enough workers to keep CI Job queue times under 30 seconds. Regular maintenance tasks should also be implemented to remove inactive workers during downtimes and replace them with fresh workers pre-seeded with the latest cache, optimizing overall efficiency.

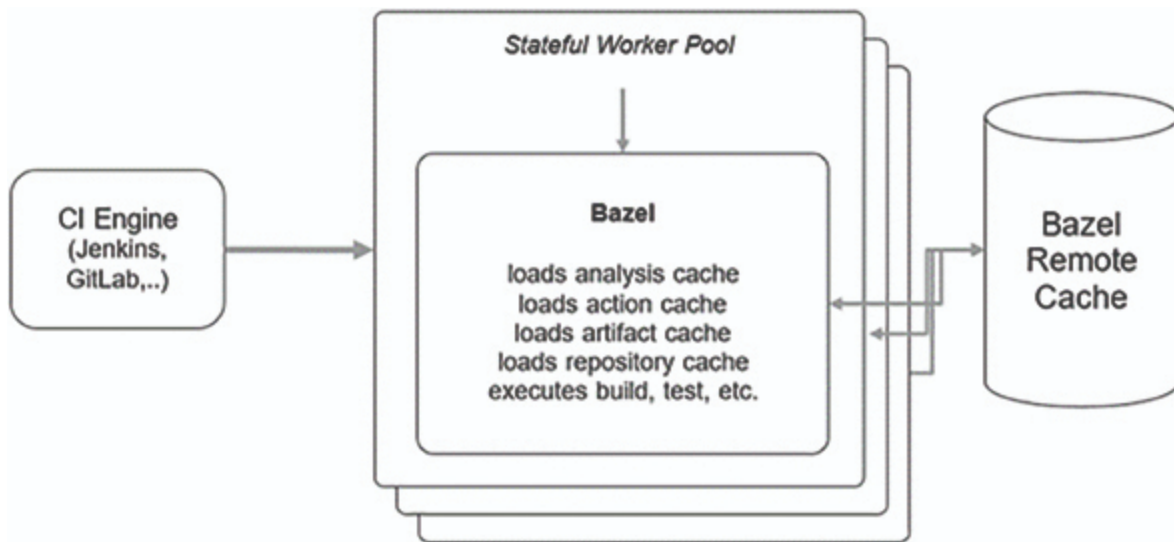
## **Hot-pool of Workers**

As your Monorepo expands, CI isolation issues are likely to become more prevalent. These can range from residual git networking failures causing lock file issues, potentially failing builds, to local cache contamination where one build adversely impacts subsequent ones. By monitoring and categorizing various CI failure types, you can gauge the value of a sanitized testing environment for your specific needs.

Another common challenge is scaling the throughput of your Bazel Remote Cache solution. With multiple workers, the local cache can become highly fragmented, leading to inconsistencies in the content downloaded from the Bazel Remote Cache. If your remote cache utilizes a multi-tier storage system (in-memory, local disk, remote object storage), these irregular downloads can disrupt the efficient management of cache entries across different storage tiers. This can result in increased network throughput and more frequent cache tier changes, slowing down the overall remote cache download speeds.

To address these issues, a more uniform operation of workers across different CI Jobs is desirable. A well-

established approach for this is the CI Worker Hot-pool of Workers model.



**Figure 7.9:** Diagram Illustrating the CI Worker Hot-pool of Workers Concept

The Hot-pool of Workers model operates as a service managing a ready pool of workers, poised to handle incoming CI Jobs. Post job completion, workers are either recycled through the Hot-pool of Workers's cleanup process or discarded, with the service preparing new ready-to-deploy workers. The choice between recycling and discarding workers often hinges on the ease of provisioning and warming up a new worker, which varies depending on whether you are using on-prem or cloud infrastructure, or different VM solutions with their own startup speeds.

Idle workers in the pool are assigned a Time To Live (TTL). Workers that remain unused past their TTL are considered cold and are subjected to the same recycling process as used workers.

This setup aims to achieve a higher degree of ephemerality for each worker serving a new CI Job, while still retaining the advantages of hot workers pre-loaded with the necessary cache.

Custom auto-scaling logic can be applied to the Hot-pool of Workers service to optimize computing costs. Emerging technologies like FireCracker MicroVMs are particularly well-suited for this model, thanks to their lower startup times and the capability to resume a VM with an intact Bazel In-Memory Cache.

## Sharded Worker Sets

As the complexity of your build graph increases, you will likely notice a corresponding growth in the Bazel Analysis Cache, leading to increased memory demands on the Bazel JVM. At this level of scale, it is common to see the Bazel JVM being terminated due to out-of-memory (OOM) issues during the loading-and-analysis phase.

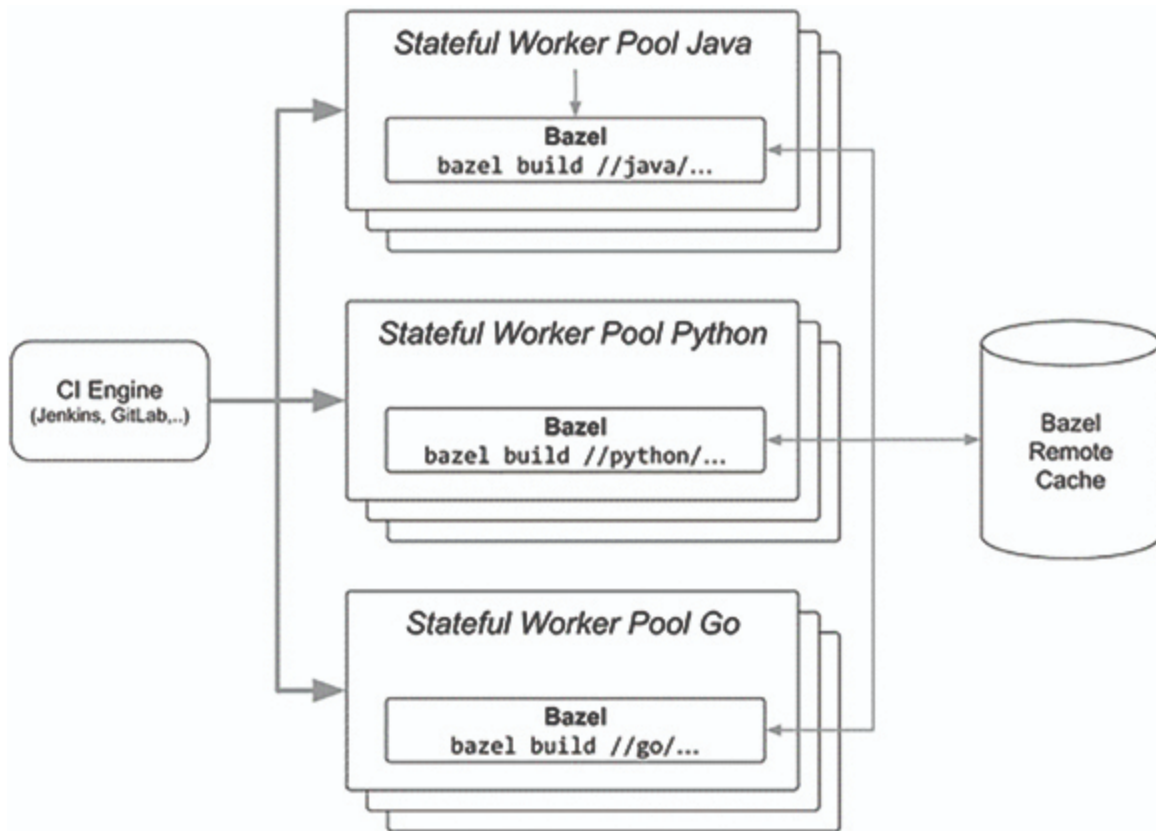
Currently, there is no universally accepted solution to this issue. Ideally, Bazel JVM would have the capability to offload its analysis cache remotely, either by serializing cache entries to a disk-based file format or storing them in a scalable remote cache. That said, this issue is not widespread, as few projects reach a scale where it becomes a significant problem. Most projects can still manage by simply increasing the resources allocated to their Bazel workers.

For those rare projects that do encounter this scale, or for those who find vertical scaling of CI workers challenging due to constraints like time-consuming hardware upgrades, Sharded Worker Sets offer a potential workaround.

The concept of Sharded Worker Sets is straightforward—instead of executing a single large build, such as ``bazel test -- //...``, the build process is divided into smaller, more manageable builds. For example:

- ``bazel test -- //service-a/...``
- ``bazel test -- //... -//service-a/...``

Visually, this approach involves splitting the “*build everything*” setup into multiple builds with smaller target patterns, thereby reducing the size of the analysis cache. Practically, this would involve:



**Figure 7.10:** Illustration of Sharded Worker Sets in CI

Establishing separate CI worker pools for each unique set of build targets to maintain the Analysis Cache effectively. This results in:

- A dedicated pool by language ( `//python/...`, `//java/...`, `//go/...`, and so on)
- A dedicated pool by product ( `//apps/productA/...`, `//apps/productB/...`, and so on)

The benefit of this profile-based sharding is that it can be implemented incrementally. Since most of the complexity associated with managing new sets of CI workers would

have been addressed at this stage, adding additional pools should not significantly burden the Build or DevX teams responsible for their management.

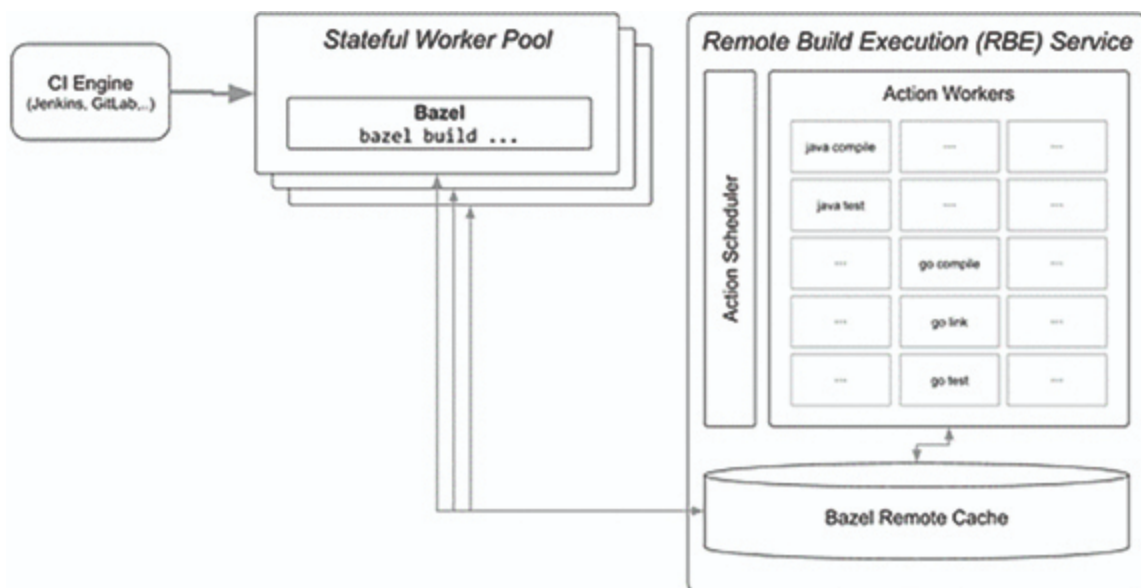
## Remote Build Execution

Sharding your build process means that build and test actions are now distributed across multiple workers, leading to increased execution parallelism due to the higher computing capacity.

Despite this, this setup often results in action duplication. Firstly, if an action is not already present in the remote action cache, two workers might end up executing the same action, especially if they schedule it closely together.

Secondly, if there are shared dependencies between different target sets, such as a common `\\proto/...` dependency for both `\\go/...` and `\\java/...`, the actions necessary to build these shared dependencies might be executed redundantly on different CI workers.

A potential solution to improve action scheduling and reduce duplication is the implementation of a Remote Build Execution (RBE) system.



**Figure 7.11:** *Illustration of Remote Build Execution in Action*

RBE systems typically include features for deduplicating action schedules. When multiple CI workers request the RBE Scheduler to execute the same action, it results in just one execution, with the outcome then distributed to all requesting workers. This mode of operation is exemplified in a case study by Stripe (<https://stripe.com/blog/fast-secure-builds-choose-two>), where they successfully implemented a similar system.

In practical terms, RBE is an effective way to scale the execution of your build graph and tests, optimizing computing resources. However, operating an RBE solution is a complex and resource-intensive endeavor. Large-scale RBE setups can significantly increase the cost and operational complexity of your CI infrastructure.

Unlike the Sharded Worker Sets approach, RBE not only increases the computing resources needed but also adds more components that require ongoing management. This complexity makes SaaS-based RBE solutions a good fit for larger organizations, although they may lack the customization and integration capabilities tailored to specific engineering workflows.

## **Conclusions About CI Worker Models**

In the discussion above, we explored various configurations for setting up CI Workers to manage Bazel workloads. Each of these configurations presents its own set of trade-offs:

- Isolation versus Reusability
- Operational Complexity versus Scalability

It is important to recognize that there is no one-size-fits-all sequence for adopting these setups. The most effective approach is to assess and select the option that best aligns with the specific needs of your Monorepo and team. These

enhancements can be implemented gradually, as each is designed to address distinct challenges within your build process. Therefore, it is crucial to avoid rushing into solutions that may not be necessary for your circumstances. Carefully measure, identify the bottlenecks, and apply the most suitable solution for each specific issue.

It is noteworthy that some organizations have successfully made the leap directly from ephemeral CI workers to RBE. This shift has been facilitated by the growing availability of commercial RBE solutions in recent years. These solutions can significantly reduce the total cost of ownership for organizations adopting Bazel. The advantage here is that it eliminates the need for recruiting a specialized engineering team dedicated to managing a custom action worker pool, a process that could traditionally span 6-12 months.

## **Managing Code Quality Tools**

Tools like code formatters, linters, and static code analyzers play a crucial role in achieving quality and maintainability. Integrating these tools with Bazel, can streamline the process and ensure consistency across the codebase.

## **Formatting**

Consistency in code formatting is a vital aspect of software development, especially in large teams or projects. It enhances readability, reduces the likelihood of merge conflicts, and ensures that the codebase maintains a uniform style. While individual developers may have personal preferences for code styles, it is crucial for a team to adopt a standard format. This not only streamlines the development process but also eases the onboarding of new team members.

To achieve this uniformity, the use of code formatters integrated into Integrated Development Environments

(IDEs) is highly recommended. Most modern IDEs for Java, such as IntelliJ IDEA or Eclipse, offer robust formatting tools that can be customized to match the team's coding standards. By configuring these tools at the IDE level, developers can ensure that every piece of code they write adheres to the agreed-upon style guide.

Although, relying solely on IDE configurations might not be foolproof, as developers might use different IDEs or forget to format before committing code. This is where Bazel, combined with Git hooks, comes into play. Bazel can be used as a reinforcement tool to automatically check and enforce code formatting rules. By integrating Bazel with pre-commit hooks in Git, teams can ensure that all committed code conforms to the standard format, thus maintaining consistency across the codebase.

To integrate a code formatter with Bazel, follow these steps:

### **Step 1: Selecting a Code Formatter**

For JavaScript, popular formatter choices include Prettier and ESLint. Prettier is widely known for its robustness and simplicity, focusing solely on formatting. ESLint, on the other hand, offers both formatting and linting capabilities, allowing for more comprehensive code quality checks.

### **Example Configuration: Using Prettier with Bazel**

1. Installation: Add Prettier to your project dependencies by running

```
npm install --save-dev prettier
```

2. Bazel Rule Creation:

- Create a Bazel rule in your BUILD file to run Prettier. This rule can be a simple shell command invoking Prettier on your JavaScript files:

```
sh_binary(  
  name = "format_js",  
  srcs = ["format_js.sh"],
```



```
    data = glob(["**/*.js"]),  
  )
```

- The `format_js.sh` script will contain the Prettier command:

```
#!/bin/bash  
prettier --write $(find . -name "*.js")
```

## **Step 2: Integrating with Git Hooks**

Integrating the Bazel formatting rule with Git hooks ensures that the code is automatically formatted before commits, guaranteeing consistency. In the next section, we will get into the details on how to do this.

## **Step 3: Continuous Integration**

To further reinforce consistency, include the formatting rule in your continuous integration (CI) pipeline. This ensures that all merged code adheres to the same formatting standards.

## **Linting and Static Code Analysis**

Code linting is a critical aspect of maintaining code quality in software development. It involves analyzing source code to detect and fix programming errors, stylistic errors, and suspicious constructs. In Java projects, linting ensures adherence to coding standards and best practices, which is essential for readability, maintainability, and reducing bugs.

### **Integrating Linters in Bazel**

Bazel, a multi-language build tool, offers robust support for integrating linters into the build process. This ensures that code quality checks are an integral part of the development workflow. In Java projects, popular linters like Checkstyle, PMD, and SpotBugs can be integrated with Bazel.

### **Example: Integrating PMD with Bazel**

This example is available at examples that can be found in the [GitHub repository](https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel) at <https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel>, specifically within the `/chapter-7/bazel_linter` directory. This example will guide our discussion and exploration throughout this part of the book. To get started with `rules_lint`, you first need to add the following to your `MODULE.bazel` file:

```
bazel_dep(name = "rules_jvm_external", version = "5.3")
bazel_dep(name = "rules_pmd", version = "0.4.1")
maven = use_extension("@rules_jvm_external//:extensions.bzl",
"maven")
maven.install(
  artifacts = [
    "junit:junit:4.13.2",
    "org.openjdk.jmh:jmh-core:1.23",
    "org.openjdk.jmh:jmh-generator-annprocess:1.23",
  ],
  lock_file = "://maven_install.json",
  repositories = [
    "https://maven.google.com",
    "https://repo1.maven.org/maven2",
  ],
)
use_repo(maven, "maven", "unpinned_maven")
```

This will fetch the pmd rule into your project.

## Creating a Lint Target

Next, you will create a lint target in your `BUILD` file. Here is an example for a Java project:

```
load("@rules_pmd//pmd:defs.bzl", "pmd_test")
...
load("@rules_pmd//pmd:defs.bzl", "pmd_test")
...
```

```
pmd_test(  
  name = "pmd_analysis",  
  srcs = glob(["src/main/java/**/*.java"]),  
  report_format = "text",  
  rulesets = ["pmd_rules.xml"],  
)  
...
```

## Running the linter

After configuring your project, you can execute the linter using the following command:

```
bazel build //app/hellotest:pmd_analysis
```

The resulting report are going to be stored at `app/hellotest/pmd_analysis_pmd_report.txt` containing feedback as:

```
.../GreeterService.java:3: CommentRequired: Class comments are  
required...  
.../GreeterService.java:3: AtLeastOneConstructor: Each class  
should declare at...  
.../GreeterService.java:4: MethodArgumentCouldBeFinal: Parameter  
'name' is not...  
.../GreeterService.java:4: CommentRequired: Public method and  
constructor commen...  
.../HelloTestMain.java:3: UseUtilityClass: All methods are  
static. Consider usi...  
.../HelloTestMain.java:3: CommentRequired: Class comments are  
required...  
.../HelloTestMain.java:4: MethodArgumentCouldBeFinal: Parameter  
'args' is not as...  
.../HelloTestMain.java:4: CommentRequired: Public method and  
constructor comment...  
.../HelloTestMain.java:5: AvoidLiteralsInIfCondition: Avoid  
using Literals in...  
.../HelloTestMain.java:6: SystemPrintln: System.out.println is  
used...
```

```
.../HelloTestMain.java:9: LocalVariableCouldBeFinal: Local
variable 'service' co...
.../HelloTestMain.java:10: SystemPrintln: System.out.println is
used...
```

Alternatively, you can incorporate this command into a Git hook.

Here is how you can set it up as a pre-commit hook:

1. **Navigate to the .git/hooks directory:** This directory is inside your Git repository. If your repository is `my_project`, the path would be `my_project/.git/hooks`.
2. **Create a Pre-commit Hook:** Inside the `.git/hooks` directory, create a file named `pre-commit`. If it already exists, you can edit the existing file.
3. **Edit the Pre-commit Hook:** Open the `pre-commit` file in a text editor and add the following script:

```
#!/bin/sh
# Run Bazel build for PMD analysis
bazel build //app/hellotest:pmd_analysis
# Check the exit status of the previous command
if [ $? -ne 0 ]; then
    echo "PMD analysis failed. Fix the issues before
    committing."
    exit 1
fi
```

This script runs the Bazel build command and checks if it exits with a status other than 0 (indicating an error). If there is an error, it prints a message and exits with status 1, preventing the commit from proceeding.

4. **Make the Hook Executable:** Change the file permissions to make the `pre-commit` script executable.

```
chmod +x pre-commit
```

5. **Testing the Hook:** Test the hook by trying to commit to your repository. If the PMD analysis finds issues, the

commit should be blocked with the message provided in the script.

Remember, Git hooks are local to your repository and are not pushed to the remote server. If you want to enforce this hook for all contributors, consider using a server-side hook or integrating it into a continuous integration (CI) pipeline.

Integrating linters such as Checkstyle and PMD into Bazel for Java projects streamlines the process of maintaining code quality. By incorporating these tools into the build process, developers can ensure that their code adheres to established standards and best practices, leading to more reliable and maintainable software.

## **Conclusion**

This chapter provided a comprehensive exploration of modern development workflows, focusing on the evolution of code contribution processes, the importance of traceability, the role of code reviews, and the effective management of CI/CD pipelines, particularly in the context of Bazel's application in Monorepo environments. This evolution, marked by the advent of tools like Git and GitHub, has transformed code contribution into a more streamlined and transparent process. The strategic use of Bazel, in conjunction with effective branching strategies and detailed commit comments, has further enhanced the process, ensuring the integrity of larger codebases and improving the overall efficiency and reliability of the development cycle. These advancements not only signify technological progress but also reflect a deeper comprehension of collaborative dynamics in software development.

Looking ahead, the balance between automation and human intervention emerges as a crucial aspect of future development workflows. While tools like Bazel and CI/CD pipelines significantly reduce manual errors and accelerate

the development process, the irreplaceable human element—characterized by creativity and critical thinking—remains vital. Maintaining this balance is essential for fostering innovation and developing robust, user-centric software. Moreover, as we navigate the complexities of large-scale software projects, particularly in Monorepo settings, it becomes imperative to continuously adapt and refine workflows to meet evolving project needs and leverage emerging technologies effectively. The chapter underscores the importance of training and documentation in empowering teams to utilize these advanced tools efficiently, setting the stage for the next era of software development marked by efficiency, quality, and innovation.

## **Recommended Reading**

- Code Contribution Workflows and Traceability:
  - Understanding Version Control:
    - “Pro Git” by Scott Chacon and Ben Straub: <https://git-scm.com/book/en/v2>
    - “Version Control with Git” by Jon Loeliger and Matthew McCullough: <https://www.oreilly.com/library/view/version-control-with/9781449345037/>
  - Effective Branching Strategies:
    - “Comparing Workflows” on Atlassian Git Tutorial: <https://www.atlassian.com/git/tutorials/comparing-workflows>
    - “Successful Git Branching Model” by Vincent Driessen: <https://nvie.com/posts/a-successful-git-branching-model/>
- Code Reviews and Collaborative Development:

- Best Practices in Code Review:
  - “Best Kept Secrets of Peer Code Review” by Jason Cohen:  
<https://smartbear.com/resources/ebooks/best-kept-secrets-of-peer-code-review/>
  - “Code Review Practices for Collaborative Software Development”:  
<https://www.infoq.com/articles/code-review-practices/>
- Leveraging Tools for Collaborative Development:
  - “GitHub and GitLab: A Comparison of Key Differences”:  
[https://www.slant.co/versus/109/110/~github\\_vs\\_gitlab](https://www.slant.co/versus/109/110/~github_vs_gitlab)
  - “Collaborative Development with Pull Requests”:  
<https://github.blog/2020-12-08-how-to-get-started-with-pull-requests/>
- CI/CD Pipelines and Bazel:
  - Continuous Integration and Deployment:
    - “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation” by Jez Humble and David Farley:  
<https://www.amazon.com/Continuous-Delivery-Deployment-Automation-Addison-Wesley/dp/0321601912>
    - “Implementing Continuous Integration and Continuous Delivery (CI/CD)”:  
<https://www.cloudbees.com/continuous-delivery/ci-cd>
  - Understanding and Implementing Bazel:

- “Bazel Documentation”:  
<https://docs.bazel.build/versions/main/bazel-overview.html>
- “Building Large-Scale Apps with Monorepo and Bazel”:  
<https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel>
- Additional Topics:
  - Monorepos and Large-Scale Development:
    - “Monorepos: Please don’t!”  
<https://mattklein123.dev/2020/01/18/monorepos-please-dont/>
    - “Scaling a Codebase with Monorepos and Code Sharing”:  
<https://medium.com/@maoberlehner/scaling-a-codebase-with-monorepos-and-code-sharing-a8c2ece5324c>
  - Advanced Git Techniques:
    - “Advanced Git Tutorials”:  
<https://www.atlassian.com/git/tutorials/advanced-overview>
    - “Git for Ages 4 and Up”:  
<https://www.youtube.com/watch?v=1ffBJ4sVUb4>



# **CHAPTER 8**

# **Structuring Monorepos for Success**

## **Introduction**

Welcome to the chapter titled “*Structuring Monorepos for Success*” in our journey through mastering Bazel. In this chapter, we will delve into the core principles and strategies for designing an effective Monorepo layout, a task fundamental to optimizing both development workflows and building performance in large-scale projects. Monorepos, which houses all of an organization’s code in a single repository, presents unique challenges and opportunities. Understanding how to effectively structure your Monorepo is key to leveraging the full potential of Bazel. We will explore best practices for repository layout, discuss the balance between granular and coarse structuring, and examine how to align your Monorepo’s structure with your team’s workflow and Bazel’s capabilities.

The subsequent sections of this chapter will guide you through organizing code into packages and modules, and managing dependencies across projects. Organizing code effectively is critical within a Monorepo environment, as it directly impacts build times, dependency management, and the ease of navigating and understanding the codebase. We will provide practical advice on how to structure your packages and modules for maximum efficiency and clarity. In addition, managing dependencies is a complex task in Monorepos, but with Bazel’s powerful tools, it becomes more manageable. We will explore strategies for declaring

and managing dependencies across various projects within the Monorepo, ensuring consistent builds and reducing the risk of dependency hell. This chapter aims to equip you with the knowledge to create a Monorepo that is not just manageable, but a robust foundation for your development efforts.

## **Structure**

In this chapter, we will cover the following topics:

- Designing an Effective Monorepo Layout
  - Directory Structure Best Practices
  - Code Sharing and Reusability
  - Testing Strategies
  - Centralized Configuration Management
  - Refactoring and Code Maintenance
  - Security Considerations
  - Common Pitfalls and Ways to Avoid Them
- Organizing Code into Packages and Modules
- Naming Conventions for Packages and Modules

## **Designing an Effective Monorepo Layout**

We will explore best practices for directory structuring, dependency management, and build optimization, ensuring that your Monorepo remains manageable and performant as it grows. Through practical guidelines and real-world examples, you will gain insights into creating a Monorepo layout that is both robust and adaptable to evolving project needs.

# Directory Structure Best Practices

In the context of a monorepository, the organization of your directory structure is key to managing complexity, ensuring efficient builds, and maintaining clarity as your project scales. This section focuses on best practices for organizing your Monorepo, with an emphasis on functional separation, hierarchical organization, centralized dependencies, dedicated tools, and DevOps integration.

## **Split Functional and Non-Functional Code**

Functional code, such as system-specific implementations and applications, should be distinctly separated from non-functional code like utility libraries. For instance, you might have directories like `/services` for your application code and `/libraries` for shared utilities. This separation ensures that changes in functional code do not unnecessarily impact utility libraries and vice versa. For example:

`/services/payment-service` - Contains code specific to the payment processing system.

`/libraries/string-utils` - Houses generic string manipulation functions.

## **Hierarchical Structure**

Organize your source code hierarchically to mirror the modular architecture of your application. This approach makes navigation intuitive and reflects the interdependencies within your codebase. To exemplify, within a service directory, you might have:

`/services/user-management/authentication`

`/services/user-management/profile-management`

## **Centralized Third-Party Dependencies**

Maintain a central directory, such as `/third-party`, for all external dependencies. This simplifies updates and audits, providing a single point of reference for all external libraries.

Within this directory, dependencies can be further organized by their purpose or nature. Example:

```
/third-party/logging  
/third-party/database-drivers
```

### **Dedicated Tools Directory**

Create a dedicated directory like `/tools` or `/scripts` for build scripts, deployment tools, and other utility scripts. This separation ensures that these necessary but ancillary elements of your project are easily accessible but distinct from the main application code. For instance:

```
/tools/build  
/scripts/deploy
```

### **DevOps Directory**

Include a `/devops` directory to host all DevOps-related pipelines and scripts. This could include continuous integration configurations, deployment scripts, and infrastructure-as-code files. This centralizes DevOps resources, making them more manageable. For example:

```
/devops/ci-pipeline.yaml  
/devops/terraform
```

In conclusion, a well-structured Monorepo facilitates better organization, clearer understanding, and more efficient navigation of the codebase. By adhering to these principles, you can ensure your Monorepo remains manageable and logical, even as it grows and evolves.

## **Code Sharing and Reusability**

Operating within a Monorepo, particularly one managed with Bazel, structuring code for maximum reusability is essential. This not only streamlines development across various projects but also ensures consistency and efficiency. Here is how you can achieve this:

### **Modular Design**

Break down your codebase into modular components. Each module should encapsulate a specific functionality or a set of related functionalities. As an illustration, you might have a logging module, a data-access module, and a user-authentication module. These modules can be independently developed, tested, and reused across different projects within the Monorepo.

### **Shared Libraries**

Create shared libraries for common functionalities that are used across multiple projects. For instance, if multiple projects need to perform JSON serialization, a shared json-utils library in your monorepo can be beneficial. This library can include functions for common serialization and deserialization tasks, ensuring that these operations are standardized and optimized across all projects.

### **Consistent API Design**

Ensure that the modules and libraries have consistent and well-documented APIs. This consistency makes it easier for developers to understand and use these components in different projects. To illustrate, if all your data-access modules follow the same method naming conventions and parameter orders, developers can quickly integrate and switch between them as needed.

### **Dependency Management**

Utilize Bazel's powerful dependency management to handle inter-module dependencies effectively. Define clear BUILD files that specify dependencies for each module. This method ensures that any changes in one module do not unexpectedly break functionality in another, promoting safer code reuse.

### **Testing for Reusable Components**

Implement comprehensive testing for your modules and shared libraries. This includes unit tests, integration tests,

and possibly end-to-end tests, ensuring that these components work reliably when integrated into different projects. For instance, the `json-utils` library should have its own suite of tests, verifying that it handles various edge cases and integrates seamlessly with other modules like `data-access`.

## **Documentation and Examples**

Provide thorough documentation and real-world usage examples for your shared modules and libraries. This documentation should include setup instructions, API usage, and common use cases. For the logging module, provide examples showing how to integrate it with different types of applications within your monorepo.

By following these principles, you can maximize code sharing and reusability in your monorepo, leading to a more efficient and maintainable codebase. This strategy not only saves development time but also ensures a high level of code quality and consistency across your projects.

## **Testing Strategies**

Effective testing strategies are crucial when working with a monorepository structure setup to ensure code quality and system integrity, especially when dealing with a large, interconnected codebase. Here, we will explore best practices for setting up unit tests, integration tests, and end-to-end tests in the framework of a monorepository system, utilizing Bazel's strengths.

### **Unit Testing**

Unit tests are the first line of defense against bugs. With the Monorepo approach, it is important to structure unit tests closely with the code they are testing. For instance, if you have a directory `services/payment`, your unit tests should ideally reside in something like `services/payment/tests`. This proximity simplifies finding and running tests related to

specific components. With Bazel, you can define a `BUILD` file within the `tests` directory to specify test targets. Bazel's caching and parallel execution capabilities ensure that only the tests affected by recent changes are run, which is highly efficient for large codebases.

Example:

```
py_test(  
    name = "payment_service_tests",  
    srcs = ["test_payment_service.py"],  
    deps = ["//services/payment"],  
)
```

## Integration Testing

Integration tests ensure that different modules or services work together as expected. In the setting of a monorepository, you can leverage Bazel's ability to understand dependencies to efficiently run these tests. Specifically, if you have a service that depends on a database and a payment gateway, your integration tests should verify the interactions between these components. You can set up a `BUILD` file in your integration test directory, specifying dependencies to all relevant modules.

Example:

```
py_test(  
    name = "integration_tests",  
    srcs = ["test_integration.py"],  
    deps = [  
        "//services/payment",  
        "//services/database",  
        "//services/gateway",  
    ],  
)
```

## End-to-End Testing

End-to-end tests validate the complete workflow of your application from start to finish, mimicking real user scenarios. Considering a monorepository, these tests can become quite complex due to the numerous interactions between different parts of the application. To manage this complexity, it is advisable to use Bazel's tagging feature to identify and run end-to-end tests selectively. Tags help in categorizing tests and running them as required, without triggering the entire test suite.

Example:

```
py_test(  
    name = "e2e_tests",  
    srcs = ["test_e2e.py"],  
    deps = ["//services"],  
    tags = ["e2e"],  
)
```

## Best Practices

- **Keep Tests Close to Code:** Maintain tests in close proximity to the code they are testing. This makes it easier to understand and maintain the tests.
- **Leverage Bazel's Features:** Utilize Bazel's caching and parallel execution features to run tests efficiently.
- **Tagging and Selective Testing:** Use tags to categorize tests and run them selectively, especially for larger and more complex test suites.
- **Continuous Integration:** Integrate testing into your CI/CD pipeline to ensure tests are run automatically on every commit, preventing the merging of failing code into the main branch.

By following these strategies and leveraging Bazel's capabilities, you can set up a robust and efficient testing framework in your Monorepo, ensuring that your code



remains reliable and maintainable, regardless of its size and complexity.

## **Handling Multiple Projects**

Managing multiple projects within a single Monorepo can be challenging, but with the right strategies, it can lead to improved collaboration, consistency, and efficiency. In this section, we will explore effective methods for handling multiple projects in a Bazel-managed Monorepo, focusing on inter-project dependencies and configurations.

## **Centralized Configuration Management**

Centralizing configuration management is key in a Monorepo. Use shared BUILD files and global settings to ensure uniformity across projects. For instance, common compiler flags, linting rules, and testing frameworks can be defined in a top-level `BUILD.bazel` or in shared files like `tools/build_defs.bzl`. This tactic ensures that all projects adhere to the same standards and simplifies maintenance.

## **Structured Directory Layout**

Organize the Monorepo with a clear directory structure that separates projects while allowing shared components. A typical structure might include directories like `/projects/projectA`, `/projects/projectB`, and `/common` for shared libraries. This layout not only makes navigation easier but also clarifies boundaries between projects.

## **Managing Dependencies**

In the scenario of a Monorepo, inter-project dependencies should be handled carefully to avoid conflicts and ensure clean builds. Bazel's dependency graph comes in handy here. To give you an idea, if Project A depends on a library in Project B, you can define this dependency explicitly in Project A's BUILD file, like so: `deps =`

`[“//projects/projectB:some_library”]`. This explicitness ensures that changes in Project B trigger a rebuild of Project A, maintaining consistency.

## **Shared Libraries and Tools**

Promote code reuse by creating shared libraries and tools. These can be placed in a common directory like ``/common/utils``. Projects can then reference these tools, ensuring a DRY (Don't Repeat Yourself) codebase. For instance, a logging library in ``/common/utils/logging`` can be used by all projects, reducing redundancy and easing updates.

## **Modules.bazel or Workspace Rules for External Dependencies**

When projects within the Monorepo have external dependencies, use Bazel's workspace rules or `Modules.bazel` to fetch and manage them centrally. Define these dependencies in the ``WORKSPACE`` file at the root. This centralized management aids in updating and auditing external libraries.

## **Versioning and Releases**

Handling versioning when utilizing a Monorepo with multiple projects can be complex. Adopt a strategy that suits your release cycle, such as using tags that prefix with project names (for example, ``projectA-v1.0.0``, ``projectB-v1.0.0``). Automate version bumping and changelog generation to maintain clarity across projects.

## **Continuous Integration and Deployment**

Set up CI/CD pipelines that can handle multiple projects in a Monorepo. Use Bazel's query language to build and test only affected projects in a commit. Consider the case of a CI job that can be configured to run ``bazel test //projects/projectA/...`` if changes are detected in Project A's directory, ensuring efficient resource utilization.

By implementing these strategies, you can effectively manage multiple projects within a single Monorepo, leveraging Bazel's powerful features to maintain a clean, efficient, and scalable codebase.

## **Refactoring and Code Maintenance**

In a large Monorepo, maintaining code quality and facilitating easy refactoring are critical to the health and longevity of the project. This section provides practical tips, supported by examples, for upholding these standards.

### **Establishing a Strong Code Review Culture**

A robust code review process is vital. It not only catches bugs but also ensures code consistency and knowledge sharing. For example, within a Monorepo environment containing a mix of Java and Python projects, enforce language-specific style guides (like Google Java Style and PEP 8 for Python) during reviews. Use tools like GitHub's pull request reviews or GitLab's merge request reviews to facilitate thorough, collaborative code reviews. Encourage team members to not only focus on finding faults but also suggest improvements and share insights.

### **Leveraging Automated Code Analysis Tools**

Automated tools can significantly reduce the burden of maintaining code quality. Static analysis tools like SonarQube or ESLint can be integrated into your CI/CD pipeline to automatically flag issues like code smells, security vulnerabilities, or style violations. For instance, integrate SonarQube in your Jenkins pipeline to analyze a multi-language codebase whenever a new commit is pushed, providing immediate feedback on potential issues.

### **Implementing Continuous Refactoring**

Embrace a culture of continuous refactoring. This does not mean large, disruptive changes; rather, encourage

developers to make small, incremental improvements regularly. For instance, if a developer is working on a feature in a Python module and notices an opportunity to simplify a function or improve naming conventions, they should feel empowered to do so. This mode of operation keeps the codebase healthy and adaptable.

### **Utilizing Feature Flags for Safe Refactoring**

Feature flags can be a powerful tool for safer refactoring, especially in the context of a monorepository where changes can have wide-ranging impacts. By wrapping new code in feature flags, you can deploy changes to production without exposing them to all users. This technique allows for testing in the live environment with limited risk. For instance, if refactoring a core library used by several services in the Monorepo, introduce a feature flag to gradually roll out changes, monitoring for issues before full adoption.

### **Automated Dependency Updates**

Operating within a Monorepo, staying on top of dependency updates is crucial but challenging. Tools like *Dependabot* or *Renovate* can automate the process of updating dependencies, submitting pull requests for review when new versions are available. This ensures that your codebase remains up-to-date with the latest patches and features without manual oversight.

### **Documentation as a Maintenance Tool**

Good documentation is often overlooked as a maintenance tool. Documenting architecture decisions, coding standards, and refactoring rationales helps new team members understand the codebase and maintain consistency. An example being create a README file in each major directory of the Monorepo, describing its purpose, structure, and any special considerations.

In summary, maintaining code quality and ease of refactoring in a large Monorepo involves a combination of cultural practices, automated tooling, and continuous improvement. By embedding these practices into your development workflow, you can ensure that your Monorepo remains a healthy and efficient foundation for your software projects.

## **Security Considerations**

When working with a monorepository structure environment, addressing security concerns is paramount, given the interconnected nature of the codebase. This section delves into key security considerations you need to be aware of when managing a Monorepo, focusing on dependency vulnerabilities, code access controls, audit trails, and secret management.

### **Managing Dependency Vulnerabilities**

In Monorepos, dependencies are often shared across multiple projects, making it crucial to keep them secure. One common vulnerability is the inclusion of outdated or compromised libraries. For instance, if a widely used logging library within your Monorepo is found to have a critical security flaw, it could impact multiple projects simultaneously. To mitigate such risks, implement automated tools that scan for known vulnerabilities in dependencies. Tools like Dependabot or Snyk can be integrated into your workflow to automatically detect and propose fixes for such issues.

### **Implementing Code Access Controls**

With multiple teams working in a single repository, controlling who can access and modify specific parts of the codebase becomes essential. Demonstrating this, you might want developers to have read-only access to certain core

libraries, while granting full access to the projects they actively work on.

Git itself does not support RBAC. To implement RBAC while using Git, you would typically manage access at the repository hosting service level (like GitHub, GitLab, or Bitbucket) where you can define roles and permissions for users on specific repositories. For more granular control within a repository, you can use Git hooks or third-party tools to enforce policies based on branch or directory, aligning with the desired roles and permissions.

Additionally, incorporating code review practices, where merges into protected branches require approval from designated maintainers, ensures an additional layer of scrutiny and security.

### **Establishing Robust Audit Trails**

Maintaining a comprehensive audit trail in the framework of a monorepository system is vital for tracking changes and identifying potential security breaches. In a scenario where a critical bug is introduced, a detailed audit trail enables you to quickly trace back to the specific commit and developer responsible. This aids not only in swift resolution of the issue but also in analyzing the root cause to prevent similar occurrences in the future. Utilize tools that log all code changes, pull requests, and deployment activities. Integrating your version control system with a CI/CD pipeline can automate this process, ensuring a transparent and up-to-date audit trail.

### **Secret Management**

Managing secrets, such as API keys, credentials, and certificates, is a critical aspect of Monorepo security. Poor handling of secrets can lead to severe security breaches. To illustrate, suppose an API key is inadvertently committed to the Monorepo; it could potentially be accessed by unauthorized personnel, leading to data breaches or

unauthorized access to external services. To prevent such scenarios, it is essential to:

- **Use Secret Management Tools:** Tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault should be integrated to manage secrets securely.
- **Encrypt Secrets:** Ensure that all secrets are encrypted at rest and in transit.
- **Access Controls for Secrets:** Implement strict access controls, ensuring only authorized personnel can access sensitive information.
- **Automate Secret Rotation:** Regularly rotate secrets automatically to minimize the risk associated with compromised credentials.
- **Audit Secret Access:** Keep detailed logs of who accessed which secret and when, to monitor for any unauthorized access or anomalies.

### **Best Practices for Secure Monorepo Management**

- **Regular Security Audits:** Conduct regular security audits of your Monorepo to identify and address vulnerabilities proactively.
- **Dependency Hygiene:** Regularly update dependencies and remove unused or outdated libraries to minimize exposure to vulnerabilities.
- **Principle of Least Privilege:** Apply the principle of least privilege in access controls, ensuring team members have only the permissions necessary for their role.
- **Continuous Monitoring:** Implement continuous monitoring tools to detect and alert on suspicious activities in real-time.
- **Education and Training:** Regularly educate and train your development team on security best practices, as

human error often leads to security breaches.

By addressing these security aspects, including the critical area of secret management, you can significantly enhance the integrity and resilience of your Monorepo, ensuring that it remains a robust and secure foundation for your software development endeavors.

## **Common Pitfalls and Ways to Avoid Them**

Designing a Monorepo layout with Bazel presents unique challenges. Without careful planning, you may encounter pitfalls that can significantly impede the efficiency and scalability of your project. This section outlines some of these common pitfalls and offers strategies to avoid them, supplemented with illustrative examples.

- **Overly Complex Directory Structures**

**Pitfall:** A common mistake is creating a deeply nested or overly complex directory structure. This can make navigation difficult and obscure the relationships between different parts of the codebase.

**Solution:** Adopt a flat and modular directory structure. For example, instead of deeply nesting related projects, group them under a single directory with clear naming conventions. This procedure enhances clarity and simplifies dependency tracking.

- **Poorly Managed Dependencies**

**Pitfall:** Inefficient dependency management, such as redundant declarations or large, unsegmented dependency blocks, can lead to longer build times and difficulty in tracking changes.



**Solution:** Use Bazel's fine-grained dependency management capabilities. Structure your BUILD files to declare only the necessary dependencies. For instance, if a module only requires a specific function from a library, avoid including the entire library as a dependency.

- **Inadequate Build Optimization**

**Pitfall:** Failing to optimize builds can result in unnecessarily long build and test times, especially as the codebase grows.

**Solution:** Utilize Bazel's caching and incremental build features. As a case in point, set up remote caching to share build artifacts across the team, reducing duplicate build efforts.

- **Neglecting Code Reusability**

**Pitfall:** Overlooking opportunities for code reuse can lead to duplicated code, increasing maintenance efforts and potential for errors.

**Solution:** Structure your Monorepo to promote reusability. Create shared libraries for common functionalities and document their usage. One example is a shared utility library for date-time operations that can be reused across different projects within the Monorepo.

- **Insufficient Testing Strategies**

**Pitfall:** Inadequate testing setup with the Monorepo approach can lead to uncaught bugs and regressions.

**Solution:** Implement comprehensive testing strategies using Bazel's testing tools. Structure your tests to run locally relevant tests first, then broader integration tests. For example, set up test suites that are automatically triggered for changes in specific directories.

- **Scaling Challenges**

**Pitfall:** As the Monorepo grows, it may become difficult to maintain performance and manageability.

**Solution:** Regularly refactor and modularize your codebase. Implement performance monitoring tools to identify and address bottlenecks. For instance, use Bazel's profiling tools to pinpoint slow build processes and optimize them.

- **Version Control Integration Issues**

**Pitfall:** Misalignments between Bazel and version control systems can lead to versioning and integration challenges.

**Solution:** Establish clear versioning practices and integrate Bazel seamlessly with your version control system. To illustrate, use Bazel's workspace rules to manage external dependencies in a version-controlled manner.

By recognizing and addressing these pitfalls, you can design a Monorepo layout that leverages Bazel's strengths, ensuring that your project remains efficient, scalable, and maintainable.

## **Organizing Code into Packages and Modules**

In an advanced Bazel environment, the strategic organization of code into packages and modules is key to maintaining a scalable, efficient, and high-quality codebase. This section delves into sophisticated approaches to package and module design, focusing on design principles, maintainability, and code quality.

### **Strategic Design Principles**

The cornerstone of an effective Bazel Monorepo is a thoughtful design of packages and modules. Opt for a domain-driven design approach, where packages align with business capabilities rather than technical concerns. For instance, instead of a generic ``utils`` package, consider

domain-specific packages like ``payment_processing`` or ``user_authentication``. This strategy enhances understandability and reduces the cognitive load for developers navigating the codebase.

## **Package Granularity and Dependencies**

In Bazel, structuring your build with fine-grained packages enables more precise dependency management and incremental builds, improving build efficiency and parallelism. However, overly granular packages can complicate dependency tracking and increase the maintenance burden, potentially causing a “**dependency hell**” where managing inter-package dependencies becomes excessively complex.

A balanced package should encapsulate a coherent set of functionalities with minimal external dependencies. To give you an idea, a package named ``invoice_generator`` might include classes for invoice creation, formatting, and data validation, but not for payment processing, even if closely related. This separation minimizes build times and improves cache effectiveness.

## **Build File Best Practices**

In your ``BUILD`` files, be explicit about dependencies. Rely on visibility attributes to control access to packages, ensuring that only intended packages can depend on a given module. This practice not only improves security but also prevents unintended coupling. To exemplify, setting ``visibility = [“//some/other:__pkg__”]` ensures that only ``some/other`` package can access the module, preventing its use in unplanned contexts.

## **Module Cohesion and Coupling**

Maintain high cohesion within modules and low coupling between them. Modules should be designed around a single, well-defined responsibility, with minimal dependencies on other modules. For instance, a ``user_profile`` module should

handle all aspects related to user profiles but should not directly manipulate database connections, which would be the responsibility of a separate `database_access` module.

### **Maintainability and Refactoring**

Regularly refactor your packages and modules to adapt to evolving project requirements. Employ automated tools within Bazel for code analysis and linting to ensure consistency and quality. Consider the case of integrating a tool like `buildifier` that can standardize `BUILD` files structure, making them easier to read and maintain.

### **Quality Assurance with Tests**

Incorporate comprehensive testing at the package and module level. Structure your tests to mirror the package structure, ensuring that each package has corresponding unit tests. This practice not only ensures code quality but also simplifies identifying the impact of changes during development.

### **Example of Effective Organization**

Consider a project with distinct functionalities like user management, product catalog, and order processing. Organize these functionalities into separate packages: `//app/user_management`, `//app/product_catalog`, and `//app/order_processing`. Each package should have its own `BUILD` file specifying dependencies. An example being, `//app/order_processing` may depend on `//app/user_management` for user details but should not directly interact with `//app/product_catalog`, enforcing a clear separation of concerns.

In compliance with these guidelines, you can create a Bazel Monorepo that is not only efficient and scalable but also maintains high standards of code quality and maintainability.

# **Naming Conventions for Packages and Modules**

Adopting consistent and meaningful naming conventions for packages and modules is essential in a Bazel Monorepo to enhance readability, maintainability, and team collaboration. This section focuses on advanced naming strategies that reflect the functionality, domain, and structure of the codebase.

## **Reflecting Functionality and Domain**

Names should clearly indicate the purpose and domain of the package or module. Avoid vague or overly generic names. For instance, prefer specific names like ``customer_data_processing`` over ambiguous ones like ``data_handlers``. This clarity aids in quickly identifying the role and scope of a module or package.

## **Hierarchy and Structure**

Align the naming convention with the repository's hierarchy. Use a hierarchical naming structure that mirrors the directory structure. For instance, a package in the path ``//app/backend/database`` should be named in a way that reflects its path and purpose, like ``database_operations`` or ``database_utils``.

## **Consistency Across the Codebase**

Maintain consistency in naming conventions across different packages and modules. If you use a particular format like `snake_case` (``user_management``) or `CamelCase` (``UserManagement``) for package names, apply it uniformly. This consistency is crucial for predictability and ease of navigation.

## **Verbosity versus Brevity**

Strive for a balance between verbosity and brevity. Names should be long enough to convey purpose but not so long that they become unwieldy. For instance, ``auth`` is preferable

to ``authentication_process_manager`` for simplicity, yet it is descriptive enough to convey purpose.

### **Avoiding Redundancy and Tautology**

Avoid redundancy in names. Demonstrating this, instead of ``user_management_management``, simply use ``user_management``. Similarly, avoid tautological phrases like ``database_db``.

### **Versioning and Experimental Features**

For packages or modules that are under development or experimental, include tags like ``experimental`` or ``beta`` in the name. For instance, ``payment_processing_beta`` indicates a module in the testing phase. Similarly, include version numbers if multiple versions of the same module coexist, like ``data_parser_v2``.

### **Example of Effective Naming**

Consider a project with a frontend and backend. Organize and name your packages to reflect their roles and hierarchy. For example:

- Frontend packages: ``//app/frontend/user_interface``,  
``//app/frontend/api_client``
- Backend packages: ``//app/backend/user_management``,  
``//app/backend/data_storage``

Each of these names provides a clear, concise, and consistent indication of what the package contains and where it fits within the overall project structure.

By applying these naming conventions, you create a more navigable, understandable, and maintainable Bazel Monorepo. Clear and consistent names act as a guide, helping developers understand the architecture and purpose of different parts of the codebase at a glance.

## **Conclusion**

In this chapter, we learned that designing an effective Monorepo layout requires a multifaceted approach, combining best practices in directory structure with a focus on code sharing and reusability. By implementing efficient testing strategies and centralized configuration management, teams can ensure robustness and maintainability of their code. The importance of regular refactoring and vigilant code maintenance cannot be overstated, as these practices significantly contribute to the overall health and scalability of the project. Security considerations are also paramount, demanding a proactive approach to safeguard the Monorepo's integrity. Furthermore, organizing code into well-defined packages and modules, coupled with thoughtful naming conventions, not only enhances readability but also facilitates easier navigation and understanding of the codebase. Collectively, these practices form the backbone of a well-organized, efficient, and secure Monorepo, laying the groundwork for successful project development and management. In the next chapter, we will delve into the intricate world of handling extensive Monorepo codebases.

## **Recommended Reading**

Here is a list of recommended readings for various sections of your book on Bazel and Monorepo management. These resources should provide additional insights and best practices:

- “Monorepos — a practical guide” — DEV Community. A comprehensive guide that discusses the best practices for scaling a Monorepo: <https://dev.to/elliotalexander/monorepos-best-practice-for-scaling-a-jam-stack-dl>
- “Monorepo CI best practices” — Buildkite. This article offers insights into continuous integration best practices

in a Monorepo setup:  
<https://buildkite.com/blog/monorepo-ci-best-practices>

- “How to Structure a Monorepo” — Luca Pette. Luca Pette provides practical advice on organizing a Monorepo for maintainability and efficiency:  
<https://lucapette.me/writing/how-to-structure-a-monorepo/>
- “Monorepos in JavaScript & TypeScript” — Robin Wieruch. This article focuses on Monorepos in the context of JavaScript and TypeScript, offering insights into code sharing and reusability:  
<https://www.robinwieruch.de/javascript-monorepos/>
- “Testing in a Monorepo” — Turborepo. This resource covers various strategies for effective testing within a Monorepo:  
<https://turbo.build/repo/docs/handbook/testing>
- “Sharing Configurations Within a Monorepo” — DEV Community. An article discussing approaches to managing configurations centrally in a Monorepo:  
<https://dev.to/mbarzeev/sharing-configurations-within-a-monorepo-42bn>
- “Monorepos: Making Development Easier and Smoother” on DEV Community— This article provides insights into how monorepos facilitate easier development and smoother refactoring and maintenance processes.  
<https://dev.to/furqanramzan/monorepos-making-development-easier-and-smoother-3kl8>).
- An article on Goldman Sachs’ developer blog discusses the risk perspective of Monorepos and offers guidance on security considerations.  
<https://developer.gs.com/blog/posts/monorepos-from-a-risk-perspective>).



- “From Monorepo Mess to Monorepo Bliss: Avoiding Common Mistakes” on InfoQ: This article details common mistakes made when managing Monorepos and provides strategies to avoid them. <https://www.infoq.com/articles/monorepo-common-mistakes/>).
- “How to think about packages in a monorepo” on Tech Shaadi: This resource offers a comprehensive guide on organizing packages and modules in a Monorepo. <https://tech.shaadi.com/2021/08/05/how-to-think-about-packages-in-a-monorepo/>).
- “Using Nx at Enterprises” on Nx.dev: This guide provides insights into code organization and naming conventions for packages and modules in enterprise-scale Monorepos. <https://nx.dev/concepts/more-concepts/monorepo-nx-enterprise>).

These resources provide a wealth of information and practical advice for effectively managing and optimizing Monorepos in various aspects.

# **CHAPTER 9**

## **Managing Large Codebases and Scale**

### **Introduction**

In this chapter, we will delve into the intricate world of handling extensive monorepo codebases, a realm where advanced techniques and strategic foresight become indispensable. Assuming familiarity with Bazel, monorepos, and dependency management, we shift our focus to the nuanced aspects of managing both internal and external dependencies, integrating code from various repositories, and handling third-party libraries. We will explore advanced modularization strategies that aid in code sharing and reuse, emphasizing the importance of an efficient code organization for enhanced readability and maintainability. This chapter also addresses the critical aspects of managing inter-module dependencies and collaborative environment management, providing insights into refining branching, merging, and code review practices to minimize merge conflicts and build breakages. Navigating the common challenges that arise in large-scale projects, we will delve into the upkeep of Bazel build configurations, debugging build issues, and best practices to avoid common pitfalls. Furthermore, we will examine performance optimization for monorepo builds, including setting up Remote Build Execution (RBE), cache management strategies, and the use of Bazel's profiling tools to identify and rectify inefficiencies. The chapter culminates with advanced caching techniques, resource management strategies, and an in-depth look at

Starlark code optimization, all designed to elevate your mastery in managing large codebases effectively and efficiently.

## **Structure**

In this chapter, we will cover the following topics:

- Dealing with Large Monorepo Codebases
- Performance Optimization for Monorepo Builds

## **Dealing with Large Monorepo Codebases**

In the context of large monorepo codebases, the management of dependencies, both internal and external, demands a sophisticated and nuanced approach. As the scale of the monorepo expands, traditional methods of dependency management often fall short, necessitating advanced strategies to maintain efficiency and reliability.

## **Managing Internal and External Dependencies**

The crux of dependency management in a monorepo setup using Bazel lies in the adept handling of numerous and often complex dependencies. This challenge is compounded when dealing with a mix of internal and external dependencies. For internal dependencies, the key is to establish a coherent versioning strategy. This involves aligning the versions of internal packages or modules to ensure compatibility and reduce conflicts. Utilizing Bazel's workspace rules effectively allows for seamless integration of these internal dependencies, where the build system can intelligently resolve and fetch the necessary components.

In contrast, external dependencies require a different approach. Often, these are incorporated into the monorepo via external repositories. The management of these external repositories within Bazel's framework is critical. It involves defining explicit rules that not only fetch the correct version of the external dependency but also integrate it seamlessly with the existing internal codebase. This process demands a keen understanding of Bazel's workspace mechanics, ensuring that the external code is compatible with the internal build environment.

## **Integration of Code from Other Repositories**

A significant aspect of dependency management in monorepos is the incorporation of code from other repositories as internal dependencies. Tools like Copybara play a pivotal role in this process. They enable the synchronization of code between the external repositories and the monorepo, allowing for a more controlled and automated integration process. This method is particularly effective when dealing with frequently updated external codebases, as it simplifies the task of keeping the internal and external code in sync. However, it requires careful configuration to ensure that the synchronization process respects the structural and operational nuances of the monorepo.

## **Handling Third-Party Libraries**

The treatment of third-party libraries with the monorepo approach environment necessitates a strategic approach, particularly when it comes to upgrades and version management. A common practice is to copy the code of these third-party libraries and build them internally within the monorepo. This approach offers several advantages.

Firstly, it provides greater control over the versioning of these libraries, ensuring that upgrades can be managed more predictably. Secondly, it allows for the customization of these libraries to better fit the specific needs of the monorepo. Finally, building these libraries internally ensures that they are fully compatible with the monorepo's build environment, reducing the likelihood of conflicts and incompatibilities.

## **Advanced Modularization Strategies**

Modularization in the setting of a monorepository context goes beyond the basic separation of concerns; it involves designing modules that are both autonomous and synergistic. Each module should encapsulate a distinct set of functionalities and have clearly defined interfaces. This isolation not only promotes easier testing and maintenance but also facilitates parallel development across teams. Nonetheless, the challenge lies in balancing module independence with the need for collaboration and cohesion across the codebase.

To achieve this, developers should employ a tiered approach to modularization. Core foundational modules, which provide generic functionalities used across various projects, should be distinguished from higher-level modules that are more specialized. This hierarchical structuring aids in understanding the dependency graph and ensures that changes in foundational modules are made judiciously, given their widespread impact.

## **Code Sharing and Reuse**

In a large monorepo, the efficiency of code sharing and reuse becomes a critical factor. Structuring a monorepo to facilitate this requires a strategic approach to code organization. Shared libraries or modules should be

centrally located and easily discoverable. It's imperative to maintain these shared resources with strict versioning control and clear documentation, as they form the backbone of multiple projects within the monorepo.

However, indiscriminate code sharing can lead to unintended dependencies and increase the complexity of the build process. Therefore, establishing guidelines on when and how to abstract common functionalities into shared modules is crucial. These guidelines should strike a balance between DRY (Don't Repeat Yourself) principles and the pragmatic needs of individual projects within the monorepo.

## **Efficient Code Organization and Readability**

Organizing code in a large monorepo demands a methodical approach. One effective technique is to group related projects or modules into sub-directories, each with its own Bazel build files. This logical grouping not only enhances readability but also streamlines the build process.

Moreover, consistent naming conventions, clear documentation, and adherence to a well-defined coding standard are vital. They ensure that developers, irrespective of their project involvements, can easily navigate and understand different parts of the monorepo. This uniformity becomes particularly valuable in large teams where developers might move between projects.

## **Managing Inter-Module Dependencies**

Inter-module dependencies in a large monorepo require careful management to avoid a tangled web of connections that can hinder development and scaling. In Bazel, explicit declaration of dependencies in BUILD files helps maintain a

clear understanding of module interactions. Developers should be encouraged to minimize dependencies, especially between high-level modules, to reduce build times and complexity.

Additionally, employing tools like dependency graph visualizers can provide valuable insights into the dependency structure, helping identify potential areas for optimization. Regular reviews of these dependencies, especially after significant development phases, are essential to ensure that the dependency graph remains manageable and logical.

## **Advanced Strategies for Collaborative Environment Management**

Considering a monorepository managed by Bazel, the collaborative environment hinges on a fine-tuned balance between autonomy and coordination. Advanced strategies involve creating a structured yet flexible environment where teams can work independently without impeding each other's progress. This necessitates a comprehensive understanding of Bazel's build graph and dependency tracking mechanisms. Teams must be adept at identifying and isolating changes that could have wide-reaching effects across the repository.

One critical aspect is the establishment of clear conventions for declaring and managing dependencies within **BUILD** files. This practice ensures that teams are aware of the implications of their changes on other parts of the monorepo. Furthermore, leveraging Bazel's query language to analyze the impact of proposed changes can serve as a preemptive measure to avoid unintended side effects.

## **Refining Branching, Merging, and Code Review Practices**

Branching and merging practices in a Bazel-managed monorepo requires a nuanced approach. Given Bazel's efficiency in handling incremental builds, a common practice is to encourage shorter-lived branches to reduce the complexity of merges. On the other hand, in a large monorepo, even small changes can have significant impacts. Therefore, implementing a robust code review process is paramount. This process should not only focus on the code changes themselves but also on the potential build and test impacts as assessed by Bazel.

For merging strategies, it's advisable to lean towards a rebase-centric workflow, which helps in maintaining a clean and linear history. This strategy, combined with Bazel's ability to precisely determine affected targets, allows for smoother integration of changes and reduces the likelihood of merge conflicts.

Moreover, in the context of code reviews, employing Bazel's remote caching and remote execution features can expedite the validation of changes. By caching build and test outputs, teams can quickly verify the impact of changes across different branches, making the code review process more efficient and reliable.

## **Minimizing Merge Conflicts and Build Breakages**

Minimizing merge conflicts and building breakages in a large monorepo managed by Bazel involves a proactive and preventive approach. One effective method is to use Bazel's sandboxing capabilities to ensure that changes are tested in an isolated environment, mirroring the main build environment as closely as possible. This tactic helps to



catch potential breakages before they are merged into the main branch.

Furthermore, setting up automated tests and builds using Bazel as part of the continuous integration pipeline is crucial. This setup not only validates individual changes but also monitors the health of the main branch. Automated tests should be comprehensive yet targeted, focusing on areas of the codebase most likely to be affected by the changes. Bazel's fine-grained dependency tracking aids in executing only the necessary tests, making this process both efficient and thorough.

Lastly, fostering a culture of ownership and responsibility is key. Encouraging developers to understand the broader impact of their changes and to take ownership of the entire lifecycle of their code - from development to deployment - plays a vital role in minimizing conflicts and building issues.

## **Navigating Common Challenges**

Monorepos managed by Bazel often encounter unique challenges that stem from their complexity and scale. Among these, dependency conflicts and build inefficiencies are the most prevalent. A common scenario is the inadvertent introduction of conflicting dependency versions, which can lead to failed builds or runtime errors. To mitigate this, it is essential to establish a centralized dependency management system. This system should enforce consistent versioning across the entire repository, ideally automated through scripts or Bazel's own dependency resolution mechanisms.

Another frequent issue arises from the misuse of Bazel's powerful, yet intricate, build configurations. Misconfigured build rules or incorrect use of Bazel's query language can lead to suboptimal build performance or even incorrect build outputs. Regular audits of build configurations and rules,

accompanied by thorough documentation, are crucial. These audits should focus on identifying redundant or conflicting build rules and streamlining build paths.

## **Upkeeping Bazel Build Configurations**

Maintaining and updating Bazel build configurations in a large monorepo is a continuous process. As the codebase evolves, so too must the build configurations. This maintenance involves not just updating Bazel versions or rules but also optimizing build files for changing dependencies and project structures. One effective strategy is to modularize build configurations. By encapsulating specific build logic within modular, reusable build files, updates become more manageable and less prone to errors.

Also, it's advisable to integrate automated checks into the continuous integration pipeline that enforces coding and configuration standards. These checks can include linters for **BUILD** files or custom scripts that validate the adherence to established best practices. Automating the detection of common misconfigurations can significantly reduce the burden of manual reviews and increase the reliability of the build process.

## **Debugging Build Issues**

Debugging in a Bazel-managed monorepo demands a comprehensive understanding of both Bazel's execution model and the specific build architecture of the project. When tackling build failures or performance issues, using Bazel's built-in profiling tools is a primary step. These tools can provide detailed insights into build processes, highlighting bottlenecks or inefficiencies.

Bazel's built-in profiling tools offer a comprehensive suite of features to analyze and optimize build performance, crucial for developers aiming to streamline their build processes. By

utilizing the `--profile=<file>` flag, developers can instruct Bazel to generate detailed JSON profiles capturing the timing information of each build step, enabling a granular analysis of the build process. This profiling data, when visualized using tools like the Chrome Trace Viewer (`chrome://tracing`), provides an intuitive graphical representation of the build's execution timeline, allowing developers to pinpoint inefficiencies and bottlenecks. Additionally, Bazel's `aquery` tool provides insights into the action graph of the build, offering a deeper understanding of how different rules and targets interact and contribute to the overall build time.

Moreover, Bazel's profiling capabilities extend to critical path analysis, helping developers identify the sequence of tasks that directly impacts the build's duration. By focusing on optimizing these critical tasks, developers can achieve significant reductions in build times. For teams leveraging Bazel's remote caching and execution features, the profiling tools also offer valuable metrics on cache performance, including hit and miss rates, which are instrumental in diagnosing and optimizing cache utilization. Through the Build Event Protocol (BEP), Bazel provides real-time data streaming of build events, enabling further customization and analysis of build metrics, thus offering a robust framework for continuous build performance improvement.

In cases of complex build errors, Bazel's query language becomes an invaluable tool. It allows developers to dissect the build graph, understand dependencies, and trace the origins of issues. It's also beneficial to develop a structured approach to debugging, starting from the reproduction of the issue in a controlled environment, followed by incremental isolation and testing of suspected components.

For persistent or non-obvious issues, engaging with the broader Bazel community can provide additional perspectives and solutions. Participating in forums or

contributing to open-source repositories related to Bazel can yield insights from other professionals who have faced similar challenges.

## **Best Practices and Common Pitfalls**

Managing dependencies with Bazel presents a unique set of challenges and opportunities. Advanced users of Bazel must navigate this landscape with a keen understanding of best practices while being aware of common pitfalls that can hinder the efficiency and scalability of their projects.

One of the core principles in handling dependencies in a large monorepo is ensuring minimal and precise dependency declarations. In Bazel, this entails explicitly specifying the exact dependencies needed for each build target. This practice not only enhances the clarity of each module's requirements but also optimizes build performance by avoiding unnecessary rebuilds of unrelated targets. However, it's crucial to balance this granularity with the overhead of managing a large number of fine-grained dependencies. Excessive fragmentation can lead to a bloated and difficult-to-maintain build configuration.

Another best practice is leveraging Bazel's capability for remote dependency caching and fetching. For large monorepos, this feature can significantly reduce build times across different machines and CI environments. It's important, though, to ensure that the cache is reliably updated and that dependencies are correctly versioned. Failure to do so can lead to inconsistent builds or issues with reproducibility.

Advanced users should also be adept at managing third-party dependencies. Utilizing tools like Bazel's external repository rules (for example, `http_archive`, `git_repository`) allows for seamless integration of external libraries and frameworks. However, this integration

introduces the risk of dependency conflicts and version mismatches. To mitigate this, it's advisable to centralize and standardize third-party dependency declarations and ensure that they are compatible with the monorepo's overall dependency graph.

Furthermore, in large monorepos, the efficient handling of transitive dependencies becomes paramount. For example, if target A depends on target B, and target B depends on target C, then target A has a transitive dependency on target C, meaning changes in C could impact the build or test outcomes of A even though A does not directly depend on C. Bazel's strict dependency checking helps in maintaining a clear dependency graph, but it requires developers to be vigilant about declaring all necessary transitive dependencies. Overlooking this can lead to failed builds or runtime errors, especially as the codebase evolves and dependencies get updated.

Versioning of dependencies in the scenario of a monorepo also demands careful consideration. While the temptation to always use the latest versions of dependencies exists, it can lead to instability and compatibility issues. Establishing a policy for regular and controlled dependency updates can help maintain stability while still benefiting from updates and security patches. A common versioning policy is to use Semantic Versioning (SemVer) for rules and dependencies, where updates are classified as major, minor, or patch, ensuring that builds remain stable and predictable as components evolve, such as defining a rule that only allows minor and patch updates automatically to prevent breaking changes.

Finally, a common pitfall in dependency management within large monorepos is neglecting the impact of dependencies on build time and resource utilization. Optimizations, such as using `select()` statements for conditional dependencies and minimizing the use of heavy libraries, are essential in

maintaining an efficient build process. Developers need to continuously monitor and profile builds to identify any dependencies that are disproportionately affecting build performance.

In summary, managing dependencies in large monorepo codebases with Bazel requires a deliberate and meticulous approach. By adhering to best practices such as precise dependency declarations, effective use of remote caching, careful integration of third-party libraries, diligent management of transitive dependencies, thoughtful versioning strategies, and continuous optimization of build performance, developers can harness the full potential of Bazel when utilizing a monorepo setup. Even so, remaining vigilant of the common pitfalls such as dependency fragmentation, cache inconsistencies, version conflicts, and build inefficiencies is equally crucial in ensuring the long-term success and scalability of the project.

## **Performance Optimization for Monorepo Builds**

In this section, we delve into advanced techniques specifically tailored for optimizing performance in monorepo builds using Bazel. Our focus will be on strategies that go beyond basic setup and configuration, targeting software developers who are already familiar with Bazel, monorepos, and dependency management within Bazel.

## **Setting Up and Configuring RBE for Large-Scale Monorepos**

The initial step towards leveraging RBE involves its meticulous setup and configuration tailored to the specific needs of a large-scale monorepo. This process includes the selection of an appropriate execution environment that

aligns with the technical requirements and constraints of the monorepo. Crucial in this setup is the configuration of Bazel's remote execution settings, ensuring that the system is primed for remote execution without compromising the integrity and security of the build processes.

The effective use of RBE in large-scale monorepos necessitates a deep understanding of the build environment, the dependencies involved, and the nature of the tasks that will be offloaded. The configuration phase should be approached with a focus on scalability and flexibility, acknowledging the dynamic nature of monorepos where the codebase and dependencies are continuously evolving.

## **Strategies for Cache Management and Sharing**

Once RBE is operational, the next significant aspect is the implementation of robust cache management and sharing strategies. Effective caching in the context of RBE is not just about storing and retrieving build artifacts; it's about intelligent cache invalidation, efficient artifact storage, and ensuring consistency across the team's build environments.

Cache management in RBE requires a fine balance: caches need to be sufficiently up-to-date to ensure that they reflect the latest state of the codebase, but also stable enough to avoid unnecessary invalidations that lead to rebuilds. This balance is critical in monorepos due to the high frequency of code changes and the interdependencies within the codebase.

Sharing caches among team members or across CI/CD pipelines can dramatically enhance build efficiency. It reduces the time spent on rebuilding similar or identical artifacts and ensures a consistent build environment across different machines and setups. This strategy, however,

demands careful management to avoid conflicts and maintain the integrity of the build artifacts.

## **Utilizing Bazel's Profiling Tools to Identify Bottlenecks**

Bazel's profiling tools serve as a cornerstone in identifying performance bottlenecks. These tools are designed to provide a granular view of the build process, enabling developers to dissect and examine each phase in detail. By engaging with Bazel's profiling capabilities, developers can track the time spent in various activities, such as fetching dependencies, analyzing targets, and executing build actions. This detailed breakdown is instrumental in identifying stages that disproportionately contribute to longer build times.

The process of utilizing these profiling tools begins with enabling profiling in Bazel builds (by adding the `--profile=/path/to/output/profile.json` parameter in the command line) and subsequently generating detailed logs. These logs can reveal granular insights, such as "Action cache hit for action 'Compiling src/main/java/com/example/app/Main.java', taking 0.045s to complete," helping developers understand specific build step performances and cache utilization. These logs offer insights into the time distribution across different build phases. For a developer, the goal is to scrutinize these logs to discern patterns and anomalies. For instance, if a significant portion of build time is spent in dependency fetching, it might signal inefficiencies in dependency management or network-related issues.

## **Analyzing Build Performance Data to Pinpoint Inefficient Patterns and**



## **Configurations**

The next step is the meticulous analysis of the build performance data gathered. This analysis is not merely about observing the durations but understanding the underlying causes of inefficiencies. It requires a comprehensive examination of how to build tasks are structured and how dependencies are managed. Are there redundant or unnecessary dependencies causing longer build times? Is the configuration optimally structured to exploit Bazel's build caching?

The effectiveness of this analysis lies in its ability to transform insights into actionable strategies. For example, upon identifying that certain targets consistently trigger extensive rebuilds, a developer might need to re-evaluate the granularity of targets or refine the dependency declarations. This phase is about connecting the dots between what the data reveals and the specific aspects of the monorepo build process that needs refinement.

## **Advanced Caching Techniques**

The cornerstone of these advanced caching strategies is the deployment of incremental and distributed caching mechanisms. Incremental caching allows for the reuse of previously computed build outputs, significantly reducing the time and resources required for subsequent builds. This method is particularly effective in monorepos where changes are often localized to specific areas of the codebase, thereby limiting the need to rebuild the entire repository. The key to successful incremental caching lies in the meticulous tracking of dependencies and source file changes, ensuring that only the necessary parts of the codebase are rebuilt.

Complementing incremental caching, distributed caching mechanisms distribute the cache across multiple machines

or even across a network. This mode of operation is particularly beneficial for teams working on the same monorepo, as it allows for the sharing of build outputs, thereby reducing redundant build efforts across the team. Distributed caching harnesses the collective resources of the team, leading to significant reductions in build times, especially in scenarios where similar build tasks are executed by different team members.

Equally important to the implementation of caching mechanisms is the development of robust cache invalidation strategies. The challenge lies in striking a balance between the need for speed and the need for consistency. Ineffective cache invalidation can lead to stale or incorrect build outputs, compromising the integrity of the build process. Hence, it is crucial to implement strategies that accurately detect when cached data is no longer valid and needs to be regenerated. This involves sophisticated tracking of changes not only in the source code but also in build configurations and dependencies. Effective cache invalidation ensures that the build system reliably reflects the most current state of the codebase, while also leveraging cached data to the fullest extent possible to accelerate build times.

## **Parallelism and Resource Management**

Maximizing build parallelism involves a deep understanding of Bazel's capabilities to manage and configure resource constraints. Bazel's build system is inherently designed to support the parallel execution of tasks, but the extent of this parallelism is heavily dependent on the correct configuration of resource constraints. Developers need to fine-tune these settings based on the specific requirements of their projects and the hardware capabilities of their build environment. This includes setting appropriate values for

flags such as `--local_cpu_resources` and `--local_ram_resources`, which dictate how Bazel allocates CPU and RAM resources during the build process. An optimal configuration ensures that Bazel utilizes the available resources effectively, leading to faster build times.

Balancing CPU and memory usage is equally important, especially in large monorepos where resource demands can be significant. The key challenge lies in finding a balance that maximizes CPU utilization without overloading the memory, which can lead to thrashing and significantly slow down the build process. Developers must consider factors such as the size and complexity of the codebase (like cyclomatic complexity), the number of dependencies, and the typical workload patterns. By analyzing these factors, one can identify the optimal ratio of CPU to memory usage that ensures a smooth and efficient build process. This might involve adjusting the granularity of build targets to better distribute the load or restructuring certain parts of the codebase to reduce memory-intensive operations.

## **Dynamic Build Graph Optimization**

Dynamic build graph optimization is predicated on the idea of minimizing the build graph's size and complexity. This process involves a meticulous analysis and restructuring of the dependencies within the monorepo. By identifying and eliminating redundant or unnecessary dependencies, the size of the build graph can be substantially reduced. This reduction is not merely a numerical advantage; it directly translates into faster build times, as Bazel has fewer paths to traverse and fewer dependencies to check for changes.

Furthermore, implementing dynamic build graphs is an advanced technique that can significantly improve build times in large monorepos. Unlike static build graphs that remain constant irrespective of the build context, dynamic

build graphs are context-aware. They adapt based on the specific components being built, the changes made in the codebase, or the targets specified for the build. This adaptability allows for a more focused and efficient build process. By constructing a build graph that only includes the necessary elements for a given build scenario, the complexity and duration of the build process are greatly reduced.

The crux of utilizing dynamic build graphs effectively lies in their intelligent configuration and integration into the build process. This requires a deep understanding of the interplay between various modules and components within the monorepo. Developers must carefully analyze the dependency graph to identify potential areas for optimization, such as common libraries or frequently changed modules. By structuring these elements in a way that maximizes the benefits of a dynamic build graph, teams can achieve significant improvements in build performance.

In essence, dynamic build graph optimization serves as a potent tool in the arsenal of a developer dealing with large monorepo codebases in Bazel. It requires a keen understanding of the underlying architecture of the codebase and a strategic approach to dependency management. When implemented correctly, it can lead to more efficient, faster, and more reliable builds, thus enhancing overall development productivity and performance in large-scale software projects.

## **Developing Custom-Build Rules for Performance-Critical Scenarios**

The development of custom-build rules in Bazel is a powerful approach to address unique performance needs within a monorepo. These custom rules, tailored to the

specific requirements and characteristics of your codebase, allow for more granular control over the build process. In performance-critical scenarios, such as builds that involve heavy computational tasks or large data processing, custom rules can significantly reduce build times. They achieve this by optimizing the way dependencies are handled and by streamlining the execution of complex tasks. The key lies in understanding the specific bottlenecks of your build process and crafting rules that directly mitigate these inefficiencies.

When designing these rules, it's essential to focus on maximizing the reuse of build outputs and minimizing redundant computations. This involves a deep understanding of the dependencies within your codebase and the interplay between different components during the build process. By creating rules that accurately reflect these dependencies and efficiently orchestrate the build tasks, you can achieve significant performance gains.

## **Optimizing Existing Build Rules for More Efficient Execution**

While developing new custom rules is important, optimizing existing build rules plays an equally critical role in enhancing build performance. Within a monorepo environment, where numerous projects and components coexist, ensuring that existing build rules are optimized for this complex environment is paramount. This optimization often involves refactoring rules to reduce their computational overhead, improve their cacheability, and enhance their parallelizability.

One effective strategy is to analyze and refactor the inputs and outputs of existing rules. By precisely defining what constitutes an input and an output for each rule, you can avoid unnecessary rebuilds triggered by changes unrelated to the rule's actual dependencies. This technique not only

enhances the efficiency of individual builds but also contributes to a more streamlined overall build process.

Moreover, in monorepo environments, it's crucial to ensure that build rules are scalable and adaptable to the evolving size and complexity of the codebase. This may involve modularizing complex rules into smaller, more manageable components or designing rules that can dynamically adjust their behavior based on the context of the build.

## **Performance Optimization for Monorepo Builds**

One of the crucial aspects of maintaining efficiency is the integration of Bazel builds within CI/CD pipelines. This integration is not merely about ensuring that Bazel runs within these systems but optimizing the entire process to leverage the strengths of both Bazel and the CI/CD ecosystem for maximum efficiency and reduced build times.

A pivotal strategy in this integration lies in understanding and manipulating the way Bazel handles dependencies and artifacts within the CI/CD workflows. Traditional CI/CD systems often operate under the assumption of starting from a clean slate, which can lead to redundant computation and time wastage, especially in the context of large monorepos where dependencies can be extensive and complex. To mitigate this, one must tailor the CI/CD environment to recognize and utilize Bazel's powerful dependency analysis and incremental build capabilities.

The key is to configure the CI/CD system to maintain a persistent workspace where Bazel can operate. This setup allows Bazel to effectively utilize its incremental build features, as the build system can quickly identify changes and rebuild only the necessary parts of the project. Such an approach drastically reduces build times compared to traditional full rebuilds. Furthermore, it's essential to ensure

that the CI/CD system correctly handles the caching of Bazel's output. Effective cache utilization not only speeds up the build process but also minimizes network traffic, which is especially beneficial in distributed CI/CD environments.

Handling dependencies in CI/CD pipelines also requires a strategic approach. In the context of a monorepository, changes in one part of the repository can potentially affect several other parts. Efficient handling of these dependencies means that the CI/CD system, in conjunction with Bazel, needs to precisely identify and act upon these interdependencies. This process involves setting up the CI/CD system to trigger builds and tests based on the dependency graph calculated by Bazel. Such an approach ensures that any change in the repository triggers a targeted and efficient CI/CD process, reducing the workload on the servers and speeding up the feedback loop to developers.

Moreover, the management of artifacts within the CI/CD pipelines plays a significant role. Bazel's ability to create reproducible builds means that the artifacts from one build can be reliably reused in subsequent ones. In this context, the CI/CD system should be configured to efficiently store and retrieve these artifacts. This setup involves creating a robust artifact management system that can quickly provide the necessary build outputs, thus avoiding redundant builds and saving significant time.

## **Writing Efficient Starlark Code**

Efficient Starlark coding begins with an understanding of its operational semantics and how they interact with Bazel's build system. Unlike general-purpose programming languages, Starlark is designed with a specific focus on configuration, which necessitates a different approach to efficiency. One crucial aspect is the minimization of

computational complexity within Starlark functions. Developers should aim to reduce the amount of computation during the loading phase, as heavy computations can slow down the build process significantly.

Another aspect to consider is the reuse of results. Leveraging immutability in Starlark, developers should structure their code to reuse previously computed values rather than recalculating them. This practice not only speeds up the build process but also ensures consistency across builds. Moreover, efficient data structures that align with Starlark's strengths should be chosen. For instance, sets and dictionaries in Starlark are often more performance-efficient than lists, especially for membership tests and deduplication tasks.

## **Profiling and Optimizing Starlark Scripts**

Profiling Starlark scripts in the context of large-scale builds is a critical step in performance optimization. Bazel provides tools that enable developers to measure the execution time of Starlark rules and functions. By utilizing these tools, developers can pinpoint performance bottlenecks in their build configurations.

Once bottlenecks are identified, optimization strategies can be employed. This involves restructuring code for better performance, such as breaking down large functions into smaller, more manageable pieces or optimizing the order of operations to minimize the build graph's complexity. In some cases, it may also mean rethinking the approach to certain build tasks, perhaps by simplifying configurations or finding alternative methods that are more efficient in the context of Starlark and Bazel.

Moreover, it's essential to continuously monitor the performance impact of any changes in Starlark scripts. This



ongoing process of profiling, optimizing, and testing ensures that as the monorepo evolves, the build system remains as efficient as possible, effectively handling the scaling challenges posed by large codebases.

## **Structuring Monorepos in Version Control Systems**

The layout of a monorepo in a version control system significantly influences the build performance. A well-structured monorepo should logically separate components and services while maintaining a coherent overall architecture. This separation facilitates selective building and testing of components, reducing the build time by avoiding unnecessary recompilation of unrelated parts of the codebase. The key is to understand the interdependencies within the codebase and organize the repository in a way that reflects these relationships. By doing so, Bazel can more effectively determine which parts of the codebase need to be rebuilt in response to a particular set of changes, thus optimizing the build process.

Another aspect of structuring involves the granularity of the build targets. A monorepo should be organized with sufficiently granular build targets to allow Bazel to cache and reuse as much as possible. Despite this, too fine granularity can lead to an overwhelming number of targets, complicating the build graph and potentially slowing down the build process. Striking the right balance is key: sufficiently granular to leverage Bazel's caching effectively, but not so granular as to overcomplicate the build graph.

## **Managing Source Code Changes**

The manner in which changes are managed and integrated into a monorepo also affects build performance. Continuous

integration practices play a critical role here. Ideally, changes should be integrated frequently and in smaller increments. This practice not only facilitates easier code reviews and quicker identification of issues but also limits the scope of what needs to be rebuilt and tested with each change. By reducing the footprint of each change, the build system can run more efficiently, as it has fewer files to consider and can more effectively utilize cached results.

Moreover, the use of feature flags or conditional code paths can be an effective way to introduce changes without triggering widespread builds. By toggling features at runtime rather than compile time, developers can introduce changes in a more controlled and less disruptive manner. This approach allows for testing new features in isolation and gradually rolling them out, which can mitigate the impact on the overall build performance.

## **Conclusion**

In this chapter, we learned that managing large codebases, particularly in the context of monorepos, is a multifaceted challenge that demands a comprehensive understanding and strategic application of various advanced techniques. Throughout this chapter, we have explored a breadth of complex topics, ranging from the intricacies of handling internal and external dependencies, integrating third-party libraries, and employing advanced modularization strategies to the nuances of code sharing, organization, and readability. We delved into the complexities of managing inter-module dependencies, fostering an efficient collaborative environment, and refining branching, merging, and code review practices to minimize conflicts and build breakages. A significant emphasis was placed on performance optimization within monorepo builds, underscoring the importance of configuring remote build execution, leveraging caching strategies, and harnessing

Bazel's profiling tools to pinpoint and rectify inefficiencies. The chapter also highlighted the critical role of writing and optimizing Starlark code, structuring monorepos for optimal version control, and managing source code changes effectively. By navigating these challenges and employing best practices, developers and teams can effectively manage and scale large codebases, ensuring efficient, maintainable, and high-performing software development processes.

The next chapter will guide us through optimizing Docker builds, highlighting techniques to enhance efficiency and reduce resource usage. It will address the intricacies of API dependency management in a microservices architecture, ensuring seamless and scalable inter-service interactions. Furthermore, we'll explore the principles of software configuration management and the orchestration of microservices within a monorepo, emphasizing streamlined workflows and maintainability in complex development environments.

## **Recommended Reading**

For advanced readers familiar with Bazel, Monorepos, and Dependency Management in Bazel, the following online resources provide deeper insights into managing large codebases and scale.

### **Dealing with Large Monorepo Codebases**

**Mastering Monorepo:** Streamlining Development for Large Codebases - This article discusses the structure of a monorepo, its benefits, and how companies like Google, Microsoft, and Meta leverage monorepos for simplified version control, improved code reuse, and streamlined development.

<https://www.connectingpointstech.com/blog/mastering-monorepo-streamlining-development-for-large-codebases>

## **Advanced Strategies for Collaborative Environment Management**

**How do modern software teams manage vast amounts of the codebase?** - Cloud2Data's article provides strategies for efficient codebase management, focusing on version control systems, modularization, documentation, testing, refactoring, and the benefits of effective codebase management for productivity and collaboration. <https://cloud2data.com/how-do-modern-software-teams-manage-vast-amount-of-codebase/>

### **Performance Optimization for Monorepo Builds**

**Improving Large Monorepo Performance on GitHub** - This GitHub Blog post details Project Cyclops and the various improvements made to enhance monorepo push performance. It covers repository maintenance, `git repack` optimizations, removing artificial limits, and precomputing checksums to improve efficiency. <https://github.blog/2021-03-16-improving-large-monorepo-performance-on-github/>

Each of these resources offers advanced insights and practical solutions for managing large codebases and optimizing performance in a monorepo setting, especially valuable for teams already familiar with Bazel and related technologies.

# **CHAPTER 10**

## **Building and Deploying Services**

### **Introduction**

In this chapter, we will dive into the sophisticated realm of service construction and deployment within the context of Bazel, a tool already familiar to our readers. This chapter is tailored for those seeking to master advanced techniques and methodologies. We will explore the optimization of the Docker builds through a combination of fine-grained targets, the utilization of OCI (Open Container Initiative) images, and a layered approach to OCI image building and exporting. Special emphasis is placed on exploiting parallelization and caching to enhance build efficiency. The chapter then shifts focus to API dependency management, addressing the intricacies of managing transitive dependencies, devising robust API versioning strategies, and implementing automated updates. In the domain of software configuration management, we tackle dynamic configuration using Bazel, the management of environment-specific configurations, and the critical aspect of securing configuration data. Lastly, we navigate the complex landscape of orchestrating microservices within a Monorepo. Here, we cover advanced orchestration techniques, the strategic implementation of feature toggling, and the integration of monitoring tools coupled with auto-scaling strategies. Each section not only provides in-depth analysis but also includes practical examples, reinforcing the understanding of how these

advanced practices can be effectively applied in real-world scenarios.

## **Structure**

In this chapter, we will cover the following topics:

- Optimizing Docker Builds
- API Dependency Management in Microservices
- Software Configuration Management
- Orchestrating Microservices in a Monorepo

## **Optimizing Container Images Builds**

In the domain of Container images builds, leveraging Bazel's robust caching and parallelization capabilities stands as a crucial strategy for optimizing build times and efficiency. Bazel's cache can be effectively utilized to store and retrieve intermediate build outputs, thus significantly reducing redundant build steps in Container image creation. This approach becomes especially powerful in a continuous integration environment, where builds are frequent, and minimal changes occur between them.

To implement this, developers should structure their Bazel build files (`BUILD.bazel`) to define fine-grained targets. By doing so, changes in source code impact only the related Container layers, leading to efficient use of Bazel's cache. For instance, segregating application source code from dependencies in different Bazel targets can ensure that a change in the source code does not necessitate a re-fetching of all dependencies.

Moreover, Bazel's parallelization capabilities can be harnessed by defining multiple build targets that can be built or tested simultaneously. This is particularly effective in multi-service architectures, where each service can be built

in isolation and parallel, thereby reducing overall build times.

To demonstrate the implementation of the aforementioned strategies for optimizing OCI image builds using Bazel, let's consider a multi-service, multi-library Java project. In this coding example, we'll create a scenario with multiple services and libraries, demonstrating how to optimize Container builds using Bazel with a focus on caching, parallelization, and the use of OCI images. Open Container Initiative (OCI) images are advantageous as they provide a standardized and open specification for container image formats, ensuring consistency and interoperability across different container runtime environments. This example is available at examples that can be found in the GitHub repository at <https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel>, specifically within the /chapter-10/bazel\_optimizing\_docker\_builds folder.

Imagine a project with the following structure:

```
.
├── BUILD.bazel
├── MODULE.bazel
├── WORKSPACE
├── maven_install.json
├── libraries
│   └── lib1
│       ├── BUILD.bazel
│       └── src/main/java/com/lib1/Lib1.java
└── services
    ├── service1
    │   ├── BUILD.bazel
    │   └── src/main/java/com/service1
    │       ├── Main.java
    │       └── Service1.java
    └── service2
        ├── BUILD.bazel
```

```

└─ src/main/java/com/service2
    └─ Main.java
    └─ Service2.java

```

Each service and library has its own **BUILD.bazel** file.

```

load("@aspect_bazel_lib//lib:tar.bzl", "tar")
load("@rules_oci//oci:defs.bzl", "oci_image", "oci_tarball")

java_binary(
  name = "build",
  srcs = glob([
    "src/main/java/**/*.java",
  ]),
  main_class = "com.service1.Main",
  resources = glob(["src/main/resources/*.*"]),
  visibility = ["//visibility:public"],
  deps = [
    "//libraries/lib1:build",
  ],
)

tar(
  name = "build_layer",
  srcs = [":build_deploy.jar"],
)

oci_image(
  name = "build_oci_image",
  base = "@distroless_java",
  entrypoint = [
    "java",
    "-jar",
    "services/service1/build_deploy.jar",
  ],
  tars = [":build_layer"],
)

oci_tarball(
  name = "build_oci_image_tar",

```



```
    image = ":build_oci_image",  
    repo_tags = ["service1:latest"],  
  )
```

The following detailed explanation unpacks the intricacies of this setup, demonstrating how each component contributes to an optimized and streamlined build and deployment process for a Java service within a multi-service architecture.

## [Fine-grained Targets](#)

The `java_binary` rule defines a target named `build` for the Java service (`service1`). This encapsulates the Java service's build process, ensuring that changes in the service's Java source code only trigger rebuilds for this specific target. By segregating the Java service build from other parts of the project, Bazel's cache is used efficiently.

## [Use of OCI Images](#)

The `oci_image` rule is used to create an OCI-compliant container image (`build_oci_image`) for the Java service. The `base` attribute specifies the base image as `@distroless_java`, which is a minimal base image optimized for Java applications. By using OCI (Open Container Initiative) standards, the container image ensures compatibility and consistency across different container runtimes.

## [Layered Approach](#)

The `tar` rule creates a tarball (`build_layer`) containing the deployable JAR file (`build_deploy.jar`). This layering approach allows for efficient use of Container layer caching, as changes in the application code result in changes only in this specific layer, rather than the entire image.

## [OCI Image Building and Exporting](#)

The ``oci_image`` rule builds the container image, incorporating the JAR file and specifying the entrypoint for running the Java application. This step integrates the application with the container runtime environment.

The ``oci_tarball`` rule exports the OCI image (``build_oci_image``) as a tarball (``build_oci_image_tar``), facilitating easy distribution and deployment of the container image. The ``repo_tags`` attribute tags the image, making it straightforward to identify and deploy the correct version.

## [Parallelization and Caching](#)

**Parallelization and Efficiency:** Bazel's ability to build targets in parallel is leveraged here, where dependencies are managed, and parallelizable tasks are executed concurrently, reducing overall build time.

Bazel's caching is effectively used to store intermediate outputs of the build process. When the Java service or its dependencies are rebuilt, Bazel retrieves the cached outputs if the source code and dependencies haven't changed, minimizing redundant build steps.

## [Running the Example](#)

To run the example provided in the advanced Bazel setup for building and deploying a Java-based service, follow these steps. These commands demonstrate how to create an OCI (Open Container Initiative) compliant Container image from the Bazel build, load it into Container, and then run it as a container.

### 1. Building the OCI Image as a Tar File with Bazel:

- Use the command `bazel build //services/service1:build_oci_image_tar`.

- This command instructs Bazel to build the `build_oci_image_tar` target located in the `service1` directory under `services`.
- The target is configured to create an OCI-compliant image of the service, packaged as a tar file. Bazel performs the necessary steps to compile the Java application, package it into a JAR file, build the OCI image, and finally package the image into a tarball.

## 2. Loading the OCI Image into Docker:

- After the build process is complete, the OCI image tar file is located at `bazel-bin/services/service1/build_oci_image_tar/tarball.tar`.
- To load this image into Docker, use the command `docker load -i bazel-bin/services/service1/build_oci_image_tar/tarball.tar`.
- This command tells Docker to load the image from the specified tar file. Docker unpacks the tarball and stores the image locally, making it available for running as a container.

## 3. Running the Container:

- Once the image is loaded into Docker, you can start a container from this image using the command `docker run --name s1 service1:latest yourname`.
- Here, `docker run` is the command to run a new container.
- `--name s1` assigns the name `s1` to the running container for easy reference.
- `service1:latest` specifies the image to use for the container, in this case, the latest version of the `service1` image.
- Finally, `yourname` is passed as an argument to the container;

#### 4. To push the Container:

- To push the image to a registry, we use the `container_push` rule. Following is an example of how to define this rule in your **BUILD** file:

```
load("@io_bazel_rules_docker//container:container.bzl"  
    , "container_push")  
container_push(  
    name = "push_hello_world_image",  
    image = ":hello_world_image",  
    format = "Docker",  
    registry = "gcr.io",  
    repository = "my_project/hello_world",  
    tag = "latest",  
)
```

- This rule pushes the `hello_world_image` to the `gcr.io/my_project/hello_world` repository with the latest tag.
- Finally, to build and push your image, run the following command in your terminal:

```
bazel run //path/to/directory:push_hello_world_image
```

This command triggers Bazel to build the `hello_world_image` and then push it to the specified container registry.

In summary, this Bazel setup for a Java service demonstrates an advanced approach to building and containerizing applications, emphasizing the efficient use of fine-grained targets, OCI standards, layered Container builds, and the powerful caching and parallelization capabilities of Bazel.

## [API Dependency Management in Microservices](#)

We will explore techniques and best practices in the realm of API dependency management. This includes managing transitive dependencies with precision, implementing robust API versioning strategies to ensure seamless and backwards-compatible transitions, and harnessing the power of automated tools to keep API dependencies updated and secure.

## Managing Transitive Dependencies

The management of transitive dependencies in a microservices architecture is a complex task, primarily due to the interconnected nature of microservices and the varied dependencies they may have. When dealing with Bazel operating within a monorepo setting, it is crucial to have a robust strategy for handling these dependencies, especially considering versioning and conflict resolution. One effective technique is the use of Bazel's dependency rules to explicitly define and isolate dependencies. This can be done through **WORKSPACE** and **BUILD** files, where dependencies are declared and versions are specified.

For instance, consider a scenario where Service A depends on Library X and Service B also depends on Library X but requires a different version. In such cases, Bazel can be configured to handle these version conflicts. Utilizing ``select()`` statements in **BUILD** files allows the build system to choose the appropriate version of the library based on the context of the build target.

```
java_library(  
  name = "service_a_deps",  
  deps = select({  
    "@//conditions:service_a":["@lib_x//:v1"],  
    "//conditions:default": [],  
  }),  
)
```

```

java_library(
  name = "service_b_deps",
  deps = select({
    "@//conditions:service_b":["@lib_x//:v2"],
    "//conditions:default": [],
  }),
)

```

In this example, `select()` is used to specify which version of Library X should be used depending on the service being built. This ensures that each service gets the correct version of the library, thereby managing transitive dependencies effectively.

## [API Versioning Strategies](#)

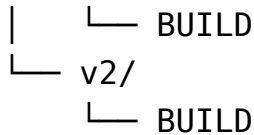
API versioning is pivotal in ensuring backward compatibility and facilitating smooth transitions between different versions of services. When working with a monorepository structure setup, API versioning can be handled through a combination of Bazel's build rules and thoughtful architectural design. One approach is to use semantic versioning for APIs, which involves modifying the major version number when introducing breaking changes, the minor version for new features, and the patch version for bug fixes.

To enforce this in the framework of a monorepository system, directories and build targets can be named according to their API versions. For instance, having separate directories for v1 and v2 of an API with corresponding Bazel build targets ensures that changes to one version do not inadvertently affect another. This separation can be mirrored in the service's endpoints, allowing clients to choose which API version to interact with.

```

my_service/
├─ v1/

```



Each BUILD file within these directories can define the build process for that specific version of the API, ensuring isolation and clarity in version management.

## [Automated Dependency Updates](#)

Keeping dependencies up-to-date and secure is a critical aspect of managing microservices. Bazel can be integrated with automated tools to ensure that dependencies are consistently updated. One such tool is Renovate or Dependabot, which can be configured to scan your Monorepo and create pull requests with updated dependency versions.

In conjunction with these tools, Bazel rules can be written to validate these updates. For example, you can create a Bazel rule that runs your test suite against the updated dependencies. This rule can be triggered automatically when a new pull request is created by the dependency update tool.

```
genrule(  
  name = "dependency_update_test",  
  srcs = glob(["**"]),  
  cmd = "$(location //my_service:test_suite) && $(location  
  @//tools:dependency_validator)",  
  tools = ["//my_service:test_suite",  
  "@//tools:dependency_validator"],  
)
```

In this code snippet, the `genrule` is used to run the test suite defined in `//my_service:test_suite` and a hypothetical dependency validator tool whenever there is a change in the source files, which would include dependency updates. This setup ensures that any updates to dependencies are

automatically tested, helping maintain the stability and security of your microservices.

## Software Configuration Management

### **Dynamic configuration with Bazel**

In the field of software configuration management, particularly within a microservice architecture, Bazel stands out for its ability to handle dynamic configurations. The key advantage lies in Bazel's capability to manage and integrate diverse configurations that evolve over time, especially in a multi-service environment. This dynamism is achieved through the strategic use of Bazel's ``select`` function, allowing configurations to adapt based on specified conditions. For instance, consider a scenario where different microservices need varied logging levels. In Bazel, one can define a ``logging_level`` configuration, and then use the ``select`` function to dynamically assign different values based on the target microservice:

```
config_setting(  
  name = "debug_logging",  
  values = {"logging_level": "debug"},  
)  
java_binary(  
  name = "service_a",  
  srcs = ["ServiceA.java"],  
  deps = ["//lib:logging"],  
  java_opt = select({  
    ":debug_logging": ["-Dlogging.level=DEBUG"],  
    "//conditions:default": [],  
  }),  
)
```

In this example, ``ServiceA.java`` receives a debug logging level based on the configuration setting. Such dynamic



configuration enhances flexibility and reduces the rigidity often encountered in managing microservices.

## **Managing environment-specific configurations**

The challenge of handling different configurations for development, testing, and production environments is adeptly addressed by Bazel. The principle here is to define environment-specific build targets or use Bazel's `--config` option to switch between different build configurations. This method is more advanced and scalable than maintaining separate configuration files for each environment. By integrating these configurations directly into the build system, developers can seamlessly switch contexts and ensure that the right configurations are applied for each environment.

For instance:

```
bazel build //my/service:service_binary --config=dev
bazel build //my/service:service_binary --config=prod
```

In the preceding commands, `--config=dev` and `--config=prod` refer to different sets of build options defined in the `.bazelrc` file. These options can include specific compiler flags, build variables, or anything else that needs to vary between environments.

## **Securing configuration data**

Securing sensitive configuration data within a Monorepo is a critical aspect of software configuration management. Bazel aids in this by isolating sensitive data and ensuring that it is not inadvertently included in the build artifacts. One advanced strategy involves using Bazel's capabilities to reference external files or environment variables at build time, rather than hard-coding sensitive data. This method ensures that sensitive information, such as API keys or database credentials, remains outside the codebase and is only referenced when necessary. For example:

```
java_binary(  
  name = "service_b",  
  srcs = ["ServiceB.java"],  
  data = ["///:sensitive_config"],  
  jvm_flags = ["-Dconfig.file=$(location :sensitive_config)"],  
)
```

In this snippet, `\///:sensitive_config`` is a reference to an external file containing sensitive configuration data. The file is not included in the source but is referenced at runtime, ensuring that sensitive data is securely managed and not exposed within the Monorepo.

SOPS (*"Secrets OPERations"*) is an open-source tool that provides secure, flexible, and user-friendly ways to manage secrets. It enables the encryption and decryption of files, ensuring that sensitive information like passwords, API keys, or certificates can be securely stored in version control systems without exposing them in plain text. This is crucial for maintaining the security and integrity of software projects, especially in a collaborative environment. Integrating sops with Bazel can enhance the security of your build and deployment processes by ensuring that secrets are only accessible to authorized personnel and systems. In a Bazel project, you can use sops to encrypt secret files and store them alongside your source code. When Bazel runs a build or deployment task, it can be configured to automatically decrypt these files using sops, injecting the necessary secrets into the environment or making them available to the application at runtime. This process can be automated and tightly controlled, ensuring that secrets are handled securely throughout the build and deployment pipeline. By integrating sops with Bazel, developers can achieve a seamless and secure workflow, where the management of secrets is both efficient and robust, aligning with best practices in software development and deployment.

In conclusion, advanced software configuration management with Bazel in a microservice architecture demands a deep understanding of Bazel's capabilities to handle dynamic configurations, manage environment-specific settings, and secure sensitive data. The examples provided demonstrate Bazel's robustness in addressing these challenges, ensuring that configurations are both flexible and secure, aligning with the demands of modern software development practices.

## **Orchestrating Microservices in a Monorepo**

Orchestrating microservices within a Monorepo using Bazel involves sophisticated techniques that go beyond basic setup and deployment. This section delves into the intricate aspects of generating Helm charts to model microservice dependencies, the implementation of feature toggling, and the integration of monitoring and auto-scaling strategies.

## **Advanced Microservice Orchestration Techniques**

With the monorepo approach, managing the relationships and dependencies between microservices is crucial for a streamlined workflow. Bazel, with its fine-grained dependency management, can be extended to generate Helm charts, which serve as templates for deploying applications in a Kubernetes environment. By automating the generation of Helm charts, teams can ensure that the deployment configuration stays in sync with the codebase.

For instance, consider a setup where a Bazel rule generates a Helm chart based on the **BUILD** files in the repository. This rule could look for specific tags or attributes in the **BUILD** files

to determine the deployment properties of each service. Here's an example of how such a rule might be defined:

```
# Define a rule in a Bazel BUILD file to generate a Helm chart
helm_chart(
  name = "my_service_chart",
  srcs = glob(["**/*.yaml"]),
  deps = [":my_service"],
  chart_name = "my-service",
  version = "1.0.0",
)
```

In this code, `helm_chart` is a custom rule that takes the source YAML files, dependencies, and chart details to produce a Helm chart for `my_service`. This approach allows developers to define deployment configurations alongside their code, reducing the risk of mismatches between the service code and its deployment specifications.

## Feature Toggling

Feature toggling is an advanced technique that allows teams to dynamically enable or disable features in a live environment without redeploying the application. In a Bazel-managed Monorepo, feature toggles can be implemented as configurable build targets. This allows for the inclusion or exclusion of certain features at build time based on the specified configuration.

As an example, a feature toggle could be implemented in Bazel as follows:

```
# Example of a feature toggle in a Bazel BUILD file
config_setting(
  name = "experimental_feature",
  values = {"define": "experimental_feature=true"},
)
cc_binary(
  name = "my_service",
```

```

srcs = ["main.cc"],
deps = select({
  ":experimental_feature":
  ["@experimental_feature//path:lib"],
  "//conditions:default": [],
}),
)

```

In this snippet, `config_setting` creates a configuration option for the experimental feature. The `cc_binary` rule then uses `select` to conditionally include the experimental feature's dependencies based on the configuration.

## [Monitoring and Scaling Microservices](#)

Integrating monitoring tools and implementing auto-scaling strategies are essential for maintaining the performance and reliability of microservices. In a Bazel-powered Monorepo, one can leverage Bazel's ability to handle external dependencies and tool integrations to set up monitoring and scaling mechanisms.

For instance, Bazel can be used to automate the deployment of monitoring agents alongside the services. The configuration for these agents can be defined in the Monorepo and linked to the service deployments. Additionally, Bazel can interact with cloud provider APIs or Kubernetes to implement auto-scaling rules based on the monitoring data.

A simplified example of integrating a monitoring tool might involve defining a Bazel rule that deploys a monitoring agent:

```

# Bazel rule to deploy a monitoring agent
k8s_object(
  name = "monitoring_agent_deployment",
  template = ":monitoring_agent_template.yaml",
  substitutions = {"{{image}}": "@monitoring_agent//:image"},
)

```

)

In this example, ``k8s_object`` is a rule that takes a Kubernetes object template (in this case, a deployment template for a monitoring agent) and applies substitutions, such as the Docker image for the agent.

In conclusion, orchestrating microservices in the setting of a monorepository with Bazel requires a deep integration of deployment, feature management, and operational tools. By leveraging Bazel's capabilities for generating deployment configurations, managing feature toggles, and integrating with monitoring and scaling tools, teams can achieve a high degree of automation and consistency in their microservice architecture. This not only enhances operational efficiency but also ensures that the services are robust, scalable, and maintainable.

## **Conclusion**

In this chapter, we explored a range of advanced topics essential for building and deploying services using Bazel within a Monorepo environment. Starting with the optimization of Docker builds, we delved into fine-grained targets and the utilization of OCI images to achieve a layered approach in containerization. We addressed the nuances of OCI image building and exporting, highlighting how Bazel's capabilities in parallelization and caching can significantly enhance build efficiency. The importance of understanding and managing transitive dependencies in microservices was emphasized, alongside strategies for API versioning and the necessity for automated dependency updates to maintain robust and secure systems. In the world of software configuration management, we focused on dynamic configurations and the challenges of managing environment-specific configurations, not overlooking the critical aspect of securing configuration data. Finally,

orchestrating microservices within a Monorepo was discussed in depth, covering advanced techniques such as generating Helm charts, implementing feature toggling, and integrating monitoring tools for scaling microservices effectively.

The integration of these advanced techniques and practices underscores the power and flexibility of Bazel in handling the complexities of modern service-oriented architectures. By leveraging Bazel's strengths considering a monorepository, developers and teams can achieve more streamlined, efficient, and reliable build and deployment processes. This not only fosters a more productive development environment but also contributes to the stability and scalability of the services being deployed. As we move forward, the continuous evolution of these practices and the adaptation to emerging technologies will remain paramount in maintaining the efficacy and relevance of these methodologies in the dynamic landscape of software development.

In the next chapter, we will focus on enhancing Bazel's performance, starting with how to monitor and interpret build metrics. We will delve into advanced remote caching and execution to improve build efficiency, explore targeted debugging techniques for Bazel rules, and conclude with performance-tuning strategies for large-scale Monorepos. This chapter aims to equip you with practical skills to optimize your Bazel builds, ensuring faster, more efficient development workflows.

## **Recommended Reading**

To further enhance your understanding and skills in managing large codebases and scaling with Bazel, here are some recommended online resources that align with the advanced topics covered as you read through this chapter:

## **Optimizing Docker Builds with Bazel**

Bazel's Docker Rules: Advanced Usage: An in-depth guide on optimizing Docker builds using Bazel's Docker rules. - [https://github.com/bazelbuild/rules\\_docker](https://github.com/bazelbuild/rules_docker)

## **Understanding Fine-Grained Targets in Bazel**

Bazel's Target Granularity for Large Codebases: Detailed documentation on the concept and benefits of fine-grained targets in Bazel. - <https://docs.bazel.build/versions/main/build-ref.html#target>

## **Use of OCI Images and Layered Approach**

Working with OCI Images in Bazel: A comprehensive blog post covering the use of OCI images in Bazel and implementing a layered approach. - <https://blog.bazel.build/2019/03/19/docker-rules.html>

## **OCI Image Building and Exporting Techniques**

Building and Exporting OCI Images with Bazel: A series of articles providing insights into effective OCI image building and exporting strategies with Bazel. - <https://www.ianlewis.org/en/tag/bazel>

## **Strategies for Parallelization and Caching in Bazel**

Improving Build Performance with Bazel: Official documentation on maximizing build performance through parallelization and caching. - <https://docs.bazel.build/versions/main/build-performance.html>

## **Advanced API Dependency Management in Microservices**

Managing Dependencies in Microservices: Martin Fowler's article on the complexities and best practices for managing dependencies in microservices. - <https://martinfowler.com/articles/microservice-testing/#conclusion-contract>



## **Dynamic Configuration Management with Bazel**

Dynamic Configuration in Bazel: A blog post exploring dynamic configuration techniques in large codebases, using Envoy as an example. - <https://blog.envoyproxy.io/dynamic-configuration-in-envoy-7f8f498e663a>

## **Securing Configuration Data in a Monorepo**

Secure Configuration Management in Monorepos: A Google Cloud blog discussing best practices for securing sensitive configuration data in large-scale projects. - <https://cloud.google.com/blog/products/identity-security/keeping-your-secrets-secret>

## **Advanced Microservice Orchestration Techniques**

Helm Chart Generation with Bazel: An article providing insights into generating Helm charts for microservices using Bazel. - <https://helm.sh/blog/helm-3-preview-charting-our-future-part-2-helm/>

## **Implementing Feature Toggling in Large Codebases**

Feature Toggles (also known as Feature Flags): Martin Fowler's comprehensive guide on implementing feature toggles in complex software systems. - <https://martinfowler.com/articles/feature-toggles.html>

## **Monitoring and Scaling Microservices with Bazel**

Scaling Microservices with Bazel: An InfoQ article discussing strategies for monitoring and scaling microservices in a Bazel environment. - <https://www.infoq.com/articles/microservices-bazel/>

These resources provide a deeper dive into the specific topics of managing large codebases and scaling, complementing the information covered through these pages. They offer practical insights and case studies that can help solidify your understanding and application of these advanced concepts.

# **CHAPTER 11**

## **Monitoring and Debugging Bazel**

### **Introduction**

In this chapter, we will deep dive into the nuances of harnessing Bazel's full potential for building and deploying complex software systems efficiently. With an emphasis on advanced users familiar with Bazel, monorepos, and its dependency management, we explore sophisticated topics ranging from monitoring Bazel performance to the intricacies of debugging custom Bazel rules. We will address the critical aspects of interpreting and visualizing profiling data to identify performance bottlenecks, alongside the utilization of custom scripts and command-line tools for in-depth analysis. The chapter also covers the conversion of Bazel's Build Event Protocol (BEP) output into formats compatible with systems like Prometheus, facilitating seamless integration into existing monitoring frameworks. We will delve into the development of automated tests for performance regression, benchmark implementation, and the continuous monitoring of performance metrics. A significant focus is placed on advanced strategies for remote caching and execution, including monitoring cache hit rates, diagnosing cache inefficiencies, and optimizing remote execution. Additionally, we will provide a comprehensive guide on debugging techniques for Bazel rules, leveraging Starlark's debugging capabilities, and implementing best practices for logging and error handling. Lastly, the chapter concludes with insights into performance

tuning for large-scale monorepos, offering advanced configuration settings, strategies for scalability, and case studies that showcase performance improvements in complex projects, embodying the essence of efficient and effective service building and deployment with Bazel.

## Structure

In this chapter, we will cover the following topics:

- Monitoring Bazel Performance
- Advanced Usage of Remote Caching and Execution
- Debugging Techniques for Bazel Rules
- Performance Tuning for Large-scale Monorepos

## Monitoring Bazel Performance

The Bazel build system offers a powerful tool for performance monitoring through the `--profile` flag. This flag generates a detailed profile of the build process, providing invaluable insights into the execution time and resource utilization of various build tasks. To leverage this tool, developers must append `--profile=<file-path>` to their Bazel build commands, where `<file-path>` is the desired location for the output profile. For instance, `bazel build //my:target --profile=/tmp/my_build.profile` will initiate a build of `//my:target` and generate a profile at `/tmp/my_build.profile`.

The generated profile is a binary file, which can be analyzed using Bazel's built-in analysis tool, `bazel analyze-profile`. This command parses the binary profile and produces a human-readable report. As an illustration, executing `bazel analyze-profile /tmp/my_build.profile` will display a comprehensive breakdown of the build process, highlighting the time spent in various activities such as fetching dependencies, executing build rules, and more.

## **Interpreting Profiling Data for Performance Bottlenecks**

Interpreting the data from Bazel's profiling tools requires an understanding of the various phases and activities involved in a build process. The profiling report segments the build into phases like initialization, analysis, execution, and completion. Within these phases, it details the time spent on individual actions, such as compiling source files, running tests, or fetching remote dependencies.

The key to identifying performance bottlenecks lies in focusing on the longest-running tasks. To give you an idea, if the report indicates that a significant amount of time is spent in the analysis phase, it may suggest inefficiencies in how dependencies are structured or in the complexity of the build rules. Similarly, extended execution times for specific build targets could indicate issues with those targets' rules or source code. By pinpointing the phases and actions consuming the most time, developers can target their optimization efforts more effectively.

## **Utilizing Custom Scripts to Parse and Analyze Profile Data**

While ``bazel analyze-profile`` provides a good overview, developers dealing with complex builds may require deeper analysis. For this purpose, custom scripts can be employed to parse and analyze profile data. These scripts can convert the binary profile data into other formats, like JSON or CSV, facilitating more detailed analysis using data processing tools.

For example, a Python script could be written to parse the binary profile and extract key metrics such as the execution time of each target or the total time spent fetching remote resources. This script could then aggregate this data to

identify common patterns or outliers in the build process. For instance:

```
import bazel_profiler
# Parse the Bazel profile
profile_data =
bazel_profiler.parse_profile('/tmp/my_build.profile')
# Analyze the data
longest_tasks =
bazel_profiler.find_longest_tasks(profile_data)
resource_usage =
bazel_profiler.analyze_resource_usage(profile_data)
# Output the findings
bazel_profiler.report_findings(longest_tasks, resource_usage)
```

In this example, `bazel_profiler` could be a custom module developed to handle the specifics of Bazel's profile format. By extracting and analyzing specific metrics, developers can gain a deeper understanding of their build's performance characteristics and identify the targeted areas for optimization.

## [Utilizing Command Line Tools to Analyze Profile Data](#)

Consider the example provided in the GitHub repository located at <https://github.com/OrangeAVA/Building-Large-Scale-Apps-with-Monorepo-and-Bazel>, particularly in the `"/chapter-6/bazel_android_kotlin"` section. When you execute the following command, Bazel will compile the Android application and simultaneously collect all the profiling data, saving it in the file named `myprofile.out`.

```
bazel build --profile=myprofile.out //app
```

To analyze this file, you can run the following command:

```
bazel analyze-profile myprofile.out
```

The output generated from this analysis will appear as follows:

INF0: Profile created on 2023-12-27T08:08:57.735955Z, build ID:...

=== PHASE SUMMARY INFORMATION ===

Total launch phase time	1.353
s 0.56%	
Total init phase time	12.313
s 5.08%	
Total target pattern evaluation phase time	15.518
s 6.40%	
Total interleaved loading-and-analysis phase time	88.288 s
36.40%	
Total preparation phase time	0.014
s 0.01%	
Total execution phase time	125.022 s
51.54%	
Total finish phase time	0.046
s 0.02%	

-----  
-----

Total run time	242.557
s 100.00%	

Critical path (28.286 s):

Time	Percentage	Description
39.2 ms	0.14%	action 'Creating source manifest for @bazel_too
1.053 s	3.72%	action 'Creating runfiles tree bazel-out/darwin
5.63 ms	0.02%	runfiles for @bazel_tools//tools/android aar_re
7.566 s	26.75%	action 'Extracting AAR Resources for @rules_jvm

4.473 s	15.82%	action 'Compiling Android resources for @rules_
1.617 s	5.72%	action 'Merging compiled Android resources for
2.891 s	10.22%	action 'Linking static android resource library
4.342 s	15.35%	action 'Processing Android resources for //app
1.645 s	5.82%	action 'Generating R Classes for //app app'
111 ms	0.39%	action 'Desugaring app/app_resources.jar for An
317 ms	1.12%	action 'Building deploy jar bazel-out/darwin_ar
3.065 s	10.84%	action 'Extracting Java resources from deploy j
75.2 ms	0.27%	action 'Generating unsigned apk'
475 ms	1.68%	action 'Zipaligning apk'
611 ms	2.16%	action 'Signing apk'

Based on this data, our focus can shift towards comprehending the duration spent in each phase. The most time-intensive phases are the loading-and-analysis phase and the execution phase. However, the extended duration of the former is attributable to it being an initial build, which will be leveraged in subsequent executions.

The following section details the duration of each action on the critical path. Actions such as “**Extracting AAR Resources for @rules\_jvm...**”, “**Compiling Android resources for @rules\_...**”, “**Processing Android resources for //app ...**”, and “**Extracting Java resources from deploy j...**” are identified as the most time-consuming. Nevertheless, similar to earlier observations, the majority of this work will be leveraged in future builds, allowing for reusability.

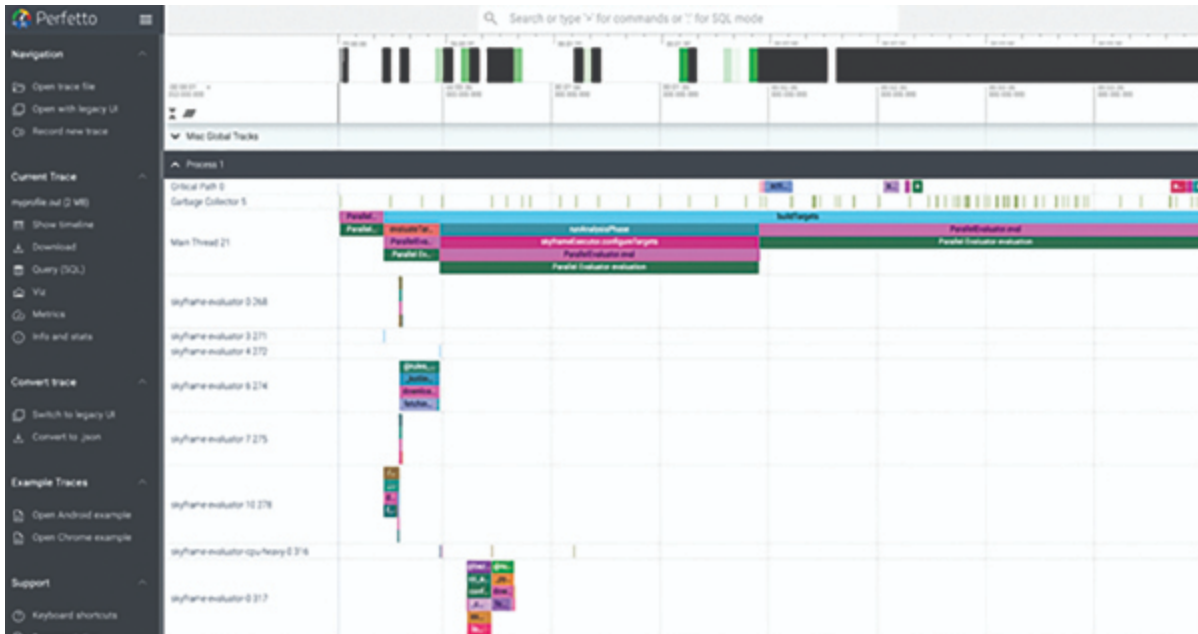
## [Visualizing Profiling Data](#)

To visualize the profile, you have the option to use either Chrome's tracing feature or other tools for analysis and post-processing. For the Chrome tracing method, navigate to `chrome://tracing` in a Chrome browser tab, select "Load", and choose your profile file, which may be compressed.

Alternatively, a more advanced tool is Perfetto, accessible at <https://ui.perfetto.dev/>. Perfetto is a comprehensive, open-source tool for performance monitoring and tracing that is designed to provide detailed insights into system behavior. Initially developed by Google and used in the Android ecosystem, it has grown to support a wide range of platforms, including Linux and Chrome OS. Perfetto's capabilities extend beyond traditional profiling tools by enabling high-resolution, system-wide tracing and logging. It captures a variety of data, such as CPU scheduling, system calls, and memory usage, offering a granular view of system performance. This tool is particularly useful for developers and system administrators looking to diagnose complex performance issues, optimize system and application performance, and understand intricate system interactions in real-time and post hoc analyses.

Here, you can upload your `myprofile.out` file by selecting the "Open Trace File" option from the left-hand menu.





**Figure 11.1:** Screen capture of the Perfetto web app

By scrolling vertically, you can navigate through all the executed threads and tasks. Clicking the boxes allows you to delve into the specifics of resource usage and metrics. At the bottom, general metrics detailing the total resource consumption by Bazel over time are displayed.



**Figure 11.2:** Screen capture of the Perfetto web app

The profiles produced by Bazel can be quite detailed, so don't hesitate to reach out on the Bazel mailing list for assistance in understanding them. Furthermore, for more information on this topic, you can explore the documentation regarding profiling.

## [Understanding and Utilizing Bazel's BEP](#)

Bazel's Build Event Protocol (BEP) is a cornerstone for integrating advanced monitoring tools. BEP provides a stream of data about the build process, encompassing information about the actions performed, targets built, and the outcomes of these actions. This protocol outputs a series of protocol buffer messages, which can be consumed by external tools to monitor and analyze build processes in real-time.

The BEP messages are structured and highly detailed, offering insights into every aspect of the build process. They include information about build targets, individual test results, the status of build actions, and even filesystem events. This granular level of detail makes BEP an invaluable resource for developing sophisticated monitoring solutions.

To integrate BEP with monitoring tools like Prometheus, it's essential to convert the BEP output into a format that Prometheus can understand. Prometheus primarily works with time-series data, requiring the transformation of BEP's protocol buffer messages into a compatible structure.

## **Converting BEP Output to Prometheus-friendly Format**

The conversion process involves parsing the BEP messages and extracting relevant metrics, which are then formatted into a time-series data format suitable for Prometheus. To achieve this, you can write a script or a small service that subscribes to the BEP event stream and processes the events as they occur.

Here's a simplified example in Python, demonstrating how to parse BEP messages and convert them into a Prometheus-friendly format:

```
import bazel.bep.proto.build_event_stream_pb2 as bes
```

```

import prometheus_client
# Initialize Prometheus metrics
build_duration_metric =
prometheus_client.Summary('bazel_build_duration_seconds',
'Duration of Bazel build')
def process_bep_event(bep_message):
    event = bes.BuildEvent()
    event.ParseFromString(bep_message)
    if event.HasField('completed'):
        target = event.completed.label
        duration = event.completed.duration_millis / 1000.0
        build_duration_metric.observe(duration)
        print(f"Build completed for {target} in {duration} seconds")
def main():
# Assuming 'bep_stream' is a stream of BEP messages
for bep_message in bep_stream:
    process_bep_event(bep_message)
if __name__ == "__main__":
    main()

```

In this example, the script listens to a stream of BEP messages, parses each message, and checks if it represents a completed build event. If so, it calculates the duration of the build and records it as a Prometheus observation. You can extend this script to capture more metrics and handle different types of BEP events based on your monitoring needs.

Integrating Bazel with Prometheus or similar tools via BEP allows teams to gain real-time insights into their build processes. These insights can drive optimizations, help in identifying bottlenecks, and ensure the efficiency and reliability of the build system in large-scale monorepo environments. By leveraging the detailed data provided by BEP and converting it into actionable metrics, development teams can significantly enhance their monitoring

capabilities and maintain high performance in their build pipelines.

## **Developing Automated Tests for Performance Regressions**

BEP provides a detailed stream of events related to the build process, which can be harnessed to detect performance regressions. By scripting against these events, developers can create a system that automatically flags deviations in build times or resource usage, indicating potential performance issues.

For instance, consider a scenario where a developer integrates a Python script that parses BEP's JSON output. This script can be designed to compare the execution time of each target against a historical average. A simple script integrated with a system that maintains historical performance data for each target enables a direct comparison to identify significant deviations.

## **Implementing Benchmarks and Performance Baselines**

Benchmarking in Bazel involves establishing a set of performance baselines against which current build metrics can be compared. These baselines act as a reference point, ensuring that any performance changes are immediately noticeable. Setting up benchmarks involves running a series of controlled builds and capturing key metrics like build time, CPU usage, and memory consumption. These metrics are then stored as the baseline for future comparisons.

To automate this process, developers can utilize Bazel's ability to output detailed metrics. To exemplify, using the `--profile` flag with Bazel generates a profile file that can be analyzed to extract necessary performance metrics. A script

can be written to parse this file, extract the relevant data, and store it as a baseline. Subsequent builds can be compared against this baseline to identify performance regressions.

## **Continuous Monitoring of Performance Metrics**

Continuous monitoring is essential for maintaining the performance integrity of a project. Integrating performance regression testing into a continuous integration (CI) pipeline ensures that any code change is automatically evaluated for its impact on performance. This approach enables real-time detection of performance issues, allowing for immediate remediation.

In practice, this involves configuring the CI system to run the Bazel builds with performance tracking enabled, followed by executing the scripts that analyze the BEP output and compare it against the established benchmarks. If a performance regression is detected, the CI pipeline can be configured to alert the development team, or even fail the build as a cautionary measure.

One example is a CI configuration snippet that might include steps like:

1. Run Bazel build with BEP and profiling enabled.
2. Execute a script that parses the BEP output and profiling data.
3. Compare current build metrics against performance baselines.
4. Alert the team if a regression is detected.

This integration ensures that performance is continually monitored and maintained throughout the development

lifecycle, making performance regression testing an integral part of the build process.

## **Advanced Usage of Remote Caching and Execution**

This section delves into advanced strategies, techniques, and insights for monitoring, diagnosing, and optimizing these aspects.

### **Monitoring Remote Cache Hit Rates**

Monitoring remote cache hit rates is a critical aspect of optimizing build times. High cache hit rates typically indicate the effective reuse of artifacts, reducing redundant computation. To monitor this, developers can utilize Bazel's in-built logging facilities. By setting the `--bes_backend` flag and specifying a build event service, you can collect detailed information about cache hits and misses. For instance:

```
bazel build //my:target --bes_backend=grpc://my-bes-server:1985
```

This command sends build events to the specified server, which can then be analyzed to compute cache hit rates. For a more granular analysis, developers can employ the `--experimental_remote_cache_log` flag, which produces a detailed log of cache-related activities, enabling precise identification of cache hits and misses. As an experimental feature, it is recommended to exercise caution when considering its utilization in production environments or to remain vigilant regarding potential modifications during forthcoming Bazel version updates.

## Diagnosing Cache Misses and Inefficiencies

Diagnosing cache misses involves understanding why certain actions were not cacheable or why cached artifacts were not reused. One common cause is non-deterministic outputs or varying inputs, which can be identified by comparing hashes of inputs and outputs of similar actions across different builds. Bazel's `aqery` tool is instrumental in this analysis. It allows developers to query the action graph and understand the dependencies and inputs of actions. A typical use case might be:

```
bazel aquery 'mnemonic("CppCompile", //my:target)' --  
output=text
```

This command retrieves information about C++ compilation actions for the specified target, which can then be analyzed to detect variations leading to cache misses.

## Optimizing Remote Execution Performance

Optimizing remote execution performance requires a holistic approach, considering both the build environment and the specificities of the build process. One advanced technique is the fine-tuning of remote execution parameters, such as the number of remote workers or the size of the input and output files. Adjusting these parameters can be done using Bazel's flags like `--remote_max_connections` or `--remote_timeout`. For instance:

```
bazel build //my:target --remote_max_connections=100 --  
remote_timeout=300
```

This command sets the max number of concurrent connections to remote workers and adjusts the timeout setting, potentially improving performance in environments

with high network latency or large numbers of parallelizable tasks.

Another facet of optimization is the strategic use of remote execution for only those parts of the build process that benefit most from it, such as large compilations or tests. This can be controlled using target-level attributes or by selectively enabling remote execution for certain builds with the `--config=remote` flag in the `.bazelrc` file.

## [Debugging Techniques for Bazel Rules](#)

Debugging custom Bazel rules requires a methodical approach, leveraging the capabilities of Starlark and adhering to best practices in logging and error handling. This section delves into these aspects, providing advanced insights for developers experienced with Bazel's ecosystem.

## [Step-by-step Approach for Debugging Custom Bazel Rules](#)

When debugging custom Bazel rules, the first step is to isolate the issue. Begin by constructing a minimal reproducible example that demonstrates the problem. This process often involves stripping down the rule or build to the simplest form where the issue still manifests. Once isolated, use Bazel's built-in debugging flags, such as `--subcommands` to understand the commands executed by Bazel. This can be instrumental in pinpointing where the rule behaves unexpectedly.

For instance, if a rule is failing to compile a set of source files, use the command:

```
bazel build //my/target:rule --subcommands
```



This will display each command Bazel executes, allowing you to trace back to the specific point of failure.

## **Utilizing Starlark's Debugging Capabilities**

Starlark, the language used to write Bazel rules and macros, does not support traditional debugging tools like breakpoints or step-through execution. However, it offers robust introspection capabilities. Use `print` statements effectively to trace the flow of execution and inspect variable states at critical junctures in the rule's logic. For a more structured approach, consider embedding logging mechanisms within the rule definition.

For example:

```
def _my_rule_impl(ctx):
    print("Debug: Starting rule implementation for target %s" %
          ctx.label)
    ...
    for src in ctx.files.srcs:
        print("Debug: Processing source file %s" % src.path)
    ...
```

## **Best Practices for Logging and Error Handling in Rule Development**

Effective logging is crucial for diagnosing issues in rule execution. Logs should provide enough context to understand the state of the build without being overwhelmed with unnecessary details. Use different verbosity levels to categorize the logs. Errors and warnings should be concise and informative, guiding the user towards potential solutions or further debugging steps.

Error handling in Bazel rules involves gracefully managing unexpected states and inputs. Use fail-fast principles, where

the rule execution halts immediately upon encountering a critical error. As an example:

```
def _my_rule_impl(ctx):
    if not ctx.attr.dependency:
        fail("Missing required dependency for target %s" %
            ctx.label)
    ...
```

This approach ensures that errors are caught early, making it easier to trace their origin. When handling non-critical errors, provide clear messages and, if possible, fallback mechanisms.

## [Performance Tuning for Large-scale Monorepos](#)

Optimizing Bazel builds can be particularly challenging due to the sheer volume of code and dependencies.

### [Divide and Conquer](#)

One effective technique is to split the monorepo using local repositories. This approach involves segmenting your monorepo into smaller, more manageable units, which can be treated as independent projects. This segmentation can be achieved using Bazel's `local_repository` rule. For instance, if you have a large monorepo with multiple independent projects, you can structure each project as a local repository:

```
local_repository(
    name = "project_a",
    path = "../project_a",
)
local_repository(
    name = "project_b",
    path = "../project_b",
```

)

By doing so, Bazel can treat these segments as separate entities, enabling more efficient builds. This setup allows Bazel to focus on the relevant parts of the codebase for each build, reducing the overall build time.

## **Advanced Configuration Settings for Improved Scalability**

To further enhance scalability, adjusting the `--jobs` flag in Bazel's configuration settings is crucial. This flag determines the number of jobs (actions) Bazel runs in parallel. For large monorepos, finding the optimal number of jobs is essential for maximizing the usage of available resources without overwhelming the system. This involves a balancing act between parallelism and the risk of resource contention.

A typical approach is to set the number of jobs to the number of cores available on the build machine. Nonetheless, for very large projects, this may not be optimal. In such cases, experiment with the `--jobs` flag to find the sweet spot. For example:

```
bazel build //my/project:all --jobs=32
```

In this example, Bazel is instructed to use up to 32 parallel jobs. The ideal number will depend on your specific build environment and the nature of the tasks being executed.

## **Case Studies on Performance Improvements in Complex Projects**

Examining real-world case studies can provide valuable insights into the effectiveness of these techniques. For instance, Google, with its massive codebase, uses Bazel to manage its monolithic repository. Google's approach involves finely tuned Bazel configurations and extensive use

of local repositories to manage different parts of its monorepo. This strategy has enabled them to handle complex builds efficiently, demonstrating the scalability of Bazel in a demanding environment.

Similarly, companies like Dropbox and Pinterest have reported significant improvements in build times and developer productivity after adopting Bazel for their large-scale monorepos. Dropbox, in particular, noted how Bazel's caching mechanisms and parallel build capabilities were crucial in handling their polyglot codebase.

In conclusion, effectively tuning Bazel for large-scale monorepos involves strategic segmentation of the codebase, careful configuration of build parallelism, and learning from real-world implementations. These approaches help in harnessing the full potential of Bazel, ensuring efficient and scalable builds in complex software development environments.

## **Conclusion**

This chapter provided a comprehensive exploration of advanced techniques for building and deploying services using Bazel, with a focus on optimizing performance and efficiency in large-scale monorepos. We delved into the intricacies of monitoring Bazel's performance, interpreting and visualizing profiling data to pinpoint bottlenecks, and the utility of custom scripts and command-line tools in analyzing this data. Understanding and leveraging the Build Event Protocol (BEP) and its integration with systems like Prometheus formed a crucial part of our discussion, highlighting the importance of continuous performance monitoring and automated regression testing. We examined the advanced aspects of remote caching and execution, emphasizing the significance of monitoring cache hit rates, diagnosing cache misses, and optimizing remote execution.

Moreover, we covered robust debugging techniques for Bazel rules, including the use of Starlark's debugging features and best practices in logging and error handling. The chapter rounded off with strategic insights into performance tuning for large-scale monorepos, showcasing real-world case studies and discussing divide-and-conquer strategies along with advanced configuration settings. This comprehensive chapter aims to empower developers with the knowledge and tools necessary to maximize the efficiency and effectiveness of their Bazel-based build and deployment processes, ensuring scalable and maintainable software development practices.

In the next chapter, we'll explore advanced Bazel features. We'll start with Bazel's caching mechanism, explaining how it reuses computed results to save time and resources. Then, we'll show how to publish your own Bazel rules and migrate a Maven project to Bazel, enhancing build efficiency. The chapter will also cover hermeticity in Bazel builds, the hot reload feature for faster development cycles, and the creation of custom toolchains for tailored build environments. We'll explain the use of Aspects and Aliases for improved dependency tracking and build configuration. Finally, we'll introduce experimental Bazel features, providing insights into the latest functionalities available in Bazel.

## **Recommended Reading**

For a deeper understanding of the topics covered in this chapter, the following online resources are highly recommended:

- "Effective Bazel Monitoring and Performance Analysis": A detailed guide on setting up comprehensive monitoring for Bazel builds, including performance

analysis and bottleneck identification. -  
<https://bazel.build/concepts/performance-analysis>

- “Advanced Profiling with Bazel”: This resource dives deep into the art of profiling Bazel builds, offering insights on interpreting profiling data for performance optimization. -  
<https://docs.bazel.build/versions/main/profiling.html>
- “Scripting with Bazel: Custom Solutions for Build Analysis”: A practical guide on writing custom scripts to parse and analyze Bazel profile data, enhancing build performance and efficiency. -  
<https://blog.bazel.build/2020/05/28/bazelcon-scripting.html>
- “Command Line Mastery for Bazel Profiling”: An advanced tutorial on using command-line tools to dissect and understand Bazel’s profiling data. -  
<https://www.tweag.io/blog/2020-06-17-bazel-profiling>
- “Visualizing Bazel Build Performance”: This article explores tools and techniques for visualizing profiling data from Bazel to identify performance issues. -  
<https://medium.com/@nlopez/bazel-visualizing-build-performance-6755a0b9b1b7>
- “Harnessing Bazel’s Build Event Protocol for Large Codebases”: A deep dive into Bazel’s BEP, offering strategies for its effective utilization in large-scale projects. - <https://docs.bazel.build/versions/main/build-event-protocol.html>
- “BEP to Prometheus: Monitoring Bazel Builds”: Learn how to convert Bazel’s Build Event Protocol output into a Prometheus-friendly format for enhanced monitoring. - <https://prometheus.io/docs/introduction/overview/>
- “Automated Performance Regression Testing in Bazel”: An insightful resource on developing automated tests to detect performance regressions in Bazel builds. -

<https://testing.googleblog.com/2020/08/advanced-bazel-testing-and-debugging.html>

- “Benchmarking and Baseline Implementations for Bazel”: Guidelines for implementing effective benchmarks and performance baselines within the Bazel ecosystem. - <https://www.infoq.com/presentations/netflix-bazel-large-scale/>
- “Continuous Monitoring of Bazel Build Metrics”: An advanced guide on setting up continuous monitoring systems for Bazel build metrics to ensure optimal performance. - <https://eng.uber.com/building-fast-monorepos-in-bazel/>
- “Optimizing Remote Caching and Execution in Bazel”: In-depth exploration of advanced techniques for optimizing remote caching and execution in Bazel, including monitoring and diagnosing cache misses. - <https://docs.bazel.build/versions/main/remote-caching-debug.html>
- “Debugging Techniques for Bazel Rules”: This resource provides an advanced perspective on debugging custom Bazel rules, including the use of Starlark’s debugging features. - <https://bazel.build/designs/2019/02/28/starlark-debugging.html>
- “Advanced Bazel Rule Development: Logging and Error Handling”: Best practices for logging and error handling in the development of custom Bazel rules. - <https://blog.aspect.dev/bazel-rules-best-practices>
- “Scaling Monorepos: Strategies and Tools”: Insights into performance tuning for large-scale monorepos, including divide-and-conquer strategies and configuration settings. -

[https://medium.com/@nrwl\\_io/scaling-monorepos-with-nx-12f217a06119](https://medium.com/@nrwl_io/scaling-monorepos-with-nx-12f217a06119)

- “Case Studies: Performance Improvements in Large-Scale Bazel Projects”: A collection of real-world case studies discussing performance improvements in complex projects using Bazel. - <https://bazel.build/case-studies>)

These resources offer a depth of knowledge and practical advice, ideal for professionals seeking to refine their skills in managing large codebases efficiently with Bazel.



## **CHAPTER 12**

# **Advanced Bazel Concepts**

## **Introduction**

As an experienced user familiar with Bazel, monorepos, and dependency management, you are now poised to extend your expertise by exploring deeper concepts such as publishing your custom Bazel rules to enhance build processes, understanding and leveraging the principle of hermeticity for consistent and reproducible builds, and effectively utilizing the 'watch' feature to maintain continuous insight into your project's state. This chapter also guides you through the intricacies of building custom toolchains, a cornerstone for tailored build environments, and the strategic use of aspects and aliases to refine and streamline your build configurations. Additionally, we will navigate the realm of experimental Bazel features, offering a glimpse into the future capabilities of this powerful tool. Each of these topics collectively forms a comprehensive guide, empowering you to master the advanced methodologies required for sophisticated service building and deployment with Bazel.

## **Structure**

In this chapter, we will cover the following topics:

- Comprehensive Exploration of Caching
- Publishing Your Own Bazel Rules
- Migrating a Maven Project to Bazel
- Hermeticity

- Bazel Hot Reload
- Building Custom Toolchains
- Aspects
- Aliases
- Exploring Experimental Bazel Features

## [Comprehensive Exploration of Caching](#)

Bazel operates as an artifact-first build tool, creating a Build Graph where artifacts are requested, and actions are executed to produce these artifacts. These artifacts encompass files within the workspace and external dependencies. Actions, which consume and produce artifacts, form the foundation of the Build Graph. An example is the execution of ``bazel build //:go_binary`` in a Golang project, where Bazel constructs a graph from the ``go_binary`` target, identifying necessary actions and input artifacts. This graph is executed in parallel by a scheduler.

The design of Bazel's Build Graph is predicated on the assumption that all actions are hermetic. This means that a fixed set of inputs will always yield a deterministic set of outputs. Hermetic actions, such as file creation from a template or concatenating files, produce predictable results. In contrast, non-hermetic actions, like those involving network calls or dependent on datetime or random elements, yield unpredictable outcomes. However, Bazel can render some non-hermetic actions hermetic, for instance, by using fixed random seeds for test actions.

Central to Bazel's caching mechanism are the Content Addressable Store (CAS) and the Action Cache (AC). In this system, artifacts are treated as blobs, each with a unique hash key based on their content, and stored in the CAS. Similarly, action metadata, including inputs and definitions,

are hashed to create cache keys stored in the AC. Bazel utilizes these caches in subsequent invocations to bypass redundant action executions. The Build Graph operates akin to a Merkle Tree, a data structure that aids in data de-duplication and caching of execution results.

Cache invalidation happens when modifications occur, such as altering `main.go`. This change updates the file's SHA256 hash, prompting the re-execution of dependent actions and updates in the CAS and AC while leaving unchanged parts of the workspace cached. Through hermetic actions, a Merkle-Tree Build Graph, and effective caching, Bazel achieves minimal, precise, and rapid incremental builds.

Exploring Bazel's local disk cache reveals its complexity. This layer of Bazel, though not frequently needing attention, has nuances and potential pitfalls important for users, especially newcomers, to understand. The local disk cache comprises two main components: the Repository Cache and the Action Cache (AC), with the focus here being on the Repository Cache. This cache, used for external dependencies declared in the WORKSPACE file, is prepared by Bazel during the loading-and-analysis phase for use in the execution phase.

The repository cache interacts mainly with three types of repository rules: `local_repository` for local dependencies, `git_repository` for remote git repositories, and `http_archive` for dependencies downloaded as archives. The `http_archive` rule is significant due to its common use and interaction with the repository cache. It involves declaring a SHA256 hash for the downloaded archive, which Bazel uses to manage and retrieve content from the CAS of the Repository Cache.

Incorrect hashes in the `http_archive` rules can lead to Bazel reusing incorrect cached content. This issue typically surfaces in a fresh build with an empty cache, as Bazel will

fail the build if the downloaded content doesn't match the expected hash.

The repository cache, enabled by default and located outside the `output_base` and `install_base` directories, can be shared across multiple Bazel workspaces and versions. It remains unaffected by the `bazel clean` command and is reused in clean build attempts. However, it's important to note that not all repository rules utilize the Repository Cache; only those employing specific Starlark APIs are cached.

Bazel's Persistent Action Cache, a built-in feature operating without extra configurations, serves to de-duplicate action execution and optimize builds. This cache doesn't store action results but tracks the state of input and output files, determining if actions need rerun based on changes. Its impact was demonstrated in a test with a minimal Bazel setup, where an initial build took about 9.7 seconds, and a fully cached build took just 0.5 seconds. Further tests showed that modifying and then reverting changes in the build graph led to no actions being executed the second time, highlighting the cache's efficacy.

The cache filters out certain records, such as those with the type `VALIDATION_KEY`, when using the `bazel dump --action_cache` command. This command is useful for understanding the contents of the Persistent Action Cache.

In summary, Bazel's Repository Cache is a crucial part of its on-disk caching system, primarily used to speed up the initial setup of a Bazel build by avoiding re-downloading external dependencies. Users can customize its location using the `--repository_cache` flag.

In CI environments, maintaining a warm Bazel JVM Server can save significant build time but may require stateful, long-lived CI workers, custom monitoring, and controlled CI processes. The benefits of faster builds and reduced

infrastructure costs often outweigh these challenges. Metrics play a key role in driving Bazel build optimizations, leveraging its advanced architecture and profiling capabilities to pinpoint and resolve performance issues.

## **Publishing Your Own Bazel Rules**

Publishing custom Bazel rules is a complex process that requires an understanding of the Bazel build system, rule writing, and repository management. This section dives into the intricacies of developing and distributing a set of Bazel rules, using the ``rules_jmh`` project available at [https://github.com/buchgr/rules\\_jmh](https://github.com/buchgr/rules_jmh) as a reference.

The initial step in creating your own Bazel rules involves setting up a repository structure that is both maintainable and accessible. Your repository should be organized in a way that separates the core logic of your rules from examples and tests. This structure not only aids in development but also helps other developers understand and use your rules more effectively. For instance, the ``rules_jmh`` project on GitHub demonstrates this approach, segregating its Java Microbenchmark Harness (JMH) rules, examples, and tests into distinct directories.

```
.
├── BUILD
├── README.md
├── WORKSPACE
├── defs.bzl
└── deps.bzl
```

When writing custom rules, it's imperative to focus on clarity and documentation. Each rule should be accompanied by comprehensive documentation explaining its purpose, inputs, outputs, and usage examples. This not only aids in maintenance but also provides valuable guidance for other developers who might use your rules. In

``rules_jmh``, each rule is documented within the source files and the **README**, offering clear guidance on how to implement benchmarks using the provided rules.

The **BUILD** file contains a simple `exports_files(["defs.bz1", "deps.bz1"])` rule, to explicitly export these two Starlark (`.bz1`) files. File `defs.bz1` contains custom rule definitions, while `deps.bz1` includes dependency declarations or macros for managing dependencies.

The `defs.bz1` file contains:

```
load("@rules_jvm_external//:defs.bz1", "maven_install")
def rules_jmh_maven_deps(
    jmh_version = "1.21",
    repositories = ["https://repo1.maven.org/maven2"]):
    """Loads the maven dependencies of rules_jmh.
```

Args:

```
    jmh_version: The version of the JMH library.
    repositories: A list of maven repository URLs where
                  to fetch JMH from.
```

```
    """
```

```
maven_install(
    name = "rules_jmh_maven",
    artifacts = [
        "org.openjdk.jmh:jmh-core:{}".format(jmh_version),
        "org.openjdk.jmh:jmh-generator-annprocess:
        {}".format(jmh_version),
    ],
    repositories = repositories,
)
```

```
def jmh_java_benchmarks(name, srcs, deps=[], tags=[], plugins=
[], **kwargs):
    """Builds runnable JMH benchmarks.
```

This rule builds a runnable target for one or more JMH benchmarks

```

specified as srcs. It takes the same arguments as
java_binary,
except for main_class.
"""
plugin_name = "{}_jmh_annotation_processor".format(name)
native.java_plugin(
    name = plugin_name,
    deps =
    ["@rules_jmh_maven//:org_openjdk_jmh_jmh_generator_annproc
    ess"],
    processor_class =
    "org.openjdk.jmh.generators.BenchmarkProcessor",
    visibility = ["//visibility:private"],
    tags = tags,
)
native.java_binary(
    name = name,
    srcs = srcs,
    main_class = "org.openjdk.jmh.Main",
    deps = deps +
    ["@rules_jmh_maven//:org_openjdk_jmh_jmh_core"],
    plugins = plugins + [plugin_name],
    tags = tags,
    **kwargs
)

```

Here's a simple breakdown of what each function does:

1. Defines the ``rules_jmh_maven_deps`` function:
  - a. Calls ``maven_install`` to define a new repository named ``rules_jmh_maven``.
  - b. Includes JMH core and annotation processor artifacts, with the version specified by ``jmh_version``.
2. Utilizes the provided ``repositories`` for fetching these dependencies.

3. Defines the ``jmh_java_benchmarks`` function:

- a. Creates a ``java_plugin`` named after the benchmark, to process JMH annotations using the JMH annotation processor.
- b. Configures this plugin to be private and applies any specified tags.
- c. Builds a ``java_binary`` with:
  - i. The provided ``name``, ``srcs``, ``deps``, ``tags``, and additional arguments (``**kwargs``).
  - ii. The main class is set to ``org.openjdk.jmh.Main``.
  - iii. Includes the JMH core library and the plugin for annotation processing in the dependencies.
  - iv. Adds the created annotation processor plugin to the build.

And the `deps.bzl` file contains:

```
load("@bazel_tools//tools/build_defs/repo:http.bzl",
     "http_archive")
def rules_jmh_deps():
    if "rules_jvm_external" not in native.existing_rules():
        http_archive(
            name = "rules_jvm_external",
            strip_prefix = "rules_jvm_external-1.2",
            sha256 =
                "e5c68b87f750309a79f59c2b69ead5c3221ffa54ff9496306937bfa1c
                9c8c86b",
            url =
                "https://github.com/bazelbuild/rules_jvm_external/archive/
                1.2.zip"
        )
```

The ``deps.bzl`` file in question performs the following functions:



- Defines a Function ``rules_jmh_deps``: This function, when called, will check and potentially download a dependency.
- Checks for Existing Dependency: Inside the ``rules_jmh_deps`` function, it first checks if the ``rules_jvm_external`` dependency is already present in the current Bazel workspace.
- Declares ``rules_jvm_external`` as a Dependency: If ``rules_jvm_external`` is not already present, it uses the ``http_archive`` function to declare it as a dependency.
- Specifies the Dependency details:
  - ``name``: The name given to this dependency is `"rules_jvm_external"`.
  - ``strip_prefix``: Indicates the prefix to strip from the archive, here specified as `"rules_jvm_external-1.2"`.
  - ``sha256``: Provides the SHA-256 checksum for integrity verification of the downloaded archive.
  - ``url``: Specifies the URL to download the ``rules_jvm_external`` archive, which in this case is version 1.2, hosted on GitHub.

Essentially, this ``deps.bzl`` file is a script used in Bazel to manage an external dependency, ensuring that ``rules_jvm_external`` is downloaded and integrated into the build environment if it's not already present.

Integration testing is a crucial aspect of rule development. Your tests should cover a variety of use cases and configurations to ensure the robustness of your rules across different environments and Bazel versions. Automated testing, as part of continuous integration, ensures that updates to Bazel or dependencies do not break your rules. The ``rules_jmh`` project employs Bazel's own testing facilities to guarantee the functionality of its rules.

After developing and thoroughly testing your rules, the next step is publishing them. You need to make your rules available for use in other Bazel projects. This can be achieved by hosting your rules in a version-controlled repository, such as GitHub. Users can then integrate your rules into their projects by adding a `http_archive` or `git_repository` declaration in their `WORKSPACE` file. For example, to use `rules_jmh`, one would add the following to their `WORKSPACE`:

```
http_archive(  
  name = "rules_jmh",  
  url = "https://github.com/buchgr/rules_jmh/archive/<commit-  
hash>.tar.gz",  
  sha256 = "<checksum>",  
)
```

Versioning plays a crucial role in the lifecycle of your rules. Adhering to semantic versioning principles allows users to understand the nature of updates and adjust their dependencies accordingly. It is also important to maintain backward compatibility as much as possible, or clearly document breaking changes in release notes.

Lastly, community engagement is vital. Actively maintaining the rule set, responding to issues, and participating in discussions can foster a community around your rules, leading to improvements, new use cases, and wider adoption.

In summary, publishing your own Bazel rules is not just about writing code. It involves thoughtful structuring of your project, meticulous documentation and testing, careful versioning, and ongoing community engagement. The `rules_jmh` project serves as an exemplary model, demonstrating best practices in each of these areas.

## [Migrating a Maven Project to Bazel](#)

Migrating from Maven to Bazel involves a paradigm shift in how build systems are managed. Bazel offers distinct advantages in terms of scalability and flexibility, particularly for large and multi-language projects. This section explores the advanced concepts of such a migration, assuming you're already familiar with Bazel, Monorepos, and Dependency Management in Bazel.

## Pre-migration Considerations

- **Parallel build systems:** It's advisable to run Maven and Bazel in parallel during the transition. This ensures a smooth migration for your development team, CI system, and other relevant systems.
- **Understanding differences:** Acknowledge the key differences between Maven and Bazel. Maven uses a top-level `pom.xml` file, whereas Bazel supports multiple `BUILD` files, offering more incremental builds. Bazel also allows expressing dependencies between languages and doesn't automate deployment as Maven does.
- **Defining the monorepo layout:** Ensure that all teams are involved in the layout design of the monorepo before beginning to transfer repositories into it. Also, be ready for multiple iterations of refactoring this layout.
- **Avoid including refactors:** While transitioning from a single repo to a monorepo, it's common to feel the urge to begin refactoring your app projects, but it's important to resist this temptation. Keep in mind that complexity and issues typically don't just add up; they often multiply on their own. Take notes and leave refactors as a post-migration task.

## Migration Process

1. **Creating the WORKSPACE file:** The migration begins by creating a `WORKSPACE` file at your project's root. This file

should list external dependencies if your project relies on files or packages outside its directories. Use ``rules_jvm_external`` to automate the listing of these dependencies.

2. **Setting up the BUILD files:** Bazel uses ``BUILD`` files to describe how to build your project, unlike Maven's ``pom.xml``. You should start by adding one ``BUILD`` file at the root of your project. This file will initially target a basic build of your project. As the migration progresses, add more ``BUILD`` files with more specific targets for incremental builds.
  - a. **Initial BUILD file:** The first ``BUILD`` file should use the ``java_library`` or ``java_binary`` rules, depending on your project structure (single or multiple Maven modules). The ``BUILD`` file specifies the ``name``, ``srcs``, ``resources``, and ``deps`` attributes.
  - b. **Adding more BUILD files:** For increased granularity and better maintainability, add ``BUILD`` files to individual Java packages, starting with those having the fewest dependencies.
3. **Building with Bazel:** Throughout the migration, regularly build your project with Bazel to ensure everything is configured correctly. Use ``bazel build //...`` to check that all targets build successfully.
4. **Refining and troubleshooting:** As you add more ``BUILD`` files, it's crucial to maintain a buildable state. Pay attention to the visibility of targets and the dependencies between them. Properly manage visibility to prevent issues such as libraries containing implementation details leaking into public APIs.
5. **CI integration:** Update your CI pipelines to accommodate Bazel builds.

6. **Testing and validation:** After completing the migration, thoroughly test the Bazel build to ensure it matches the outputs of the Maven build.
7. **Team training:** Ensure that your team is up to speed with Bazel's workflow and concepts.

## Post-migration Considerations

- **Revision of documentation:** It's important to revise all development documentation to reflect the new Bazel building process, the updated layout, and the revised management of both internal and external dependencies.
- **Introduce project refactors:** Aim to create numerous fine-grained projects from the existing coarse-grained ones. Consider factors like functional alignment, architectural stack, and the principle of single responsibility, among others, to guide the process of splitting projects in this refactor.
- **Minimize the size of the repository:** Begin by pinpointing assets that are large, duplicated, or unnecessary. Search for items such as log files, build directories, `node_modules`, Rust's target directories, inadvertently included Docker-compose volumes, and similar elements.

Migrating from Maven to Bazel, especially for large-scale projects, requires careful planning and execution. By understanding Bazel's unique features and incrementally adapting your build files, you can achieve a smooth transition. Remember, the key to a successful migration is maintaining both build systems in parallel, iterative refinement of `BUILD` files, and constant validation.`

For more detailed examples and guidance, you can refer to the comprehensive resources provided by [Bazel's official migration guide](<https://bazel.build/migrate>) and [Google's

source documentation](<https://bazel.googlesource.com>). These sources provide practical examples, such as the migration of the Guava project, and in-depth explanations of the steps involved.

## Hermeticity

Hermeticity in Bazel refers to the concept of ensuring that builds are self-contained and unaffected by the external environment. This characteristic is vital for achieving reproducible builds and effective use of features like remote caching and execution.

It is enforced through various mechanisms, but it's less strict compared to systems like Nix. In Bazel, hermeticity can be ensured by using sandboxes that isolate build steps from the surrounding environment. This isolation helps in specifying all build inputs explicitly, improving artifact reproducibility, and preventing unintended environmental influences on the build.

Bazel offers different execution strategies that control how sandboxing is applied. The strategies include local (or standalone), sandboxed, worker, docker, and remote executions. Of these, the `local` strategy should be avoided for hermetic builds as it doesn't employ sandboxing. Utilizing the `sandboxed` execution strategy ensures that commands are executed in an isolated environment, thereby maintaining hermeticity. It's also important to note that certain environments, like Windows, might not support sandboxing, which can affect the ability to enforce hermeticity.

Environment variables are a common source of non-hermetic behavior. Inherited environment settings can introduce variability into builds, which is counterproductive for achieving hermetic builds. Avoiding inheritance of the

host environment and being cautious with the use of flags like `--action_env` can help maintain hermeticity.

Some Bazel rules might default to using binaries from the system's `PATH`, which can vary between machines, affecting hermeticity. Moreover, the workspace status feature in Bazel can potentially introduce non-hermetic behavior, especially when its output is used to stamp build results. It's crucial to use this feature judiciously to avoid inadvertently breaking hermeticity.

Detecting issues related to hermeticity can be challenging. A recommended approach is to build the project in various environments and compare execution logs. Differences in these logs can reveal sources of non-hermetic behavior.

To illustrate hermeticity in practice, consider integrating Nix with Bazel for building a Haskell project. In this setup, Nix is used to build the entire compiler toolchain and system libraries, ensuring that every aspect of the build environment is controlled and reproducible. This approach ensures that the compiler, header files, and system libraries are consistent across builds, making them hermetic. The build targets are easy to cache, as they don't depend on the external environment.

The Bazel `WORKSPACE` file and `BUILD` files are configured to utilize these Nix-built components, ensuring that the build process is entirely self-contained. This setup illustrates how Bazel's flexible configuration can be used in conjunction with tools like Nix to achieve hermetic builds.

Achieving hermeticity in Bazel requires a careful setup of execution strategies, environment configurations, and rule selections. By understanding and applying these concepts, developers can create builds that are consistent, reproducible, and efficient, benefiting from features like remote caching and execution. Hermetic builds are a

cornerstone of reliable and scalable build systems, and mastering them is crucial for advanced Bazel users.

For a detailed exploration, including practical examples and configurations, refer to the articles on Tweag's blog:

- “How to keep a Bazel project hermetic?” - <https://www.tweag.io/blog/2022-09-15-hermetic-bazel/>,
- “Nix + Bazel = fully reproducible, incremental builds” - <https://www.tweag.io/blog/2022-09-15-nix-bazel/>,
- “Integrating Testwell CTC++ with Bazel” - <https://www.tweag.io/blog/2022-09-15-testwell-ctc-bazel/>).

These resources provide in-depth insights into achieving and maintaining hermetic builds using Bazel.

## **Bazel Hot Reload**

Bazel watcher, also known as ``ibazel``, is a source file watcher for Bazel projects that allows automatic rebuilding of targets when source files change. Key features include supporting build, test, and run commands, and the ability to integrate with live reloading for browsers.

For installation, various methods are available, including Homebrew for macOS:

```
brew install ibazel
```

NPM for JavaScript developers

```
npm install @bazel/ibazel
```

And building from source using Bazel itself as explained in the GitHub repository.

Running a target with ``ibazel`` involves using the command:

```
ibazel run //path/to/my:target
```

When source files are changed, ``ibazel`` automatically rebuilds the target. It can also produce and run commands



based on the output of Bazel commands, especially useful for automatically fixing common issues detected during a build.

Also, `ibazel` includes profiling capabilities with the `--profile_dev` flag, generating a profile output file about the build process and timing. This profiling feature provides detailed information about various events, such as the start, failure, and completion of builds, tests, runs, and source changes.

For more detailed information and code examples, you can refer to the Bazel watcher GitHub repository at <https://github.com/bazelbuild/bazel-watcher>.

## **Building Custom Toolchains**

This section is intended for software developers well-versed in Bazel, offering a deep dive into the advanced aspects of toolchain creation and management.

Toolchains in Bazel are pivotal for specifying the tools and compilers to be used for different platforms. When building a custom toolchain, the primary components to consider are the toolchain rule, the toolchain definition, and its registration.

A toolchain rule in Bazel is a Starlark rule that declares the set of tools it provides. This rule must produce a `ToolchainInfo` provider, which encapsulates the information about the provided tools. For instance, a simple C++ toolchain rule might look like this:

```
def _cpp_toolchain_impl(ctx):
    toolchain = platform_common.ToolchainInfo(
        cc = ctx.executable.cc,
        cxx = ctx.executable.cxx,
    )
    return [toolchain]
cpp_toolchain = rule(
```

```

implementation = _cpp_toolchain_impl,
attrs = {"cc": attr.label(executable = True),
        "cxx": attr.label(executable = True)},
)

```

In this example, the ``cpp_toolchain`` rule specifies the C and C++ compilers (``cc`` and ``cxx``). The attributes point to labels that correspond to the actual compiler binaries.

Next, the toolchain definition ties the rule to a specific set of tools. It's implemented as a BUILD file target and associates the toolchain rule with the actual tools for a specific target platform. A sample toolchain definition could be:

```

filegroup(
    name = "gcc-compiler",
    srcs = ["gcc"],
)
filegroup(
    name = "gxx-compiler",
    srcs = ["g++"],
)
cpp_toolchain(
    name = "my_cpp_toolchain",
    cc = ":gcc-compiler",
    cxx = ":gxx-compiler",
)

```

The toolchain is then registered in the `WORKSPACE` file. Registration makes the toolchain available for Bazel to select during the build. Registering a toolchain involves adding a reference to its definition in the ``toolchains`` attribute of the ``register_toolchains`` function. For example:

```

register_toolchains("//:my_cpp_toolchain")

```

Advanced usage of toolchains also involves understanding constraints. Constraints are used to specify the conditions under which a particular toolchain should be chosen. They are particularly useful when the same set of source code

must be built for different environments, such as different operating systems or custom hardware.

When defining constraints, one would typically define a set of constraint values and a `constraint_setting` to group them. For instance:

```
constraint_setting(  
  name = "os",  
)  
constraint_value(  
  name = "linux",  
  constraint_setting = ":os",  
)  
constraint_value(  
  name = "windows",  
  constraint_setting = ":os",  
)
```

In this scenario, the ``os` constraint_setting` allows the specification of different operating systems as constraint values. The toolchain rule can then use these constraints to select the appropriate compiler based on the target platform.

For an in-depth understanding of building custom toolchains in Bazel, it is recommended to refer to the two key resources. The first is John Millikin's detailed guide on Bazel toolchains, available at <https://john-millikin.com/bazel-school/toolchains>. This guide offers a comprehensive overview, from the basics to more intricate aspects of toolchain configuration. The second essential resource is Jay Conrod's insightful article on writing Bazel rules, platforms, and toolchains, accessible at <https://www.jayconrod.com/posts/111/writing-bazel-rules--platforms-and-toolchains>. Jay's article explores the practicalities of rule writing and the utilization of platforms and toolchains, providing valuable examples and expert

insights. Together, these resources serve as a robust foundation for mastering custom toolchain creation in Bazel. In summary, building custom toolchains in Bazel involves defining a toolchain rule, creating a toolchain definition, registering the toolchain, and optionally using constraints for environment-specific tool selection. This process enables developers to have fine-grained control over the tools used in their builds, ensuring compatibility and efficiency across various development environments.

## Aspects

Aspects provide a mechanism to traverse the dependency graph and apply additional actions to targets in this graph. This is particularly useful for scenarios such as code generation, linting, or gathering metrics where actions need to be performed on a set of targets and their dependencies.

Aspects are defined similarly to rules in Bazel. They consist of an implementation function and an attribute dictionary. The implementation function is where the aspect's logic resides. It receives two key pieces of information: the target that the aspect is attached to and the context for the aspect's execution. Through this context, the aspect can access the target's attributes, dependencies, and outputs.

To illustrate, consider an aspect designed to gather code coverage information. This aspect, let's call it ``code_coverage_aspect``, would traverse the targets and their dependencies, attaching itself to each. For each target, it would collect coverage data, possibly by utilizing additional tools or scripts.

Here's an example of how such an aspect might be defined:

```
def _code_coverage_impl(target, ctx):  
    # Logic for collecting coverage data  
    coverage_data = collect_coverage(target)  
    # Actions to perform with coverage data
```

```

process_coverage_data(coverage_data)
return [OutputGroupInfo(coverage=coverage_data)]
code_coverage_aspect =
    aspect(
        implementation=_code_coverage_impl,
        attr_aspects=["deps"],
    )

```

In this example, ``collect_coverage`` is a hypothetical function that gathers coverage data, and ``process_coverage_data`` is another function that performs necessary actions with this data, such as generating reports. The aspect is then attached to the ``deps`` attribute of the target, which means it will propagate along the dependencies of the target.

One of the critical aspects of Bazel aspects is how they propagate through the dependency graph. The ``attr_aspects`` attribute controls this propagation. In our ``code_coverage_aspect``, specifying ``["deps"]`` means that the aspect will propagate along the dependencies of the targets to which it is applied. This allows for a thorough analysis of the entire codebase that the target depends on.

Furthermore, aspects can be used in conjunction with rules. Demonstrating this, a rule that defines a library or binary can be enhanced with an aspect to perform additional checks or actions specific to that rule. This makes aspects a versatile tool for extending the functionality of existing rules without modifying them.

```

my_rule = rule(
    implementation = _my_rule_impl,
    attrs = {
        "deps": attr.label_list(aspects = [code_coverage_aspect]),
    },
)

```

In this rule definition, the ``code_coverage_aspect`` is attached to each target in the ``deps`` attribute, ensuring that the

coverage data is collected not just for the target defined by ``my_rule``, but for all its dependencies as well.

Aspects in Bazel provide a robust mechanism for extending the functionality of builds and analyses, offering deep insights and controls over the build process. They are particularly useful in large-scale projects where understanding and controlling the behavior of a complex dependency graph is crucial. By leveraging aspects, developers can create more modular, maintainable, and comprehensive build and test environments.

## Aliases

The concept of aliasing in Bazel is a powerful tool for managing dependencies and restructuring projects without altering the actual physical paths of the targets. An alias in Bazel serves as a redirect, or a pointer, to another target. It allows users to reference a target by a different name, providing flexibility in organizing and refactoring codebases, especially in monorepos where the structure and dependencies are complex.

One advanced use of aliases is to facilitate the migration of targets without disrupting dependents. For instance, if a target needs to be moved or its visibility needs to be changed, an alias can temporarily redirect dependents to the new target location, ensuring a smooth transition. This is particularly useful in large projects where direct modification of all dependents is impractical.

In Bazel, an alias is defined using the ``alias`` rule. The rule has a ``name`` attribute, which is the alias name, and an ``actual`` attribute, which is the label of the target that it points to. An optional ``visibility`` attribute can also be set to control which packages can access this alias.

A practical code example of using an alias might look like this:

```
alias(  
  name = "my_lib",  
  actual = "//path/to/real:lib",  
)
```

In this example, ``my_lib`` acts as an alias for the target ``//path/to/real:lib``. This allows other targets to depend on ``//path/to:my_lib`` instead of the actual target. Such abstraction enables the underlying target to be restructured or moved without requiring changes in all the dependents.

Another sophisticated usage of aliasing is in handling the versioning of dependencies. To exemplify, if a project depends on multiple versions of a library, aliases can be used to point to the specific version required by different parts of the project. This technique simplifies dependency management, particularly in a monorepo setup where multiple projects with varying dependency requirements coexist.

Aliases also play a helpful role in creating a more intuitive and user-friendly structure in a project. They can be used to create shorter or more meaningful target names, enhancing the readability and maintainability of the build scripts. For instance, rather than using long and complex target paths, an alias can provide a simple and clear reference that is easier to understand and use.

To sum up, Bazel aliases are a versatile feature that, when used strategically, can significantly improve the organization, maintainability, and scalability of large codebases. They offer an advanced method for managing dependencies, refactoring projects, and simplifying build configurations in complex Bazel environments.

## **[Exploring Experimental Bazel Features](#)**

Bazel continually introduces experimental features that offer advanced functionalities not yet available in the stable releases. To enable an experimental feature, developers must use the `--experimental_<feature_name>` flag in their Bazel command. For instance, to experiment with a new optimization algorithm, you might use:

```
bazel build --experimental_optimization=algorithm2
//my/project:target
```

This command instructs Bazel to apply the specified experimental optimization algorithm during the build process.

It's important to approach these experimental features with caution. As they are not part of the stable release, they may have unknown bugs or might change significantly before being officially released. It is advisable to test these features in a controlled environment, such as a separate branch or a dedicated testing project, to avoid disrupting the main development workflow.

One area where experimental features shine is in advanced dependency analysis. Bazel's experimental dependency graph analyzers can provide deeper insights into the relationships between components in a monorepo. For instance, an experimental feature might allow developers to generate a more detailed visualization of the dependency graph, aiding in the identification of bottlenecks or unnecessary dependencies:

```
bazel query --experimental_dep_graph_visualization
//my/project:target
```

This command could generate an enhanced graphical representation of the dependencies, revealing complex interconnections that are not evident in standard visualizations.

Performance tuning is another domain where experimental features can be invaluable. Bazel may introduce



experimental flags that alter the way resources are allocated during builds, potentially leading to significant performance improvements. For example, a new scheduler algorithm can be tested using:

```
bazel build --experimental_scheduler_algorithm=new_algo
//my/project:target
```

This command tells Bazel to use a new, experimental scheduling algorithm that might improve build times, especially in large-scale projects with numerous dependencies.

Experimental features often include new capabilities for custom rule development. These can range from new APIs to enhanced debugging tools, providing rule authors with more powerful and flexible ways to create and test custom-build rules. For instance, an experimental API might be accessed as follows:

```
load("@bazel_skylib//rules:experimental_api.bzl",
"experimental_rule")
```

In this snippet, a new, experimental API from the `bazel_skylib` repository is loaded, allowing the developer to experiment with functionalities not available in the main API.

The exploration of experimental features in Bazel is a journey that requires a careful balance between the excitement of new capabilities and the caution needed to maintain a stable build environment. These features, while potentially transformative, should be integrated thoughtfully and tested rigorously. As Bazel evolves, these experimental features often pave the way for the next generation of build automation techniques, making their exploration a valuable endeavor for any advanced Bazel user.

## [Conclusion](#)

We began with an overview of the process, progressing through the intricacies of publishing custom Bazel rules, which enhance the flexibility and applicability of Bazel in diverse environments. We discussed the concept of hermeticity, underlining its importance in achieving reproducible and reliable builds. The utility of ‘watching’ projects for changes was explored, emphasizing its role in continuous integration and development workflows. We delved into the creation of custom toolchains, a critical aspect of supporting diverse platforms and languages. The chapter also covered the use of Aspects and Aliases, powerful features for extending and customizing build behaviors. Lastly, we navigated the realm of experimental Bazel features, a segment vital for staying ahead in rapidly evolving development ecosystems. Each topic in this chapter collectively contributes to a deeper understanding of how Bazel can be leveraged for robust and efficient building and deployment of services, reinforcing its status as a versatile tool in the arsenal of modern software developers.

In the next chapter, we will explore three case studies demonstrating Bazel’s application in software development. The first case study focuses on building a full-stack digital service with Bazel, illustrating its use in managing dependencies and builds. The second case study examines how Bazel aids in developing a serverless service platform, emphasizing build optimization. The third case study discusses Bazel’s role in managing an enterprise software monorepo, showcasing its effectiveness in handling large-scale codebases. These examples aim to provide a practical understanding of Bazel’s implementation in different development contexts.

## **[Recommended Reading](#)**

For readers looking to deepen their understanding of advanced Bazel concepts, the following resources are highly recommended:

- **How Bazel Works:** This article serves as an excellent starting point for understanding Bazel's fundamental concepts and how its build process operates. It lays the groundwork for comprehending the intricacies of Bazel's caching system. - <https://sluongng.hashnode.dev/bazel-caching-explained-pt-1-how-bazel-works>
- **Bazel In-Memory Cache:** A deep dive into Bazel's in-memory cache, this piece elucidates how Bazel optimizes builds within a single session, highlighting the role and limitations of the ephemeral in-memory cache. - <https://sluongng.hashnode.dev/bazel-caching-explained-pt-2-bazel-in-memory-cache>
- **Repository Cache:** This article focuses on the repository cache, a key component in Bazel's caching strategy. It discusses how Bazel caches external dependencies, a crucial feature for the efficient management of large-scale projects. - <https://sluongng.hashnode.dev/bazel-caching-explained-pt-3-repository-cache>
- **Persistent Action Cache:** The final piece in the series, this article explores the persistent action cache. It provides an in-depth look at how Bazel caches actions across builds, ensuring efficiency and consistency over time. - <https://sluongng.hashnode.dev/bazel-caching-explained-pt-4-persistent-action-cache>
- **Extending Bazel:** To gain insights into advanced concepts like writing custom rules and distributing them, as well as understanding macros, depsets, aspects, and repository rules. This resource also covers the Starlark language used in Bazel - Extending Bazel on Bazel.build - <https://bazel.build/docs/extending.html>

- **Extension Overview:** Offers a comprehensive guide on extending the BUILD language using macros and rules, detailing the evaluation model of Bazel and providing practical guidance on creating your own extensions - Extension Overview on Bazel.build - <https://bazel.build/docs/extension-overview.html>
- **Workspaces, Packages, and Targets:** This resource is essential for understanding the organization of code in Bazel, encompassing workspaces, packages, and targets, and their roles in the build process - Workspaces, Packages, and Targets on Bazel.build - <https://bazel.build/docs/workspaces-packages-targets.html>
- **Hermeticity in Bazel:** For an advanced understanding of hermeticity and its importance in reproducible builds, refer to the relevant sections under Bazel documentation - Hermeticity on Bazel.build - <https://bazel.build/docs/hermeticity.html>
- **Watching Projects with Bazel:** Discover advanced techniques for monitoring changes in your Bazel projects. This includes setting up watchers and using Bazel's query language for real-time insights - Bazel Query Language - <https://bazel.build/docs/query.html>
- **Building Custom Toolchains:** Delve into the creation and utilization of custom toolchains in Bazel, an advanced topic crucial for cross-platform and language-specific builds - Toolchains on Bazel.build - <https://bazel.build/docs/toolchains.html>
- **Aspects in Bazel:** Aspects are a powerful feature for traversing and manipulating the build graph. This resource provides an in-depth look at their usage and applications - Aspects on Bazel.build - <https://bazel.build/docs/aspects.html>

- **Understanding and Using Aliases:** For advanced manipulation of dependencies and targets, understanding aliases in Bazel is crucial - Aliases on Bazel.build - <https://bazel.build/docs/be/general.html#alias>
- **Exploring Experimental Bazel Features:** Stay up-to-date with the cutting-edge features being tested in Bazel, understanding their potential applications and implications - Bazel Release Notes - <https://bazel.build/docs/release-notes.html>
- **Bazel Community and Support:** Engaging with the Bazel community can provide additional insights and support for advanced Bazel concepts - Bazel Community - <https://bazel.build/docs/community.html>

These resources will provide a comprehensive understanding of advanced Bazel topics, supporting a deeper dive into this powerful build automation tool.

# **CHAPTER 13**

## **Case Studies and Real-World Examples**

### **Introduction**

In this chapter, we explore three compelling case studies that exemplify the implementation of cutting-edge technologies in diverse environments. The first case study focuses on the creation of a full stack digital service, emphasizing the integration and challenges of melding front-end and back-end technologies. The second case study delves into the development of a serverless service platform, illustrating the advantages and complexities of serverless computing. The third case study examines the application of Bazel in an Enterprise Software Monorepo, providing insights into managing and scaling software builds in a complex Monorepo codebase. Together, these studies offer a rich perspective on modern software development trends, challenges, and innovations.

### **Structure**

In this chapter, we will cover the following topics:

- Case Study 1: Building a Full Stack Digital Service
- Case Study 2: Building a Serverless Service Platform
- Case Study 3: Applying Bazel in an Enterprise Software Monorepo

## Case Study 1: Building a Full Stack Digital Service

The first case study is about a modular, end-to-end solution designed to enable firms to create and deploy cloud-based digital banking entities more rapidly than before. It is a comprehensive system that integrates various components essential for modern digital banking operations. Moving forward, this solution will be identified as “CS1” due to restrictions imposed by a non-disclosure agreement and here are some of its key aspects:

- **Rapid Deployment:** CS1 allows for the quick setup of a new bank, significantly reducing the time to market. It is possible to complete the initial feature setup for a virtual banking operation in as little as three months.
- **Cloud-Based Solution:** Being a cloud-based platform, CS1 offers flexibility and scalability, which is crucial for adapting to changing market demands and customer needs.
- **Integration with AWS:** CS1 is built around Vault (ThoughtMachine’s cloud-native core banking engine), Mambu (another modern core banking engine), and utilizes AWS’s native cloud services. This integration ensures that CS1 is hosted on a robust and secure infrastructure, offering high reliability and performance.
- **Modular System:** The platform is fully modular, allowing banks to create highly flexible end-to-end solutions. This modularity enables banks to add new functionalities as required and adapt the platform to their specific needs.
- **Cost and Time Efficiency:** By leveraging CS1, firms can enter new markets quickly and reduce operational costs. The platform is designed to streamline the

banking setup process, making it more efficient in terms of both time and resources.

- **Pluggable Components:** CS1 offers pluggable components such as a predefined operating model and standardized business processes. This feature reduces time to market, lowers project risk, and optimizes return on investment.
- **Support for Multiple Entities:** CS1 delivers the capabilities required to support multiple-entity banking operations, allowing for a diverse range of banking services and products to be managed under a single platform.

In summary, CS1 represents a significant advancement in digital banking technology, offering a rapid, scalable, and flexible solution for setting up cloud-based digital banks. Its integration with AWS and the modular, cloud-native core banking engine Vault, positions it as a comprehensive solution for modern banking needs.

This project, envisioned as a virtual bank accelerator, aimed to provide a foundational structure for greenfield projects, allowing them to avoid starting from scratch. It was brought to life by a dedicated team of over 30 professionals, encompassing a range of multidisciplinary roles. The team structured their work into two main streams: one focused on defining business features and the other on development and operations. Together, they successfully developed more than 30 generic business features. The technical stack was diverse, utilizing Java, Kotlin, Swift, HTML, CSS, TypeScript, Python, and Starlark, showcasing the team's broad range of skills and adaptability to different programming languages and technologies.

In the initial stages of our project, we adopted a poly-repository approach. However, we soon encountered several challenges with this strategy. Coordinating changes across



multiple repositories proved difficult, as it required synchronizing updates and ensuring compatibility. We also faced challenges in refactoring code, as changes often needed to be mirrored across different repositories, complicating the process. Additionally, managing versions became complex, as we had to track which versions of different repositories were compatible with each other. This approach also led to reduced visibility and discoverability within our codebase, making it harder for team members to be aware of all existing components and their interactions. Finally, we struggled with tracking and implementing global changes and impacts, such as updates to coding standards or security policies, as these had to be replicated across all repositories.

## **Motivation for Adopting Bazel**

For our CS1 project, the decision to use Bazel as our build tool brought numerous benefits, aligning well with our specific needs and challenges. Bazel's ability to support multiple languages and platforms was a crucial factor, as it offered built-in support for a variety of programming languages and the capability to build software for diverse platforms such as Linux, Windows, and macOS, all from a single source tree. Furthermore, Bazel's integration with testing frameworks significantly enhanced our development efficiency. Its support for automated testing allowed our developers to run tests only on the parts of the codebase that were affected, thanks to its fine-grained dependency graph. This precise and granular dependency management also proved invaluable in effectively handling our project's complex dependency graphs. Moreover, the query language provided by Bazel enabled a deeper understanding of our complex codebase, making it easier to manage and evolve. Lastly, being a product originally developed by Google, Bazel is supported by a growing community and ecosystem,

offering robust support, along with a range of plugins and extensions that further augment its functionality. These aspects collectively made Bazel an optimal choice for our project.

The primary goal and expected outcome of implementing Bazel for the CS1 project centered around achieving team alignment and synchronicity in the development process. By adopting Bazel, we aimed to ensure that each business feature drove the code contributions, fostering a more cohesive and streamlined approach to development. This methodology was particularly significant in aligning our development flow with agile practices, which was a crucial aspect of the project's success. Bazel's efficient management of dependencies and its support for multiple languages and platforms facilitated this alignment, enabling the team to work cohesively and react swiftly to changing requirements. The ability to integrate seamlessly with various testing frameworks and the fine-grained control over builds and tests further reinforced our agile methodology. In essence, Bazel acted as an accelerator, not just in terms of build efficiency, but more importantly, in harmonizing our team's efforts and ensuring that our development practices were in lockstep with agile principles, thereby greatly contributing to the project's overall success.

## **Implementation Strategy**

In integrating Bazel into the existing development workflow for our project, we focused on managing deliverables at every architectural layer with an emphasis on both the overarching components and the individual, atomic elements. Bazel was configured to handle the compilation and assembly of all deliverable components, which included produced binaries from various segments of our architecture. This setup ensured that each discrete project

or module, no matter how small, was accounted for and managed efficiently. The integration extended beyond just managing the build processes; it was pivotal in maintaining consistency and efficiency across all development stages. Bazel was employed not only in the local development environments of our team members but also as a critical component in our continuous integration (CI) pipelines. By doing so, we achieved a seamless transition between local development and the more extensive integration and testing stages. This strategy ensured that the build and test processes were uniform, regardless of where they were executed, thereby enhancing both developer productivity and the reliability of our builds. The implementation of Bazel thus became a cornerstone in our development process, bridging the gap between individual development efforts and larger-scale integration and deployment activities.

During the implementation process of Bazel for our project, we made several key considerations and decisions to define an effective Monorepo layout. The chosen structure, as illustrated, was carefully designed to cater to our project's unique needs while promoting clarity and efficiency.

```
.
├── BUILD
├── CODEOWNERS
├── CONTRIBUTING.md
├── README.md
├── WORKSPACE
├── components
│   ├── android
│   ├── backend
│   └── ios
├── samples
│   ├── android
│   ├── backend
│   └── ios
```

```
|— single-developer-hub
|   |— back-end
|   └─ front-end
|— third-party
|   |— android
|   |— ios
|   |— java
|   |— nodejs
|   |— python
|   └─ web
|— tools
|   |— AWS
|   |— angular
|   |— ios
|   |— rules
|   |— scripts
|   |— setup
|   └─ tulsi
└─ virtual-banking
    |— bank-as-a-service
    |— channels
    |— corebank
    └─ user-acceptance-test
```

The root of the repository contains essential files like **BUILD**, **WORKSPACE**, and **README.md**, which are pivotal for Bazel's operation and provide initial guidance for developers. The **BUILD** file defines the build rules, whereas **WORKSPACE** establishes the workspace context, crucial for Bazel's functionality. **CODEOWNERS** and **CONTRIBUTING.md** were included to streamline contribution processes and maintain code quality.

The components directory is segregated by platform (Android, iOS, and so on), ensuring a clear separation of concerns and facilitating platform-specific optimizations.

This structure aids developers in quickly locating and focusing on relevant parts of the codebase.

The samples folder, particularly with its iOS example, serves as a reference model for developers, illustrating best practices and providing a template for new components.

In single-developer-hub, we divided the application into back-end and front-end sections, embracing a modular approach that allows for independent development and scaling of each part.

The third-party directory is organized by technology stacks (for example, Android, iOS, Java), which simplifies dependency management and streamlines the integration of external libraries and frameworks.

Under tools, we included various utilities and scripts, categorized by their purpose or technology (like AWS, Angular, iOS). This not only supports Bazel's operations but also enhances the developer experience by providing essential tools for different aspects of the development process.

Lastly, the virtual-banking section represents a specific project within the Monorepo. It is further broken down into sub-components like bank-as-a-service, channels, and core bank, exemplifying how larger projects can be modularized within a Monorepo for better manageability.

The decision to adopt this layout was driven by the need for a scalable, organized structure that supports a diverse range of technologies and components while maintaining ease of navigation and build efficiency. This structure facilitates Bazel's performance in managing dependencies and builds across various components, ensuring a streamlined workflow that aligns with our agile practices and overarching architectural vision.

## **Challenges and Solutions**

During the implementation of Bazel in our project, we encountered several specific challenges, particularly with building TypeScript and iOS applications and managing dependencies. These challenges stemmed largely from the limitations in the existing Bazel rules for these technologies.

Firstly, building TypeScript applications proved to be a significant hurdle. The available Bazel rules for TypeScript were somewhat limited, lacking the flexibility we needed for our project's specific requirements. This limitation necessitated a deeper dive into custom configurations and workarounds to ensure that our TypeScript builds were efficient and aligned with our development practices.

Similarly, iOS development presented its own set of challenges. The complexity of iOS builds and dependency management within the Bazel ecosystem was a considerable obstacle. Initially, managing dependencies for iOS projects was particularly troublesome. However, we found a solution by integrating Tulsi, a Bazel build system generator for Xcode projects. Tulsi significantly streamlined our iOS builds, making the process more manageable and consistent with our overall development workflow.

In addition to these platform-specific challenges, we also faced issues with code generation. To address this, we developed a custom Bazel rule for OpenAPI code generation. This rule proved to be highly beneficial as it allowed us to tailor the code generation process to our project's evolving needs, which included several customizations throughout the project's lifecycle. The flexibility and control afforded by this custom rule were instrumental in maintaining efficiency and ensuring that generated code met our standards and requirements.

Furthermore, we encountered challenges with Spring Boot code generation and packaging. The standard Bazel rules did not cater to the specific customizations our project

required. To overcome this, we developed another custom Bazel rule tailored to our Spring Boot applications. This rule enabled us to efficiently handle the generation and packaging of Spring Boot code, incorporating the numerous customizations that were essential for our application. The development of this rule was a significant undertaking, but it paid off by providing a tailored solution that perfectly fit our project's unique requirements.

## **Results and Impact**

Adopting Bazel for our project yielded significant quantitative and qualitative outcomes that positively impacted our team dynamics and overall development efficiency. One of the most noteworthy changes was the enhanced ability of the team to collaborate and contribute across the full stack. This cross-functional understanding and involvement led to an impressive acceleration in development, quantified at a 24% increase in overall pace. In terms of build times, we observed a remarkable reduction, with builds that previously took 18 minutes now completed in less than 6 minutes. This decrease in build times, alongside reports from developers of improved productivity, greatly contributed to the scalability of our codebase, allowing us to manage and expand our project more effectively. An unexpected benefit that emerged from this transition was the opportunity for developers to learn new coding languages, an aspect that, while challenging, contributed to the team's growth and versatility. The adoption of Bazel, thus, not only met our initial objectives but also fostered a more skilled and agile development team.

## **Lessons Learned**

The journey of implementing and using Bazel in our project was rich with learning experiences and insights, although not without its challenges. A key takeaway was the steep learning curve associated with Bazel, which required a significant investment of time and effort to master. Moreover, we encountered compatibility issues, most notably with Windows, where Bazel's support was relatively poor. This was further compounded by the realization that community rules, while helpful, are not always robustly supported or maintained, posing additional challenges in integration and usage.

Despite these hurdles, the insights we gained were invaluable, particularly for projects or teams with similar characteristics to ours. We found that Bazel is particularly beneficial for highly skilled engineering teams dealing with a multitude of programming languages and managing large volumes of code. Its effectiveness becomes even more pronounced in environments where complex business products are being developed, as Bazel's strengths in build optimization and dependency management can significantly streamline the development process.

For organizations or teams considering adopting Bazel, we recommend dedicating at least two senior engineers to manage the deployment and ongoing support of Bazel. Their expertise will be crucial in navigating the intricacies of Bazel and addressing the unique challenges that arise. Furthermore, for teams working in Windows environments, we suggest leveraging the Windows Subsystem for Linux (WSL) to circumvent some of the compatibility issues with Windows. This tactic can help in creating a more seamless and efficient development experience when integrating Bazel into your workflow.

## **Future Plans and Considerations**



As we continue to evolve and enhance our use of Bazel, there are several ongoing and planned improvements that are set to further optimize our development process. A significant focus is on enhancing the performance of our continuous integration (CI) system. We plan to achieve this by implementing persistent Bazel workers, which will allow for more efficient utilization of resources, and by setting up a dedicated cache node, which is expected to drastically reduce build times and improve overall CI efficiency. In addition to these performance enhancements, we anticipate a shift in our approach to rule writing. Specifically, we aim to transition many of our existing `genrule` commands into pure Starlark rules, which will offer more flexibility and maintainability. This shift aligns with our broader strategy of leveraging Bazel's capabilities to their fullest extent. Looking further ahead, our long-term vision for Bazel includes its integration into dynamic aspects of our workflow, such as the generation of dynamic Helm chart code. This integration is expected to bring a new level of automation and efficiency to our deployment processes, solidifying Bazel's role as a central pillar in both our project and organizational infrastructure.

## **Outcome**

This case study offers a comprehensive insight into the deployment of a modular, cloud-based digital banking solution. This project, aimed at rapidly deploying virtual banking operations, leveraged a diverse technical stack and encompassed over 30 generic business features. Initially, the team faced challenges with a poly-repository approach, encountering issues in coordination, code refactoring, and version management. The adoption of Bazel emerged as a pivotal decision, aligning the team's development efforts with the project's agile practices and complex technical demands.

Bazel played a crucial role in this use case success. It enabled multi-language support, streamlined testing processes, and provided fine-grained dependency management, all of which were integral to the project's needs. The tool facilitated team alignment, contributing to a 24% increase in development speed and reducing build times from 18 to less than 6 minutes. Bazel's implementation not only accelerated development but also fostered a more cohesive and agile team dynamic, crucial for the project's ambitious goals.

This case study underscores the significance of choosing the right build tool in complex software development projects. Bazel's effectiveness in handling diverse languages, large code volumes, and intricate project structures demonstrates its suitability for high-level engineering teams. The lessons learned from this use case project highlight the importance of dedicated resources for tool implementation and the potential challenges in learning curves and compatibility issues. These insights are valuable for the broader software development community, especially for those contemplating Bazel in similar multifaceted environments. The future enhancements planned for Bazel usage in this use case, including the transition to Starlark rules and the integration of dynamic code generation, suggest a long-term commitment to leveraging Bazel's full potential, signaling its growing importance in the field of software development.

## **Case Study 2: Building a Serverless Service Platform**

The project in question was an ambitious undertaking by a global bank to develop a financial services platform tailored for a new regional market, specifically targeting the family customer segment. This platform was unique in its approach, as it integrated services from various third-party

providers, thereby offering a comprehensive suite of financial solutions to its users. The objective was to create a seamless, user-friendly experience that catered to the specific needs of families in this region.

From a contextual standpoint, the project was extensive and complex, encompassing over 400 distinct business operations. The initial development team consisted of more than 40 developers, which eventually expanded to over 110 engineers. This growth was indicative of the project's scale and the diverse skill sets required to bring it to fruition. The technical scope included the development of a hybrid mobile application, middleware, core banking functionalities, along with query and operational persistence systems. Besides, the project involved numerous internal and external integrations, making it a highly interconnected and multifaceted initiative.

In terms of infrastructure, the decision was made to utilize cloud-native services and serverless architectures. This method was strategically chosen to manage the unpredictability of scalability demands and to minimize operational costs. By leveraging the flexibility and efficiency of cloud-based solutions, the project aimed to ensure robust performance and reliability, even under varying load conditions.

A critical aspect of the project was the stringent timeline. The team was tasked with delivering a 'friends and family' version of the platform in less than 18 months. This deadline was driven by regulatory requirements specific to the new region. The tight timeline added an extra layer of complexity to the project, requiring meticulous planning, efficient execution, and the ability to rapidly adapt to evolving needs and challenges.

## **Motivation for Adopting Bazel**

The decision to choose Bazel over other build tools for our project was driven by a variety of specific reasons, each contributing to the overarching goal of optimizing our development process. Firstly, Bazel's capacity for highly scalable builds made it an ideal choice for our expanding project needs. Its robust support for multiple languages and platforms was another critical factor, as it allowed for a more versatile and inclusive development environment. The integration with various testing frameworks offered by Bazel greatly streamlined our testing process, ensuring more reliable and efficient quality control. Also, Bazel's fine-grained dependency management system allowed for precise control over our project's complex dependencies, enhancing overall build integrity and consistency. The deep codebase understanding facilitated by Bazel was invaluable for navigating and managing our large and intricate codebase. The strong community and ecosystem surrounding Bazel provided us with the necessary support and resources, further cementing our choice. Lastly, the performance optimization capabilities of Bazel were a key consideration, promising to enhance the efficiency and speed of our builds.

The implementation of Bazel was aimed at achieving specific goals and expected outcomes that were critical to the success of our project. Our primary objective was to significantly speed up the development cycle, enabling a more agile and responsive development process. We also sought to foster an environment where full-stack development teams could collaborate effectively around the same business feature, thereby promoting better integration and synergy within the teams. Enhancing the developer experience was another vital goal, as we aimed to provide a more streamlined and user-friendly development workflow. In addition, enabling our teams to work with ephemeral environments was a crucial outcome, as it allowed them to

test and showcase the features they were developing in a more flexible and dynamic manner. These goals and outcomes were central to our decision to implement Bazel, guiding our approach and strategies throughout the integration process.

## Implementation Strategy

The integration of Bazel into our existing development workflow was meticulously planned and executed, aligning with our adoption of trunk-based development. This mode of operation facilitated a more streamlined and continuous integration of changes, enhancing our development agility. To complement this, we engaged GitHub Actions for our continuous integration (CI) operations, which allowed us to automate our build and testing processes efficiently. This integration with GitHub Actions ensured that every commit was built and tested automatically, reducing the likelihood of integration issues and ensuring code quality.

```
.
├── BUILD.bazel
├── README.md
├── WORKSPACE
├── commons
│   ├── aws-dynamodb
│   ├── aws-lambdas
│   ├── aws-s3
│   ├── aws-utils
│   ├── core
│   ├── external-callers
│   ├── infra
│   ├── logging-service
│   ├── parameter-service
│   ├── parameters
│   └── smartcontracts
└── dockerfile
```

```
|— docs
|   |— DevEnvironment.md
|   |— GettingStarted.md
|   |— GettingStarted_old.md
|   |— Images
|   |— LineSeparatorGuide.md
|   |— dev.md
|   └─ dotfiles.md
|— domains
|   |— accounts
|   |— cards
|   |— commonfnc
|   |— customer-care
|   |— data-astrum
|   |— data-ingestor
|   |— data-sendevent
|   |— onboarding
|   |— sandbox
|   └─ security
└─ tools
    |— clu
    |— docker
    |— inception_sdk
    |— python
    |— renderer
    |— requirements.txt
    |— rules
    |— smartcontracts_simulation_tests
    └─ test_utils
```

Our repository was structured to optimize the use of Bazel, starting with the root level containing essential Bazel files like `BUILD.bazel`, `README.md`, and `WORKSPACE`. These files are critical for defining build rules and setting up the Bazel workspace, providing clear guidance for developers.

The ``commons`` directory was a key component of our structure, encompassing various AWS services like DynamoDB, Lambda, S3, and utility libraries. This modular approach allowed for better management and isolation of common functionalities, such as logging services, parameter handling, and smart contract integrations.

Documentation was another critical aspect, housed in the ``docs`` directory. This included comprehensive guides for setting up development environments, getting started instructions, and best practices, ensuring that team members had easy access to the information they needed to work effectively.

Our ``domains`` folder represented the core business logic, segregated into domains like accounts, cards, customer care, and security. This separation into domains allowed teams to focus on specific areas of the application without interference, reducing complexities in understanding and maintaining the codebase.

Finally, the ``tools`` directory housed a variety of utilities and scripts, including custom Bazel rules, test utilities, and SDKs. This centralization of tools ensured consistency in development practices and streamlined the process of updating and maintaining these utilities.

## **Challenges and Solutions**

During the implementation of Bazel, our project encountered a range of specific challenges that required innovative solutions. One major issue was dealing with the serverless packaging process, which necessitated the implementation of tree shaking techniques to reduce the artifact size. This was crucial for optimizing performance and efficiency. Another challenge involved updating packages for cloud-native services, which required careful handling to ensure compatibility and stability. Similarly, we

faced the task of building and packaging custom code for TM Vault, our core banking application, which presented its own unique set of complexities.

To address these challenges, we focused on automating various processes using Bazel rules, enhancing the developer experience without obscuring understanding. It was vital that these processes were transparent to developers, avoiding the creation of “magic black boxes” that could hinder comprehension and troubleshooting. Our approach was to provide tools and rules that were accessible and understandable to all team members, ensuring they were fully equipped to manage and contribute to these automated processes effectively.

In terms of solutions and workarounds, we developed a suite of custom rules and tools tailored to our project’s needs. These included managing Python dependencies, which was fundamental for our serverless functions and other Python-based components. We automated the building, packaging, deploying, and testing of AWS Lambda functions, streamlining our serverless architecture. Building and deploying AWS infrastructure became more efficient with custom Bazel integrations. We also implemented solutions for executing AWS Systems Manager commands, which were essential for maintaining our cloud infrastructure.

For our Java applications, we developed optimizations and handled build dependencies and execution challenges, ensuring our Java components were efficient and reliable. We also created a process for updating environment variables specifically for Bazel, enhancing the flexibility and adaptability of our build environment.

Likewise, we tackled the management of smart contract deployment, validation, module handling, and code generation. This was particularly important for our blockchain-related components, where precision and



reliability are paramount. Finally, we implemented a system for generating client code from Swagger definitions, which streamlined our API development and ensured consistency across our services.

## **Results and Impact**

The adoption of Bazel in our project brought about both quantitative and qualitative benefits that markedly improved our development process. Most notably, there was an 81% improvement in build times, which significantly expedited the development cycle. In terms of productivity, our full-lifecycle developers experienced a 28% enhancement, a testament to the efficiency and effectiveness of Bazel in managing complex workflows. This improvement in productivity also extended to the scalability of our codebase, ensuring its maintainability over time. As well as, feedback gathered through developer polls highlighted an improved developer experience, further validating the positive impact of Bazel on our project's development environment.

An unexpected yet valuable outcome of this transition was the blurring of traditional boundaries between front-end and back-end development. Developers, who previously specialized in one area, found themselves gaining insights and contributing to the other side of the codebase. This cross-functional collaboration not only enriched their skill sets but also fostered a more integrated and cohesive team dynamic. Furthermore, Bazel simplified the update and release processes of our cloud-native services. By managing infrastructure as code within Bazel projects, we streamlined and unified our approach to deploying and maintaining our cloud infrastructure, leading to more efficient and error-free operations. These unexpected benefits underscored the versatility and far-reaching impact of Bazel, extending

beyond just build and test processes to encompass broader aspects of software development and deployment.

## **Lessons Learned**

The experience of implementing and using Bazel in our project brought forth several key takeaways. One significant realization was the effectiveness of managing infrastructure as code within Bazel projects. This process streamlined our deployment processes and enhanced the reproducibility of our environments. Plus, we successfully managed cloud-native services configurations as code, integrating them seamlessly into our Bazel projects. This integration provided a unified framework for both application and infrastructure management. Furthermore, adopting an API-First approach, we managed asynchronous APIs as code, which reinforced a consistent and forward-thinking development practice, ensuring that API design and implementation were thoroughly aligned.

From this journey, we gained valuable insights that could benefit similar projects or teams. We learned that while it is advantageous to reuse existing community rules for common operations, developing custom rules for specific, unique requirements of a project is crucial. This approach ensures that the tooling and processes are perfectly tailored to the project's needs. Equally important, we discovered that optimizing Bazel is an ongoing process that requires maturity and a consistent investment of engineering resources over time. Another critical insight was the importance of decoupling deployment operations from the building process. By focusing solely on the building process, we could streamline and optimize this phase without the added complexity of deployment considerations.

For others considering adopting Bazel, we recommend a few strategic approaches. Firstly, it is highly beneficial to hire at

least one engineer with deep knowledge and experience in Bazel. This expertise is invaluable in navigating the complexities of Bazel and guiding the project towards effective implementation. Secondly, it is crucial to onboard the entire development team in the principles and basic usage of Bazel. Ensuring that the team has a foundational understanding of how Bazel works will facilitate smoother integration and collaboration. Lastly, providing formal training for DevOps teams is essential. This training should focus not just on the technical aspects of Bazel but also on how it can be leveraged to enhance development and operational workflows. Such an investment in knowledge and skills will pay dividends in the successful adoption and utilization of Bazel in the project.

## **Future Plans and Considerations**

As we continue to evolve and enhance our use of Bazel, one of our key ongoing initiatives involves integrating the Quality Assurance (QA) team more deeply into our automated processes. We are in the process of incorporating automated user acceptance tests into our Bazel workflows. This integration not only aligns with our commitment to quality and efficiency but also involves setting up the necessary infrastructure to provide a robust QA environment. In doing this, we aim to streamline our testing processes and ensure that our products meet the highest standards of quality before they reach the end-users.

In terms of expanding the use of Bazel within our organization, we are planning an internal engineering session dedicated to sharing our experiences with Bazel. This session is designed to be a show-and-tell event for other teams within the organization who are interested in understanding Bazel's capabilities and considering its adoption. By doing this, we hope to foster a knowledge-

sharing culture and assist other teams in realizing the potential benefits of integrating Bazel into their workflows.

Looking at the long-term vision for Bazel within our project and organization, we are increasingly considering the adoption of Bazel and Monorepo strategies for existing projects, particularly those within modernization programs. This consideration stems from the positive impact we have observed in our current projects, where Bazel has significantly improved our development processes. The move towards Bazel and Monorepos for brownfield projects represents a strategic step in modernizing our technology stack, enhancing collaboration, and streamlining our build and deployment processes across various initiatives.

## **Outcome**

Bazel's ability to handle multiple languages and platforms, along with its integration with various testing frameworks, proved crucial in managing the project's intricate codebase and diverse technology stack. The implementation of Bazel was driven by the goal to expedite the development cycle and enhance full-stack collaboration, thereby aligning with the project's stringent timeline and quality standards.

Reflecting on Bazel's contribution to the project, it becomes evident that its capacity for scalable builds, fine-grained dependency management, and performance optimization were key factors in the project's success. Bazel not only facilitated a significant reduction in build times and an increase in developer productivity but also fostered an environment conducive to agile development practices. This resulted in a more responsive development process, enabling the team to meet the tight deadline and deliver a high-quality product. Bazel's role extended beyond just improving build and test processes; it was instrumental in transforming the development workflow, ensuring

consistency, and maintaining code quality across a large and dynamic team.

The broader implications of this case study for the software development community are profound. It underscores the transformative potential of adopting tools like Bazel, particularly in complex, large-scale projects. The experience shared here demonstrates the benefits of aligning build tools with project goals and team dynamics, and it highlights the importance of choosing a tool that not only addresses immediate technical needs but also supports long-term strategic objectives. This case study serves as a valuable reference for other organizations considering a similar transition, offering insights into the challenges, solutions, and best practices for implementing Bazel in a demanding and fast-paced development environment.

### **Case Study 3: Using Bazel in a Developer Hub**

This case study examines a central European private bank deeply engaged in a modernization program. Faced with limitations in the local software engineering market, including high costs and hiring restrictions, the bank decided to establish a near-shore developer hub. This hub was strategically created to build the next generation of enterprise software, addressing the challenges in resource availability and expertise.

A critical success factor for this developer hub was the ability to standardize methods and tools across various software products. This standardization was essential for efficiently scaling the team size over time, consolidating talent within the hub, and extending their capabilities across different products. The operational model of the hub was grounded in agile principles and team topologies, fostering a flexible and responsive development environment.

The developer hub embarked on a multi-year roadmap, aiming to modernize over 50 systems. This ambitious project involved scaling the team to 150 developers and incorporating external collaborators, presenting a unique set of logistical and managerial challenges.

Before the introduction of Bazel, the hub faced several initial challenges. One of the primary issues was ensuring that all developers were aligned and working cohesively towards common goals. The need to let business features drive code contributions was paramount, as was the ability to support ephemeral environments. Additionally, there was a requirement to provide support for both cloud and on-premise infrastructures. This infrastructure needed to adhere to the principles of infrastructure as code, utilizing tools like Helm and Kubernetes for automated code generation. These challenges highlighted the need for a robust and versatile tool that could unify the development process across the various projects and teams within the hub.

## **Motivation for Adopting Bazel**

Our decision to adopt Bazel as our primary build tool was influenced by several key factors that aligned with the needs and goals of our project. The vibrant community around Bazel was a significant draw, offering a wealth of knowledge and support which is invaluable for any developing project. Furthermore, the experience of our team members with Bazel played a crucial role in this choice; their familiarity with its functionalities and best practices promised a smoother integration into our workflow. Moreover, the scalability offered by Bazel, crucial for managing our growing codebase, alongside its robust multi-language support, made it an ideal candidate. These features of Bazel ensured that it could efficiently handle the

diverse technologies and the increasing complexity of our project.

The goals and expected outcomes of implementing Bazel were centered around enhancing the overall developer experience. We aimed to simplify our tooling ecosystem, reducing the overhead associated with managing multiple build systems. Bazel's ability to homogenize configurations across different environments was another key motivation, as it promised to streamline our development processes. Moreover, we sought to leverage Bazel's capabilities to share common architectural components more effectively, fostering a more cohesive and modular codebase. The implementation of Bazel was envisioned not just as a technical upgrade but as a strategic move towards a more integrated, efficient, and developer-friendly environment.

## **Implementation Strategy**

The integration of Bazel into our existing development workflow was strategically planned and executed, focusing on enhancing our development practices while introducing new efficiencies. We adopted a trunk-based development approach, which streamlined our development process and reduced the complexity of managing multiple long-lived branches. Alongside this, we implemented GitOps practices, which further aligned our development and operational workflows. To maintain consistency and clarity in our version control system, we enforced specific patterns for branch naming and commit descriptions. This standardization not only made our repository more organized but also facilitated easier navigation and understanding of the code changes.

Another key consideration during the implementation was to bring a higher level of code traceability into our processes. This was achieved by linking branch names directly with user story IDs in Jira. This connection between the code and

the tracking system ensured a clear and traceable path from requirements to implementation, enhancing accountability and transparency in the development process. Besides, we adopted the practice of engaging pull request commit descriptions with conventional commit standards. This practice was instrumental in automatically generating comprehensive and informative changelogs. The conventional commit approach allowed us to maintain a clear history of changes, simplifying the process of tracking modifications and understanding the evolution of the codebase over time. These strategic decisions in the implementation of Bazel not only streamlined our development workflow but also significantly improved the management and traceability of our code.

The chosen structure was designed to maximize efficiency and maintainability while ensuring ease of navigation and coherence throughout the codebase.

```
.
├── BUILD.bazel
├── MODULE.bazel
├── README.md
├── WORKSPACE.bazel
├── apps
│   ├── app-1
│   ├── app-2
│   ├── dev-portal
│   └── app-3
├── babel.config.json
├── catalog-info.yaml
├── commons
│   ├── backend
│   ├── catalog
│   ├── frontend
│   ├── infra
│   └── metadata
```



```
|— docs
|   |— architecture-decisions
|   |— code-contribution-model
|   |— images
|   |— local-development
|   └─ testdata
|— domains
|   |— domain-1
|   |— domain-2
|   |— domain-3
|   |— domain-4
|   └─ domain-5
|— jest.config.ts
|— jest.preset.js
|— maven_install.json
|— nx.json
|— openapitools.json
|— package.json
|— pnpm-lock.yaml
|— pnpm-workspace.yaml
|— project.json
|— tools
|   |— rules
|   |— scripts
|   |— tsconfig.tools.json
|   └─ validators
└─ tsconfig.base.json
```

At the root level, we placed key Bazel and project configuration files such as `BUILD.bazel`, `MODULE.bazel`, `WORKSPACE.bazel`, and `README.md`. These files are essential for Bazel's operation, providing a clear entry point and guidelines for new developers. The `apps` directory was created to house individual applications (`app-1`, `app-2`, `dev-portal`, `app-3`), allowing for clear separation and

modular management of each application within the same repository.

In the ``commons`` folder, we segregated shared resources into logical subcategories like ``backend``, ``frontend``, ``infra``, and ``metadata``. This organization aids in the reuse of common code and resources across different applications, promoting consistency and reducing redundancy.

The ``docs`` directory contains essential documentation, including ``architecture-decisions``, ``code-contribution-model``, and guides for ``local-development``. This centralized documentation ensures that team members have easy access to important project information and guidelines.

We introduced a ``domains`` directory, starting with ``domain-1``, to encapsulate domain-specific logic, making it easier to manage and evolve these aspects independently.

Configuration files such as ``babel.config.json``, ``jest.config.ts``, and ``pnpm-workspace.yaml`` are placed at the root to apply consistent settings across the entire Monorepo. This method simplifies configuration management and ensures that all apps and packages within the repo adhere to the same standards.

Lastly, the ``tools`` directory is designated for custom tools and scripts like ``rules``, ``validators``, and ``scripts``. This separation of tools from the main application code helps in maintaining a clean and organized codebase.

Overall, this layout was designed with the intent of leveraging Bazel's capabilities to the fullest, ensuring a scalable and manageable codebase while accommodating the diverse aspects of our project. The structure facilitates easy navigation, efficient resource sharing, and clear separation of concerns, all of which are crucial for the smooth operation and growth of a large-scale project managed in a Monorepo setup.

## **Challenges and Solutions**

The implementation of Bazel in our project brought with it specific challenges, the most significant of which was aligning different teams across the company around the Monorepo and Bazel approach. This alignment was crucial, as it involved not only a change in tools but also a shift in the development culture and workflow. The challenge lay in bringing various teams, each with their unique processes and priorities, onto a common platform. This required a thoughtful approach to ensure that the transition was smooth and that all teams were on board with the new system.

To address these challenges, we established two key governance structures—the Engineering Solutions Board and the Architectural Review Board. The Engineering Solutions Board was tasked with identifying and designing the best solutions for the engineering challenges we faced during the implementation. This board played a crucial role in troubleshooting, innovating, and refining our approach to integrating Bazel. Meanwhile, the Architectural Review Board was instrumental in decision-making and consensus management. This board ensured that decisions regarding architecture and tooling were made collaboratively and that all teams had a voice in the process. Their role was vital in maintaining harmony and alignment among the various teams during the transition to Bazel and the Monorepo setup.

In addition to these governance structures, we developed specific solutions and workarounds to overcome the limitations we encountered with Bazel, particularly in managing the Next.js building process. To tackle this, we integrated Narwhal building tools into Bazel, which significantly improved our handling of the frontend. This integration exemplifies our approach to addressing the

challenges faced — by being open to external tools and custom solutions, we were able to effectively complement Bazel's capabilities and ensure a more seamless development process. The combination of strategic governance and technical workarounds thus played a pivotal role in successfully implementing Bazel in our project.

## **Results and Impact**

The adoption of Bazel in our project, while not quantitatively measured in precise metrics, brought about significant qualitative improvements as per the feedback received from our team. The most notable improvements were seen in build times, which became noticeably faster, enhancing the overall development cycle. This acceleration in build processes directly contributed to increased developer productivity, as team members were able to spend more time on development rather than waiting for builds to complete. Also, the scalability of our codebase saw substantial improvement, which, coupled with enhanced quality and maintainability, positioned our project on a trajectory for sustained growth and evolution. These changes, largely attributed to the integration of Bazel, underscored the tool's impact in streamlining and optimizing our development processes.

An unexpected yet highly beneficial outcome of adopting Bazel was the evolution of our team dynamics, particularly between the frontend and backend teams. Previously operating in more siloed environments, the teams began to engage more collaboratively in debugging and problem-solving across the full stack. This cross-functional interaction led to a deeper understanding of the entire system, with frontend and backend developers contributing to areas beyond their traditional scopes. This bridging of roles not only enhanced our team's versatility and skill set but also

fostered a more cohesive and collaborative work environment. The adoption of Bazel, thus, transcended its primary role as a build tool and became a catalyst for a more integrated and efficient team structure.

## **Lessons Learned**

Throughout the process of implementing and using Bazel, several key takeaways emerged that underscored the importance of strong leadership across the organization. Despite the segregation of responsibilities, which is a common practice in many projects, we learned that having a unified leadership approach was crucial. This was particularly evident in the strategic decision to enable various boards and committees, which played a pivotal role in guiding the project. Their involvement was not just about oversight; it was about providing direction, resolving conflicts, and ensuring that all teams were aligned with the project's goals and objectives. This tactic proved to be instrumental in the successful integration and utilization of Bazel in our project.

In addition, our experience highlighted the significance of assessing and managing the corporate engineering culture, especially when deploying a Monorepo approach and introducing Bazel as a build tool. The shift to a Monorepo and the adoption of Bazel require a cultural change that goes beyond mere technical implementation. It involves rethinking how teams collaborate, how code is managed, and how builds are handled. For projects or teams considering a similar transition, one crucial piece of advice is to define formal operations before implementation. While this may seem basic, it is a complex task in practice and is essential for ensuring a smooth transition. Clear operational guidelines help in aligning everyone's efforts, avoiding misunderstandings, and setting a strong foundation for the effective use of Bazel.

## **Future Plans and Considerations**

Our future plans for Bazel usage in the project involve significant enhancements, particularly in the management of external dependencies. We aim to refine how we handle both consumers and providers of these dependencies. This enhancement is expected to streamline our development process further, making it more efficient and less prone to errors related to dependency issues. By improving this aspect of our Bazel usage, we hope to achieve a more seamless integration of various components and libraries, enhancing the overall robustness and reliability of our builds.

In terms of expanding the use of Bazel, we are looking forward to extending its application beyond our current project scope. Other projects within the organization are set to be coached through the process of Bazel adoption. This initiative is driven by the success we have experienced so far and the belief that Bazel's strengths in managing large and complex codebases can be beneficial across different teams and projects. By sharing our learnings and best practices, we aim to facilitate a smoother transition for these projects, helping them to leverage Bazel's capabilities effectively.

Looking at the long-term vision, we are committed to fully embracing Bazel within our organization. A key aspect of this vision is encouraging each business unit to create and manage their own Monorepos in the medium term. Over time, we plan to progressively consolidate these into a more unified structure. In addition, we are enthusiastic about contributing to the broader Bazel community. Plans include open sourcing our custom Bazel rules and actively contributing content to technical blogs and talks. Through these efforts, we aim not only to share our knowledge and experiences but also to attract external talent and foster a

collaborative environment around Bazel usage. This long-term strategy aligns with our goal of continuous improvement and innovation in our development practices.

## **Outcome**

Faced with challenges in aligning developers and managing a complex infrastructure that included cloud and on-premise elements, the Developer Hub adopted Bazel. This decision was driven by Bazel's vibrant community, scalability, and multi-language support, which were crucial for the project's diverse technology needs. The implementation of Bazel aimed to streamline tooling, harmonize configurations, and foster a cohesive codebase, using a trunk-based development (see <https://trunkbaseddevelopment.com/>) approach and GitOps practices for improved code traceability and management.

The integration of Bazel into the hub's workflow marked a significant turning point. The tool's impact was profound, notably in accelerating build processes, which, in turn, boosted developer productivity and enhanced the scalability of the codebase. Bazel's role extended beyond a mere technical solution; it acted as a catalyst for team collaboration, breaking down silos between front-end and back-end teams and fostering a unified, agile development culture. The success of Bazel in this context was not just in its technical prowess but in how it reshaped the team dynamics and development practices, aligning them with the project's ambitious goals.

The lessons learned from this case study offer valuable insights for the wider software development community. Strong, unified leadership and a thorough understanding of the engineering culture are essential in navigating the transition to tools like Bazel and approaches like Monorepos. These insights emphasize the importance of not just the

technical aspects but also the cultural and operational shifts required for successful implementation. The project's future plans to refine dependency management with Bazel, mentor other projects in its adoption, and contribute to the Bazel community, highlight a commitment to continuous improvement and knowledge sharing. This approach serves as a model for other organizations embarking on similar journeys, underscoring the transformative potential of tools like Bazel in large-scale software development projects.

## **Conclusion**

The first case study, exemplifies a significant leap in digital banking technology, offering a rapid, scalable, and flexible cloud-based solution. The project's success, marked by rapid deployment, modularity, and cloud-based flexibility, underscores the profound impact of leveraging advanced technology in financial services. The integration with AWS and use of Vault and Mambu engines highlight the importance of robust and secure infrastructure in modern banking operations. The adoption of a comprehensive system for new market ventures demonstrates the potential to revolutionize digital banking by significantly reducing time to market and operational costs, while offering a customizable and efficient banking platform.

The second case study showcases the transformative potential of serverless architecture in developing a financial services platform for a new market. The project's ability to integrate services from various providers into a comprehensive suite of solutions illustrates the power of cloud-native services in creating a user-friendly and efficient platform. The strategic use of Bazel significantly enhanced the development process, leading to an 81% improvement in build times and a 28% increase in developer productivity. This case study serves as a testament to the agility and scalability afforded by serverless computing and the pivotal



role of tools like Bazel in managing complex, large-scale software development projects.

The final case study from a central European private bank's developer hub brings to light the challenges and rewards of modernizing enterprise software. The implementation of Bazel facilitated a unified approach to software development, breaking down traditional front-end and back-end barriers, and fostering a more collaborative and efficient environment. The hub's journey illustrates the critical role of standardization, agile principles, and the right tooling in scaling and enhancing software development processes. The significant improvements in build times and productivity achieved through Bazel's integration highlight its effectiveness in large-scale development projects, emphasizing the importance of both technical prowess and cultural adaptation in software modernization initiatives.

Across these diverse case studies, Bazel emerges as a unifying and transformative force in software development, adept at handling various languages, platforms, and complex project structures. Its impact goes beyond technical efficiencies, influencing team dynamics, development culture, and operational practices. These case studies collectively demonstrate the versatility and power of Bazel in different contexts, offering valuable insights and lessons for the broader software development community. The ongoing commitment to leveraging Bazel's full potential and sharing knowledge within and beyond individual projects signifies a broader shift towards more integrated, agile, and efficient software development practices in the digital era.

# **CHAPTER 14**

## **Future Trends and Considerations**

### **Introduction**

In this chapter, we will explore the forefront of development practices and technological integrations that are shaping the future of software builds. We will examine the evolving landscape of Monorepo development, emphasizing how AI and machine learning are being incorporated into Bazel builds to optimize efficiency and decision-making. The chapter also navigates through the intricacies of enhanced remote caching and execution strategies, highlighting their impact on build performance. A critical analysis of advanced dependency management techniques is presented, alongside an exploration of the latest security enhancements vital for robust Monorepo infrastructure. This discussion extends into the realms of fostering collaboration through improved code review and integration practices, and how sustainable development practices are being embedded in Monorepo management. We then venture into the future of Bazel and Monorepo tooling, including a comprehensive overview of the key advances in Bazel 7, the modular approach of Bzlmod, and the exciting planned features for Bazel versions 7 and beyond. The chapter not only anticipates the trajectory towards Bazel 8 and 9 but also discusses the broader implications for Monorepo tooling and the considerations necessary for future development. As we anticipate challenges and adapt strategies, topics like migration to Bzlmod, enhanced performance, caching,

expanded language support, a robust extension model, and adapting to continuous updates are critically analyzed. Finally, practical guidance on migrating existing projects from Bazel 6 is provided, ensuring readers are well-equipped to navigate the evolving landscape of software development with Bazel.

## **Structure**

In this chapter, we will discuss the following topics:

- Evolving Practices in Monorepo Development
- The Road Ahead for Bazel and Monorepo Tooling
- Anticipating Challenges and Adapting Strategies
- How to Migrate Existing Projects

## **Evolving Practices in Monorepo Development**

As we delve into the evolving landscape of Monorepo development, particularly through the lens of Bazel, it is imperative to acknowledge the rapidly changing dynamics of software engineering. This section focuses on advanced topics, derived from the most recent internet publications as of 2023, which are instrumental for software developers seeking to stay ahead in their Monorepo management journey.

## **Integrating AI and Machine Learning in Bazel Builds**

One of the most significant trends in Monorepo development is the integration of artificial intelligence (AI) and machine learning (ML) into the build process. This approach aims to optimize build times and resource allocation. With Bazel's

ability to handle large, interconnected codebases, incorporating AI can lead to predictive build analytics. Such systems can anticipate build failures, suggest optimal build paths, and efficiently allocate resources based on historical data patterns.

## **Enhanced Remote Caching and Execution Strategies**

As Monorepos continue to grow, remote caching and execution emerge as critical components for scalability. Advanced strategies in 2023 focus on optimizing these aspects to reduce build times drastically. Techniques such as layered caching, where dependencies are cached in layers, and prioritized artifact retrieval, where frequently accessed artifacts are retrieved faster, are becoming mainstream. Additionally, adaptive execution strategies, which adjust resource allocation based on real-time system load and build complexity, are also gaining traction.

## **Advanced Dependency Management Techniques**

While dependency management in Bazel is not a new topic, the methods are continuously evolving. The recent trend is towards more dynamic and intelligent dependency resolution mechanisms. These involve analyzing codebase changes in real-time to update dependencies, thereby ensuring that builds are always using the most efficient and relevant set of dependencies. This approach reduces redundancy and enhances build performance.

## **Security Enhancements in Monorepo Infrastructure**

As the complexity and size of Monorepos increase, so do the security challenges. Advanced security practices in 2023 revolve around automated vulnerability scanning and patching within the Bazel build pipeline. Integrating security at the build level ensures that every component of the Monorepo is continuously assessed for vulnerabilities, with patches automatically applied, ensuring a robust and secure codebase.

## **Fostering Collaboration through Enhanced Code Review and Integration Practices**

Collaboration in large Monorepos can be challenging due to the sheer size and complexity of the codebase. The future trend is towards integrating more sophisticated code review and integration tools directly into the Bazel workflow. These tools are designed to handle large-scale changes efficiently, providing developers with contextual insights and suggestions, thereby streamlining the review process and reducing integration conflicts.

## **Sustainable Development Practices in Monorepo Management**

Sustainability in software development is becoming a crucial consideration. In Monorepo development, this translates to practices that ensure long-term maintainability and scalability of the codebase. This includes modularization strategies, where the Monorepo is structured into well-defined, maintainable modules, and the adoption of green coding practices, where builds are optimized for energy efficiency.

As we look towards the future, it is clear that the practices in Monorepo development, particularly in environments

managed by Bazel, are rapidly evolving. The integration of AI, advanced caching techniques, dynamic dependency management, robust security measures, collaborative tools, and sustainable practices are not just trends but necessities for managing the complex software development landscapes of tomorrow. Embracing these advanced methodologies will not only streamline development processes but also ensure that the Monorepos remain scalable, secure, and efficient in the face of ever-growing demands.

## [The Road Ahead for Bazel and Monorepo Tooling](#)

As an advanced software developer well-versed in Bazel, Monorepos, and dependency management in Bazel, it is crucial to understand the emerging trends and considerations that will shape the future of these technologies.

The journey of Bazel, particularly with its latest major release, Bazel 7, and the projected path towards Bazel 8 and 9, indicates a significant evolution in its capabilities and functionalities.

### **Key Advances in Bazel 7**

- **Performance Improvements:** Bazel 7 introduces significant performance enhancements, primarily through optimized dependency analysis and build execution, leading to reduced build times and increased efficiency.
- **Expanded Language and Platform Support:** The update broadens the scope of Bazel, accommodating more programming languages and platforms.
- **Enhanced Caching and Remote Execution:** Advanced caching mechanisms and remote execution

capabilities in Bazel 7 further improve build speed and consistency.

- **Robust Extension Model:** The extension model has been made more robust, offering greater customization and flexibility in the build process.

## Bazel's Modular Dependency Management: Bzlmod

A key feature in Bazel 7 and the future releases is **Bzlmod**, Bazel's new modular external dependency management system. **Bzlmod** is now enabled by default, replacing the older **WORKSPACE** mechanism. This system enhances the scalability and reliability of managing transitive dependencies.

## Planned Features for Bazel 7 and Beyond

The planned features for Bazel 7 and beyond highlight a significant evolution in its capabilities, focusing on enhancing performance, scalability, and user experience. These features not only strengthen Bazel's role in build automation but also emphasize its adaptability to future development needs.

- **Bzlmod - Modular Dependency Management:** A standout feature in Bazel 7 is **Bzlmod**, a new modular external dependency management system. It replaces the older **WORKSPACE** mechanism and aims to improve the scalability and reliability of managing transitive dependencies.
- **Repository Cache:** Bazel 7 plans to introduce a true repository cache. This feature is intended to cache the extracted contents of fetched repositories, potentially

allowing different workspaces on the same machine to share extracted repos.

- **Offline and Vendor Mode:** This mode will enable developers to prefetch all necessary external repositories for offline work. The fetched repos can optionally be incorporated into the source tree, effectively vendoring them.
- **REPO.bazel File:** This new file type will replace `WORKSPACE` as the repository boundary marker, in addition to the `MODULE.bazel` file. It will allow the specification of repository-wide common attributes.
- **Performance Improvements:** Bazel 7 boasts significant performance improvements, particularly in terms of optimized dependency analysis and build execution, leading to reduced build times.
- **Enhanced Support for New Languages and Platforms:** The update broadens Bazel's support for several new programming languages and platforms.
- **Advanced Caching and Remote Execution:** Enhanced caching mechanisms and remote execution capabilities further improve build speed and consistency.

### **Beyond Bazel 7:**

- **Build Without the Bytes (BwoB):** This feature, aimed at improving remote build performance, will be further optimized. Bazel will download only the outputs of requested top-level targets from the remote server, reducing unnecessary data transfer.
- **Skymeld Project:** This project focuses on improving multi-target build performance by merging the analysis and execution phases, allowing targets to be executed as soon as their analysis is completed.



- **Platform-based Toolchain Resolution for Android and C++:** This update aims to streamline the toolchain resolution API across all rulesets and remove technical debt by having Android and C++ rules use the same toolchain resolution logic as other rulesets.
- **Continuous Enhancement in Performance and Scalability:** As Bazel evolves, ongoing efforts will be made to further enhance its performance, especially in handling large and complex projects.

These features represent a forward-thinking approach by the Bazel team, addressing current challenges and preparing for future developments in software build automation. The emphasis on modular dependency management, performance optimization, and support for diverse languages and platforms underlines Bazel's commitment to evolving alongside the changing landscape of software development.

For more detailed information on these developments, you can visit:

- Google Open Source Blog - <https://opensource.googleblog.com>
- Bazel Blog - <https://blog.bazel.build>
- Bazel Roadmap - <https://bazel.build>

## **The Future Path: Bazel 8 and 9**

The future of Bazel, particularly with the upcoming Bazel 8 and 9 releases, is marked by significant changes and enhancements, primarily focused on further refining the build system's functionality and user experience. These changes are not only indicative of Bazel's continuous evolution but also reflect the broader trends and demands in software build and dependency management.

- **Integration and Default Enabling of Bzlmod:** One of the most notable changes will be the further integration and default enabling of Bzlmod, the modular external dependency management system introduced in Bazel 7. This system is set to become the standard for dependency management in Bazel, gradually phasing out the older WORKSPACE mechanism.
- **Phasing Out of WORKSPACE:** By the time Bazel 9 is released, the WORKSPACE file is expected to be completely phased out. This transition underscores Bazel's commitment to more efficient and scalable project management methodologies.
- **Improvements in Remote Execution and Caching:** Bazel 8 and 9 are likely to continue enhancing remote execution capabilities and caching mechanisms. These improvements are aimed at optimizing build performance, particularly for large-scale and complex projects.
- **Platform-based Toolchain Resolution:** The releases are also expected to focus on streamlining toolchain resolution across various languages and platforms, reducing the complexity and potential errors in multi-language, multi-platform builds.
- **Enhanced Language and Platform Support:** Continuing the trend from previous versions, Bazel 8 and 9 will likely expand support for more programming languages and platforms, catering to a broader range of development needs.
- **User Experience and Workflow Improvements:** With each release, Bazel aims to enhance the developer experience. This includes making the system more intuitive and user-friendly, and improving documentation and support resources.

- **Security and Compliance Tools:** Given the increasing focus on software security and compliance, future versions of Bazel might introduce more robust tools for OSS license compliance and security checks.
- **Community-Driven Features:** Bazel's roadmap includes a strong emphasis on community feedback and contributions. Future versions will likely incorporate more features and improvements based on user requests and contributions from the open-source community.

For software developers, particularly those using Monorepos, these advancements imply a need for adaptation and continuous learning. As Bazel evolves, developers will need to stay updated with the latest features and best practices, ensuring that their build pipelines remain efficient, secure, and compliant with the latest standards.

The path towards Bazel 8 and 9 represents an exciting phase in the evolution of this build system, with a strong emphasis on efficiency, scalability, and user experience. As these versions materialize, they will significantly influence the way developers handle large codebases, dependencies, and build processes, marking a new era in software development tooling.

To get updated and more detailed information on Bazel 8 and 9, it is advisable to follow official sources such as the Bazel Blog - <https://blog.bazel.build/>) and the Bazel Roadmap at <https://bazel.build/roadmap.html>

## **Implications for Monorepo Tooling**

The advancements in Bazel, particularly in terms of dependency management and build performance, will significantly influence the way Monorepos are handled. The shift towards a more modular and efficient build system

aligns with the core principles of Monorepo management — centralization, consistency, and streamlined workflows.

- **Transition to New Systems:** The migration to systems like Bzlmod will require careful planning and execution, particularly in large Monorepos where the transition can be complex and time-consuming.
- **Adapting to New Workflow Dynamics:** As Bazel evolves, development teams will need to adapt to the changing dynamics of build and dependency management within Monorepos.
- **Training and Up-skilling:** Keeping up with Bazel's evolving features and best practices will necessitate ongoing training and up-skilling for development teams.

## Considerations for Future Development

The advancements in Bazel and Monorepo tooling reflect a broader trend in the software development industry towards more efficient, scalable, and reliable build systems. As Bazel continues to evolve, it will offer developers new opportunities to optimize their build processes and manage dependencies more effectively, thus shaping the future of software development and Monorepo management.

- **Migration to New Systems:** As Bazel evolves, developers must consider the implications of migrating to newer systems like Bzlmod, weighing the benefits against the transitional challenges.
- **Compliance and Security:** With the introduction of tools for OSS license compliance, developers will need to be more vigilant about the compliance and security of the packages they use.

- **Multi-language, Multi-platform Builds:** The standardized platforms API in Bazel will simplify the architecture configuration for multi-language, multi-platform builds, reducing development-time errors and complexity in large projects.

## Anticipating Challenges and Adapting Strategies

With the release of Bazel 7, developers and organizations using Bazel for their build and test automation need to anticipate potential challenges and adapt their strategies accordingly. Bazel 7 introduces several new features and improvements, which, while beneficial, may require adjustments in existing workflows and systems.

### Migration to Bzlmod

One of the significant changes in Bazel 7 is the introduction of `Bzlmod` as the default module for managing external dependencies, replacing the older `WORKSPACE` system.

#### **Challenges:**

- **Adapting to New Dependency Management:** Developers accustomed to the `WORKSPACE` system need to familiarize themselves with `Bzlmod`'s mechanisms.
- **Migration Overhead:** Transitioning existing projects to `Bzlmod` might require a significant effort, especially for complex builds.
- **Ensuring Compatibility:** Ensuring that all dependencies and tools are compatible with the new system can be a time-consuming process.

#### **Strategies:**

- Gradual Transition: Approach the migration process in stages, starting with smaller, less critical projects.
- Leverage Documentation and Community Support: Utilize available resources like migration guides and community forums for smoother transition.
- Test Thoroughly: Conduct extensive testing during the transition to catch and resolve issues early.

## **Enhanced Performance and Caching**

Bazel 7 boasts improved performance and advanced caching mechanisms.

### **Challenges:**

- Optimizing Build Configurations: The new caching and performance features might require reevaluation and optimization of existing build configurations.
- Resource Management: Enhanced performance features might behave differently in varied hardware and network environments, necessitating careful resource management.

### **Strategies:**

- Performance Benchmarking: Regularly benchmark the performance after upgrading to identify areas of improvement.
- Optimize Hardware and Network Usage: Adjust resource allocation based on the performance characteristics of Bazel 7.

## **Expanded Language and Platform Support**

With broader support for various languages and platforms, Bazel 7 increases its versatility.

## **Challenges:**

- **Integrating Diverse Technologies:** Teams may face challenges integrating new languages and platforms into their existing Bazel workflows.
- **Ensuring Cross-Platform Compatibility:** Maintaining builds that are compatible across multiple platforms can become more complex.

## **Strategies:**

- **Incremental Integration:** Gradually integrate new languages and platforms to manage complexity.
- **Cross-Platform Testing:** Implement comprehensive cross-platform testing to ensure compatibility and performance across different environments.

## **Robust Extension Model**

The more robust extension model in Bazel 7 allows greater customization in the build process.

## **Challenges:**

- **Learning Curve:** The new extension model might have a steep learning curve for developers not familiar with advanced customization.
- **Maintaining Custom Extensions:** Developing and maintaining custom extensions can be resource-intensive.

## **Strategies:**

- **Training and Skill Development:** Invest in training for team members to effectively utilize the new extension model.
- **Collaborate with the Community:** Engage with the Bazel community for shared development and maintenance of common extensions.

## [Adapting to Continuous Updates](#)

Bazel's roadmap includes continuous updates, with future releases like Bazel 8 and 9 already in planning.

### **Challenges:**

- **Keeping Up with Changes:** Continuous updates require teams to stay informed and adapt to changes regularly.
- **Balancing Stability and Upgrades:** Deciding when to upgrade to newer versions while ensuring stability can be challenging.

### **Strategies:**

- **Regular Review of Bazel Roadmap:** Keep track of upcoming changes and plan for them in advance.
- **Balanced Upgrade Policy:** Develop a policy for upgrading that balances the need for new features with the stability of the build environment.

The release of Bazel 7 presents exciting opportunities for enhanced efficiency and flexibility in software builds. However, it also brings challenges that require careful anticipation and strategic adaptation. By understanding these challenges and employing thoughtful strategies, developers and organizations can leverage the full potential of Bazel 7 while minimizing disruption to their workflows.

## [How to Migrate Existing Projects from Bazel 6](#)

Bazel 7.0 LTS comes with a range of new features and backward-incompatible changes. Key highlights include:

- **Adopting `Bzlmod`:** Transitioning to `Bzlmod` involves creating a `MODULE.bazel` file if one does not exist in your project. You need to be aware that even with an empty



`MODULE.bazel` file, Bazel 7 might behave differently due to `Bzlmod` being enabled, especially in label stringification and run files handling.

- **Handling Dependencies:** For dependencies not available in a Bazel registry or those that are not Bazel projects, you can use `use_repo_rule` or module extensions in your `MODULE.bazel` file. This involves implementing a module extension, which can be shared between `WORKSPACE` and `Bzlmod` during the migration.
- **Resolving External Dependency Conflicts:** `Bzlmod` allows the resolution of conflicts in external dependencies, particularly when multiple dependencies require different versions of a shared external resource. Module extensions can be used to select appropriate versions based on the dependency graph.
- **Integrating Third-Party Package Managers:** Utilize module extensions to integrate package managers for specific languages, enhancing rulesets that manage dependencies.
- **Detecting and Registering Toolchains:** When Bazel build rules need to detect available toolchains on your host machine, you can introduce these using module extensions in the `MODULE.bazel` file.

The transition to Bazel 7.0 LTS requires careful planning and execution, especially considering the significant changes in dependency management and build optimization features. As the Bazel ecosystem continues to evolve, it is crucial to stay informed about the upcoming changes and prepare for future versions like Bazel 8 and 9.

## [Recommended Readings](#)

**Fostering Collaboration through Enhanced Code Review and Integration Practices**

- **“Monorepos: Web Development Trends in 2023” by Wingravity:** This article delves into the benefits and challenges of using Monorepos in web development, highlighting the importance of balancing code sharing and modularity. It provides a nuanced view of how Monorepos can enhance development efficiency, code consistency, and project management. - <https://www.wingravity.com/monorepos-web-development-trends-in-2023/>)
- **“Benefits and Challenges of Monorepo Development Practices” by CircleCI:** CircleCI offers a comprehensive overview of the advantages and challenges of Monorepo development, especially in the context of microservices architecture. The article discusses the misconceptions about Monorepos and provides strategies for successfully implementing this approach in development environments. - <https://circleci.com/blog/benefits-and-challenges-of-monorepo-development-practices/>)
- **“Elevating Enterprise Deployment: Introducing an Enhanced Monorepo Experience on Netlify”:** This resource provides insights into using Nx in Monorepo development and differentiates between “*integrated*” and “*package-based*” setups in Monorepos. It is particularly useful for understanding advanced concepts in Monorepo tooling and deployment strategies. - <https://www.netlify.com/blog/2023/03/01/elevating-enterprise-deployment-introducing-an-enhanced-monorepo-experience-on-netlify/>)
- **Microsoft DevBlogs — “Using monorepos to increase velocity during early stages of product development”:** This resource dives into the use of Monorepos in the early stages of product development, focusing on its impact on system architecture and protocols. It highlights how Monorepo-style

development, particularly with tools like Nx, can enhance development speed, facilitate code sharing, and impose constraints on the dependency graph of the repo. This method aligns with sustainable development by promoting efficient and flexible project management. - <https://devblogs.microsoft.com/creatingstartups/using-monorepos-to-increase-velocity-during-early-stages-of-product-development/>)

## **Integrating AI and Machine Learning in Bazel Builds**

- **SpotDraft Blog — “How We Used Bazel to Streamline Our AI Development”**: This article from SpotDraft provides a practical perspective on how Bazel was used to manage dependencies in a machine learning context. It covers the integration of Bazel in a microservices architecture, emphasizing its efficiency in handling Python rules and Docker containers. The piece also discusses the challenges and solutions encountered during the adoption of Bazel, particularly in a startup environment. - <https://www.spotdraft.com/engineering-blog/how-we-used-bazel-to-streamline-our-ai-development>)
- **ai-jobs.net — “Bazel: The Build System for AI/ML and Data Science Projects”**: This comprehensive guide offers an overview of Bazel’s features and its applicability in AI/ML and data science projects. The article explains how Bazel supports scalability, multi-language and multi-platform projects, and provides reproducibility and determinism, which are crucial in AI/ML workflows. It also discusses Bazel’s integration with popular AI/ML frameworks and libraries, making it a valuable resource for professionals in the field. - <https://ai-jobs.net/blog/bazel-explained/>)
- **Arm Community — “Building Bazel and TensorFlow on AArch64”**: This blog post details the

process of building TensorFlow using Bazel on AArch64 architecture. It offers a step-by-step guide on setting up the environment, addressing challenges related to TensorFlow's dependency on Bazel, and provides practical tips for managing resource constraints during the build process. -

<https://community.arm.com/developer/tools-software/oss-platforms/b/blog/posts/building-bazel-and-tensorflow-on-aarch64>)

## Security Enhancements in Monorepo Infrastructure

- **Goldman Sachs Developer Blog — “Monorepos from a Risk Perspective”**: This resource provides an in-depth analysis of security risks associated with Monorepos and offers detailed mitigation techniques. It covers essential topics such as access control, high availability and performance, breaking changes, supply chain protection, and cache poisoning. The article is particularly useful for understanding the complexities of managing security in large-scale Monorepo environments. -

<https://developer.gs.com/blog/posts/monorepos-from-a-risk-perspective>)

- **Bridgecrew Blog — “5 ways to configure a monorepo for DevSecOps efficiency”**: Bridgecrew's article focuses on configuring Monorepos effectively for DevSecOps. It discusses challenges like access control and supply chain attacks, and offers practical solutions such as maintaining a directory structure for mappability, compartmentalizing builds, and leveraging Git's metadata and tagging tools for searchability. This resource is beneficial for organizations looking to balance security concerns with the operational efficiency of Monorepos. - <https://bridgecrew.io/blog/5->

[ways-to-configure-a-monorepo-for-devsecops-efficiency/](#))

- **GitKraken Blog — “Monorepos Explained: Examining the Benefits, Costs, and Tools”:** GitKraken’s blog post provides a comprehensive overview of the benefits and costs of Monorepos, with a focus on security challenges like access control and build times. It also explores essential tools for Monorepos, which can help mitigate some of these security risks. This resource is useful for understanding the broader context of Monorepo adoption and its impact on security practices. - <https://www.gitkraken.com/blog/monorepos-explained>)
- **Thoughtworks - “Monorepo vs. multi-repo: Different strategies for organizing repositories”:** This article discusses the trade-offs between Monorepos and multi-repo strategies. It emphasizes the importance of making the right decision early in project development to prepare for future challenges. While not directly addressing sustainability, it provides a foundational understanding of Monorepo and multi-repo structures, which is essential for implementing sustainable practices. - <https://www.thoughtworks.com/insights/blog/monorepo-vs-multi-repo-different-strategies-organizing-repositories>)

## **Sustainable Development Practices in Monorepo Management**

- **“Developing climate solutions with green software” - MIT Technology Review:** This article discusses the importance of understanding the hardware on which software runs to achieve greater energy efficiency. It emphasizes the role of AI in optimizing software for energy efficiency, highlighting

the potential of AI to suggest more efficient coding practices and to tune hardware for specific software needs. The article also addresses the challenge of measuring the energy efficiency of software, a crucial step in developing greener software solutions. - <https://www.technologyreview.com/2023/04/20/1068143/developing-climate-solutions-with-green-software/>)

- **“Sustainable Software Design: Background and Best Practices” — Fraunhofer IESE:** Fraunhofer IESE provides an extensive discussion on sustainable software design, including the impact of programming languages on energy consumption. It advocates for the careful selection of programming languages based on their energy efficiency, especially for intensively used modules. The article also explores the influence of mobile and cloud computing on sustainable software design and offers practical tips for energy-efficient software development, such as using efficient algorithms, reducing network traffic, and leveraging cloud computing. - [https://www.iese.fraunhofer.de/en/press\\_media/press\\_releases/2023/sustainable\\_software\\_design.html](https://www.iese.fraunhofer.de/en/press_media/press_releases/2023/sustainable_software_design.html))
- **“How to Achieve Sustainable Software Development” — InformIT:** This resource from InformIT outlines the concept of sustainable software development as a means to maintain an optimal development pace over time. It emphasizes the need for keeping the cost of change low and maintaining a manageable number of defects. The article also suggests that sustainable development can lead to rapid response to changes at a lower cost, highlighting the importance of this tactic in staying competitive and proactive in the face of new technologies. - <https://www.informit.com/articles/article.aspx?p=1398617>)

## The Road Ahead for Bazel and Monorepo Tooling

- **Bazel's Official Roadmap:** The Bazel roadmap outlines several key initiatives and predictions for Bazel's future development. The roadmap for 2023 includes plans for the Bazel 7.0 release, which aims to deliver many feature improvements requested by users. Some highlights for Bazel 7.0 include better cross-platform cache sharing, remote execution improvements, and enhancements to Bzlmod, Bazel's external dependency management system. Moreover, Bazel 7.0 is focusing on build analysis metrics and introducing Skymeld, an evaluation mode that reduces the wall time of multi-target builds. To learn more, visit the Bazel Roadmap at <https://bazel.build/roadmap>
- **Developments in Bzlmod:** Since its official launch in Bazel 6.0, Bzlmod has undergone significant improvements, particularly in preparation for Bazel 7.0, where it will be enabled by default. Key changes include lockfile support, a new mod command for inspecting the external dependency graph, and comprehensive documentation updates. These improvements are geared towards making Bzlmod more efficient and user-friendly for managing external dependencies. For more information on these updates and future plans for Bzlmod, see the Bazel Blog at <https://blog.bazel.build/2023/07/24/whats-new-with-bzlmod.html>.
- **The Release of Bazel 7:** Bazel 7, the latest major release on the long-term support (LTS) track, includes several new features and improvements. Notably, Bzlmod is now enabled by default, and Build without the Bytes (BwoB) for builds using remote execution is also enabled by default, enhancing remote build performance. The release also introduces merged analysis and execution (Skymeld) and platform-based

toolchain resolution for Android and C++. The full release notes for Bazel 7 provide a comprehensive overview of all new features and improvements. For the complete details, visit Google Developers Blog at <https://developers.googleblog.com/2023/12/bazel-7-release.html>.

- **Toptal - “A Guide to Monorepos for Front-end Code”**: This guide provides an overview of Monorepo management, focusing on front-end code. It discusses the disadvantages of Monorepos, such as security issues and poor Git performance in large-scale projects, and reviews tools like Bazel, Yarn, and Lerna for managing Monorepos. Understanding these tools and their application in the scenario of a Monorepo setup is crucial for maintaining sustainable and efficient development practices. - <https://www.toptal.com/front-end/guide-to-monorepos>)

## How to Migrate Existing Projects to Bazel

- **Bazel Official Migration Guide**: Bazel’s official documentation offers a detailed migration guide that includes various scenarios like migrating from Maven, Xcode, and CocoaPods. This guide is particularly helpful for understanding the fundamental steps and considerations for migrating different types of projects to Bazel. - <https://bazel.build/migrate>)
- **Bzlmod Migration Guide**: The Bzlmod Migration Guide on Bazel’s website is a useful resource for projects moving to Bzlmod, Bazel’s module system. It provides guidance on fetching external dependencies with module extensions and resolving conflict external dependencies with module extensions. This guide is especially useful for projects that need to handle complex dependencies and want to leverage Bazel’s



latest features. - <https://bazel.build/docs/bzlmod-migration-guide>)

- **Harness Experience with Migrating to Bazel:** Harness offers a practical perspective based on their experience migrating from Maven to Bazel. The article covers their proof of concept, migration strategy, and the challenges they faced, such as managing a large codebase and running all unit tests. This resource provides real-world insights into the process and considerations of migrating to Bazel in a complex project environment. - <https://harness.io/blog/migrating-to-bazel-as-a-build-tool>)

These resources collectively cover a range of scenarios and challenges you may encounter when migrating and upgrading to Bazel, offering both theoretical knowledge and practical experiences to guide you through the process.

## **Conclusion**

In this chapter, we delved into the dynamic and evolving landscape of Monorepo development, emphasizing the integration of cutting-edge technologies such as artificial intelligence and advanced caching strategies within Bazel builds. These advancements are pivotal in redefining the efficiency and effectiveness of build processes, particularly in the context of large, interconnected codebases. The exploration of AI's role in predictive analytics, the sophistication of remote caching, and execution strategies, alongside the continuous evolution of dependency management techniques, underscores a forward-moving trajectory in software build and management practices. Moreover, the incorporation of robust security measures and the emphasis on sustainable and collaborative development practices highlight a holistic approach to Monorepo

management, ensuring scalability, security, and maintainability.

Looking ahead, the roadmap for Bazel, particularly with its transition to versions 7 and beyond, presents a landscape rife with enhancements and new features aimed at optimizing performance, broadening language and platform support, and introducing a more modular and efficient approach to dependency management. The transition to Bzlmod as a core component of Bazel's dependency management system signifies a shift towards more scalable and maintainable build processes. This evolution, coupled with improvements in caching, remote execution, and toolchain resolution, illustrates a commitment to addressing the growing complexities and demands of modern software development, ensuring that Bazel remains at the forefront of build automation technologies.

As we anticipate the future, the interplay between Monorepo management and Bazel's tooling evolution beckons a proactive approach from developers and organizations. Adapting to these advancements requires not only technical acumen but also a strategic perspective to navigate the migration to new systems, optimize workflows, and leverage the full spectrum of features offered by Bazel. With continuous updates on the horizon, the imperative to stay informed and agile in adapting to new versions and features is paramount. By embracing these changes and preparing for the challenges they present, developers can harness the potential of Bazel and Monorepo tooling to foster more efficient, scalable, and collaborative software development environments.

# APPENDIX A

## Bazel Cheat Sheet

### Quick Reference Guide to Bazel Terminology

This guide is tailored for software developers who are either new to Bazel or looking to deepen their understanding of its concepts. The terms are presented in an easy-to-understand format, ensuring you can quickly grasp the essentials of Bazel.

- **Action:** A command line invocation and the set of inputs and outputs associated with it, executed by Bazel during the build.
- **Aspect:** A mechanism in Bazel for cross-cutting concerns affecting multiple targets, like code generation or linting.
- **Bazel Server:** A long-running process that improves build performance by keeping some data in memory between builds.
- **Bazel:** A powerful build tool developed by Google, used for building and testing software of any size, quickly and reliably.
- **Bazelisk:** A wrapper for Bazel that manages multiple versions of Bazel.
- **.bazelrc File:** A configuration file for Bazel that allows setting default flags and options.
- **Build Event Protocol (BEP):** A protocol for reporting build events and results to external systems during a Bazel build.

- **BUILD File:** A file that specifies what software artifacts to build using Bazel. It is written in Starlark, Bazel's domain-specific language.
- **BUILD.bazel File:** An alternative to the BUILD file, used to specify build targets and rules.
- **Cache:** Bazel's mechanism for storing and reusing the results of previous builds to speed up subsequent builds.
- **Crosstool:** Configuration for Bazel that defines how to build C/C++ code for different platforms.
- **Dependency Graph:** A directed acyclic graph (DAG) representing dependencies between targets in a Bazel build.
- **Gazelle:** A tool for automatically generating and updating BUILD files for projects in languages like Go.
- **Genrule:** A rule in Bazel for generating one or more files using a shell command.
- **Incremental Build:** Bazel's process of rebuilding only the parts of software that have changed, saving time and resources.
- **Label:** A unique identifier for a target, including the package name and target name, formatted as `//package:target`.
- **Package:** A directory within the workspace that contains a BUILD file, defining a set of targets.
- **Persistent Worker:** A process that stays alive between builds to reduce overhead for certain build actions.
- **Query Language:** Bazel's language for querying the build dependency graph. Useful for understanding build dependencies and structure.
- **Remote Execution:** Bazel's ability to execute builds on remote servers, which can significantly speed up the build process.

- **Repository:** In Bazel, a repository is a collection of files and directories needed for a build, identified by a unique name.
- **Rule:** Specifies how to derive outputs from inputs. Rules are functions in BUILD files that define build targets.
- **Sandboxing:** Isolating build actions to ensure reproducibility and correctness by preventing undeclared dependencies.
- **Skyframe:** The evaluation framework Bazel uses to compute the dependency graph and execute builds.
- **Skylark Rule:** Custom rules defined using the Starlark language for specific build tasks.
- **Starlark:** The language used to write BUILD files and .bzl files. It is a dialect of Python designed for configuration.
- **Target:** A set of source files and build instructions. In Bazel, targets are specified in BUILD files.
- **Toolchain:** A set of programming tools used by Bazel to build software for different environments or platforms.
- **Workspace Rule:** A rule in the WORKSPACE file that fetches external dependencies needed for the build.
- **Workspace:** The top-level directory of a project where Bazel starts to evaluate builds. It contains a WORKSPACE file.

This quick reference guide is designed to be a starting point in your journey with Bazel. For more detailed explanations and advanced topics, you can delve deeper into the official Bazel documentation and community resources.

## [Quick Reference Guide to Bazel Commands](#)

This section provides a concise yet comprehensive guide to essential Bazel commands. It serves as a quick reference, designed to enhance your workflow.

## Bazel Command Structure

Bazel commands generally follow this structure:

```
bazel <command> <options> <targets>
```

- **<command>**: The action you want Bazel to perform (for example, build, test).
- **<options>**: Additional flags to customize the command behavior.
- **<targets>**: The specific code components (for example, modules, packages) to which the command applies.

## Core Commands

- ``bazel build``
  - Purpose: Compiles and assembles the specified targets.
  - Usage: ``bazel build [options] <targets>``
  - Example: ``bazel build //my/app:app_target``
- ``bazel test``
  - Purpose: Runs tests for the specified targets.
  - Usage: ``bazel test [options] <test_targets>``
  - Example: ``bazel test //my/app:test_all``
- ``bazel run``
  - Purpose: Builds and runs the specified target.
  - Usage: ``bazel run [options] <target>``
  - Example: ``bazel run //my/app:app_binary``

- `bazel clean`
  - Purpose: Removes artifacts and data from previous builds.
  - Usage: `bazel clean [options]`
  - Options:
    - `--expunge`: Also removes the entire working tree.
- **Useful Options**
  - `--config`: Specifies a configuration to use.
  - `-c` or `--compilation_mode`: Sets the compilation mode (for example, debug, optimized).
  - `--define`: Overrides a build variable.
  - `--test_filter`: Runs only the tests that match a specific filter pattern.

## Advanced Commands

- `bazel query`
  - Purpose: Queries the dependency graph of your build.
  - Usage: `bazel query 'query_expression'`
- `bazel coverage`
  - Purpose: Generates test coverage reports.
  - Usage: `bazel coverage [options] <test_targets>`

# APPENDIX B

## Additional Resources

### Recommended Books

Here are notable books that would be valuable for a software developer interested in learning about Monorepo and Bazel:

- ***Effective React Development with Nx***: This book, authored by Jack Hsu from Narwhal Technologies Inc., covers Nx and Monorepo-style development for React applications. It offers insights on how large companies like Google, Facebook, and Microsoft use the Monorepo approach. The book dives into creating React workspaces, running various tests using Nx, understanding and enforcing workspace structures, and scaling Monorepo and CI/CD pipelines. It is an excellent resource for understanding how to build React applications more effectively within a Monorepo setup.
- ***Enterprise Monorepo Angular Patterns***: Written by Victor Savkin and Nitin Vericherla, also from Narwhal Technologies Inc., with contributions from Thomas Burleson, this book focuses on angular development within a Monorepo context. It includes strategies for organizing code into single-purpose libraries for composing large applications, enforcing codebase consistency, and using Nx tools for workspace analysis. This book is particularly beneficial for those working in large organizations with complex angular projects.
- ***Getting Started with Bazel* by Benjamin Muschko**: Published in February 2020 by O'Reilly Media, Inc., this



book is an excellent resource for understanding Bazel's basics and its application in Java projects. It covers topics like setting up a Java-based project, defining dependencies, performing automated tests, and extending Bazel's capabilities. It also provides insights into improving build performance and executing Bazel projects on continuous integration servers.

- ***Beginning Bazel: Building and Testing for Java, Go, and More*** by P.J. McNerney: This book, released in December 2019 by Apress, guides readers through using Bazel in various programming languages including Java, C++, Android, iOS, and Go. It addresses how to speed up builds and tests, and run Bazel on different operating systems like Linux, macOS, and Windows. This book is particularly beneficial for experienced programmers looking for alternative build/test tools.
- ***Agile Testing: A Practical Guide for Testers and Agile Teams*** by Lisa Crispin and Janet Gregory: This book offers insights into integrating testing into the agile process, emphasizing a collaborative approach and the importance of iterative quality assurance.
- ***Software Testing: A Craftsman's Approach*** by Paul C. Jorgensen: This comprehensive guide covers the technicalities and nuances of software testing, with a focus on both traditional and object-oriented software.
- ***Software Test Automation: Effective Use of Test Execution Tools*** by Mark Fewster and Dorothy Graham: This book dives into automated software testing, offering practical advice and real-world examples.
- ***Foundations of Software Testing ISTQB Certification*** by Rex Black, Erik van Veenendaal, and Dorothy Graham: A crucial resource for those preparing

for the ISTQB certification, this book covers software testing principles, practices, and terminologies.

- ***Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*** by James Whittaker: Focused on exploratory testing, this book emphasizes adaptability and continuous discovery in the testing process.
- ***Software Testing*** (2nd Edition) by Ron Patton: An excellent introduction to software testing, this book covers both manual and automated testing techniques.
- ***Buddha in Testing: Finding Peace in Chaos*** by Pradeep Soundararajan: This book is particularly helpful for testers facing the challenge of balancing mental chaos and high-quality work.
- ***Penetration Testing - A Hands-On Introduction to Hacking*** by Georgia Weidman: Ideal for those interested in ethical hacking and penetration testing, this book offers informative content and techniques for evaluating enterprise defenses.
- ***Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*** by Elisabeth Hendrickson: This book illuminates the dynamic approach of exploratory testing, emphasizing adaptability and risk management.
- ***Leading Quality*** by Ronald Cummings-John and Owais Peer: Focuses on the essence of quality leadership in software development, advocating for a holistic approach that transcends traditional testing.
- ***Software Engineering at Google*** by Titus Winters, Tom Manshreck, Hyrum Wright: This book includes a chapter on Dependency Management, offering insights into the challenges and best practices of managing networks of libraries, packages, and dependencies in software engineering.

- ***Software Architect's Handbook*** by Joseph Ingeno: It provides a comprehensive look at software versioning, a crucial aspect of software development. The book discusses the importance of using a formal convention for software versioning and its role in effective dependency management.
- ***Implementing Azure DevOps Solutions*** by Henry Been, Maik van der Gaag: This book covers Dependency Management in the context of Azure DevOps. It offers practical guidance on how to identify shared components and use package management effectively.
- ***KANBAN: Streamlining Workflow for Effortless Efficiency (2023 Guide for Beginners)*** by Margot Jackson: This book is a comprehensive guide to understanding and implementing the Kanban method, a tool for optimizing processes and enhancing productivity. It covers various aspects such as Kanban principles, the design and implementation of Kanban boards, setting Work in Progress (WIP) limits, and the use of visual metrics for performance tracking. The book also includes real-world case studies and discusses how to scale Kanban for different levels of an organization.
- ***Streamlining Workflows: A Beginner's Guide to Project Management Tools***: This guide explores the value of project management tools in streamlining workflows, particularly in the field service industry. It discusses the challenges faced by this industry, such as coordinating multiple teams and managing complex schedules, and how project management tools can help overcome these by automating and organizing tasks, reducing human errors, and improving work quality. The book emphasizes the importance of choosing the right tool for your specific needs and the transformative

potential these tools have in optimizing workflow and enhancing customer satisfaction.

- ***Building Software: A Practitioner's Guide*** by John Minnihan: This book analyzes the practical aspects of software construction, focusing on modern tools and techniques. While it covers a range of topics, its sections on build systems and deployment are particularly relevant for those interested in mastering Bazel.
- ***Continuous Integration: Improving Software Quality and Reducing Risk*** by Paul M. Duvall, Steve Matyas, and Andrew Glover: This book explores the concept of Continuous Integration (CI), a practice integral to modern software development and deployment. It provides insights into how tools like Bazel can be integrated into CI pipelines to streamline and improve the build and deployment processes.
- ***Mastering DevOps: A Practical Guide to Building and Maintaining Large Scale, High Performance DevOps Pipelines*** by Jonathan McAllister: This comprehensive guide covers the full spectrum of DevOps practices, with a focus on automation and scalability. It offers insights into how Bazel can be leveraged within a DevOps context to enhance efficiency in building and deploying large-scale services.

All of these books provide comprehensive insights into Bazel and are tailored for developers who wish to deepen their understanding and skills in this powerful build and test tool.

## **Online Communities**

As a software developer diving into Bazel and Monorepos, engaging with their online communities can be highly beneficial. This section guides you through participating in

these communities, enhancing your understanding and contribution.

## **Understanding Bazel and Monorepo Communities**

- Forums and Discussion Boards:
  - Stack Overflow at <https://stackoverflow.com/questions/tagged/bazel>,
  - Reddit at <https://www.reddit.com/r/bazel/>,
  - Specialized forums are great for discussions like Bazel User Group at <https://groups.google.com/g/bazel-discuss>
  - Bazel GitHub Discussions at [github.com/bazelbuild/bazel/discussions](https://github.com/bazelbuild/bazel/discussions) (<https://github.com/bazelbuild/bazel/discussions>).
- Social Media Platforms: Follow discussions on Twitter, LinkedIn, and Facebook.
- GitHub Repositories: Explore Bazel on GitHub at <https://github.com/bazelbuild/bazel> for code, issues, and discussions.

## **Participating in the Community**

- Asking Questions: Be inquisitive; the community thrives on curiosity. Bazel Slack Channel: <https://slack.bazel.build>.
- Answering Questions: Share your knowledge to help others. Bazel GitHub Issue Tracker at <https://github.com/bazelbuild/bazel/issues>.
- Respectful Interactions: Maintain respect and constructiveness.

## Contributing to the Community

- Contributing Code: Contribute to open-source projects on GitHub at <https://github.com/bazelbuild>.
- Writing Blogs and Articles: Share experiences through blogs and articles.
- Presenting at Meetups and Conferences: Engage in local events and conferences.
- Creating Tutorials and Guides: Help beginners with educational content.
- Leveraging Community Resources for Learning and Growth
- Awesome Bazel Resources at <https://github.com/jin/awesome-bazel>
- Following Roadmaps and Changelogs: Stay updated with Bazel's roadmaps at <https://bazel.build/roadmaps> and changelogs.
- BazelCon Videos as BazelCon on YouTube at [https://www.youtube.com/results?search\\_query=BazelCon](https://www.youtube.com/results?search_query=BazelCon)
- Webinars and Workshops: Participate in educational events.
- Case Studies and Reports: Explore real-world applications and insights.
- Actively participating in Bazel and Monorepo communities enhances your skills and contributes to the technology's evolution. Every interaction enriches the entire community.

# Index

## Symbols

.bazelrc file  
  about [62](#)  
  best practices [63](#)  
  command, utilizing [64](#)  
  directory, utilizing [62](#)  
  syntax [62](#), [63](#)

## A

Android/Kotlin  
  about [113](#)  
  application, enhancing [114-120](#)  
  best practices [124](#), [125](#)  
  key, building [121-124](#)  
  key, directories [115](#)  
  setting up [113](#), [114](#)

## B

Bazel  
  about [13](#), [249](#)  
  Build Event Protocol (BEP) [253](#), [254](#)  
  CI, enabling [179](#), [180](#)  
  Code coverage [80-82](#)  
  command line tools, utilizing [250](#), [252](#)  
  concepts [176-179](#)  
  data, interpreting [249](#)  
  data, profiling [252](#), [253](#)  
  features [13](#), [15](#)  
  install, setting up [20](#)  
  Monorepo, preventing [16](#)  
  principles [15](#), [16](#)  
  project, building [22-24](#)  
  responsibilities [175](#), [176](#)  
  scenarios [16](#), [17](#)  
  scripts, utilizing [249](#), [250](#)  
  setting up [174](#), [175](#)  
  TestSuite, managing [73-79](#)  
  unit tests, running [68-73](#)  
  uses [17](#), [18](#)

- Bazel 7, challenges
  - Bzlmod, migrating [315](#)
  - mechanisms, enhancing [316](#)
  - platform support, expanding [316](#)
  - projects, existing [317](#), [318](#)
  - robust, extension [316](#)
  - updates, adapting [317](#)
- Bazel, best practices
  - features, planning [311](#), [312](#)
  - future development, considering [314](#), [315](#)
  - Future Path, utilizing [313](#), [314](#)
  - modular dependency, managing [311](#)
  - Monorepo Tooling, implecating [314](#)
- Bazel, case studies
  - challenges, utilizing [303](#)
  - future plan, considering [305](#)
  - impact, results [304](#)
  - infrastructure, managing [306](#)
  - key factors, adopting [300](#)
  - responsibilities, learning [305](#)
  - strategy, implementing [301](#), [303](#)
- Bazel, concepts
  - .bazelrc file [62](#)
  - BUILD Files [27](#)
  - workspace file, analyzing [24-26](#)
- Bazelisk, installing [21](#), [22](#)
- Bazel Modules
  - about [99](#)
  - advantages [99](#)
  - Air-Gapped, configuring [102-104](#)
  - dependency, declaring [99-102](#)
  - graphs, gettings [105-109](#)
  - IDE, integrating [109](#), [110](#)
- Bazel Sandboxing
  - about [38](#)
  - concepts [38](#)
  - task, lifecycle [39](#)
  - uses [39](#)
- Bazel's Build Graph
  - about [265](#), [266](#)
  - aspects, analyzing [278](#), [279](#)
  - concepts, preventing [279](#), [280](#)
  - features, exploring [280](#), [281](#)
  - Hermeticity [273](#), [274](#)
  - Maven, migrating [271-273](#)
  - project, utilizing [275](#)
  - repository, managing [266-270](#)
  - toolchains, building [275-277](#)



## Bazel's rules

- about [43](#)
- core, dissecting [45-49](#)
- directory, initializing [44](#), [45](#)
- genrule, executing [55-58](#)
- key, advantages [43](#)
- key, components [52](#), [53](#)
- key concepts [50](#), [51](#)
- reasons, customizing [51](#), [52](#)

## Bazel, techniques

- rules, debugging [258](#)
- Starlark, capabilities [258](#)

## Bazel, testing strategies

- efficient, testing [66](#), [67](#)
- implementation, testing [67](#)
- reliable, testing [67](#)
- scalable, testing [67](#)

## Bazel Watcher

- about [179](#)
- installing [180](#)
- scenarios [180](#)
- using [180](#)

## Build Event Protocol (BEP)

- about [253](#), [254](#)
- baselines, implementing [255](#)
- continuous monitoring [256](#)
- format, converting [254](#), [255](#)
- performance, developing [255](#)

## BUILD Files

- about [27](#)
- architecture, analyzing [27](#), [28](#)
- attributes, preventing [30](#)
- dependencies [37](#), [38](#)
- key aspects [27](#)
- label, preceding [34](#), [35](#)
- packages, including [35](#), [36](#)
- queries [36](#), [37](#)
- rules, importing [29-31](#)
- targets, optimizing [32](#), [33](#)

## C

### Centralized Configuration Management [204](#)

#### CI

- about [179](#)
- concepts [180](#), [181](#)
- key aspects, analyzing [189](#), [190](#)

- remote execution, building [188](#), [189](#)
- CI, models
  - ephemeral [181-183](#)
  - hot-pool [186](#), [187](#)
  - multiple stateful [184-186](#)
  - sharded [187](#), [188](#)
  - single stateful [183](#), [184](#)
- CI, roles
  - formatting [190](#), [191](#)
  - static code, analyzing [192-194](#)
- Code Contribution
  - about [168-170](#)
  - CI/CD Pipeline, analyzing [170](#)
  - feature, branching [170](#), [171](#)
  - GitFlow [171](#), [172](#)
  - Trunk-based Development [172](#)
- Container Images Builds
  - about [233](#)
  - Bazel, running [236-238](#)
  - concepts [233-235](#)
  - fine-grained, targets [235](#)
  - image, exporting [236](#)
  - layered, approach [236](#)
  - OCI Image, using [236](#)
  - parallelization, caching [236](#)

## D

- dependency management
  - about [93](#)
  - conflicts, utilizing [97](#), [98](#)
  - external, dependencies [95-97](#)
  - internal, dependencies [94](#), [95](#)
- Diagnosing Cache Misses [257](#)
- Digital Service, case studies
  - Bazel, adopting [287](#)
  - challenges, utilizing [290](#), [291](#)
  - future plans, considering [292](#)
  - impact, results [291](#)
  - insights, learning [291](#)
  - stack, comprehensive [292](#), [293](#)
  - strategy, implementing [288-290](#)

## G

- genrule [55-58](#)
- GitFlow [171](#), [172](#)

## Golang

- about [149](#)
- app, building [153](#), [154](#)
- best practices [155](#), [156](#)
- environment, setting up [149](#)
- project, organizing [149-153](#)

## I

### Integration Testing [270](#)

## iOS

- about [156](#)
- app, running [160-162](#)
- best practices [163](#), [164](#)
- environment, setting up [157](#)
- external, dependency [163](#)
- project, organizing [157-160](#)

## J

java\_plugin [45](#), [46](#), [51](#)

## L

### Large Monorepo Codebases

- about [216](#)
- Bazel, configuring [220](#)
- best practices [221](#), [222](#)
- branch, merging [219](#)
- challenges, navigating [220](#)
- code, sharing [217](#)
- conflicts, minimizing [219](#)
- dependencies, managing [216](#)
- debugging, issues [220](#), [221](#)
- environment strategies, managing [218](#)
- inter-module dependencies, managing [218](#)
- repositories, integrating [216](#)
- strategies, modulazing [217](#)
- third party, handling [217](#)

## M

### Microservices API, dependency

- API, versioning [239](#), [240](#)
- automate dependency, updating [240](#)
- software configure, managing [241](#), [242](#)
- transitive, managing [238](#), [239](#)

Monitoring Remote Cache Hit [256](#)

Monorepo

about [1-3](#)

benefits [4, 5](#)

challenges, implementing [8, 9](#)

continuous, integrating [3](#)

history, optimizing [5, 6](#)

Polyrepo Approach [4](#)

typical fears, utilizing [6-8](#)

Monorepo Builds

about [223](#)

bottlenecks, utilizing [224](#)

cache strategies, managing [223](#)

caching, techniques [225](#)

data, configuring [224](#)

dynamic build graph, optimizing [226](#)

Efficient Starlark, writing [228](#)

execution, optimizing [227](#)

performance, optimizing [228](#)

resource, managing [225](#)

scenarios, developing [227](#)

setting up [223](#)

source code, managing [230](#)

Starlark Scripts, profiling [229](#)

version control system, structuring [229](#)

Monorepo compatibility, enforcing [104](#)

Monorepo Development [12, 13](#)

Monorepo Development, best practices

AI/ML, integrating [309](#)

dependency, managing [310](#)

execution strategies, enhancing [309](#)

foster, collaborating [310](#)

security, enhancing [310](#)

sustainable, developing [310](#)

Monorepo Layout

about [199](#)

Centralized Configuration Management [204, 205](#)

code, maintaining [205, 206](#)

code reusability, sharing [201, 202](#)

package modules, organizing [210, 211](#)

packages, adopting [212, 213](#)

Pitfalls, ways [208-210](#)

security, considering [206-208](#)

structure, directory [199, 200](#)

testing, strategies [202-204](#)

Monorepos, dependencies

case studies [260](#)

divide, conquer [259](#)

scalability, improving [260](#)

## N

NodeJS/Typescript

about [133](#)

app, building [147](#)

aspect, building [133](#), [134](#)

best practices [147](#), [148](#)

environment, setting up [134](#)

project, organizing [134-139](#)

## O

Optimizing Remote Execution [257](#)

Orchestrating Microservices

about [243](#)

dependencies, techniques [243](#)

features, toggling [244](#)

Microservices, scaling [244](#), [245](#)

## P

Performance Testing [82-85](#)

Polyrepo Approach [4](#)

Python

about [125](#)

application, building [131](#)

best practices [131](#), [132](#)

environment, setting up [126](#)

project, organizing [126-130](#)

## S

Serverless Service Platform, case studies

ability, integrating [299](#)

Bazel, adopting [294](#)

challenges, analyzing [296](#), [297](#)

future plans, considering [298](#)

impact, results [297](#)

infrastructure, learning [298](#)

strategy, implementing [294](#), [296](#)

SOPS [242](#)

## T

Test Isolation [88](#)

Test Parallelism [88](#), [89](#)  
Test Parallelism, best practices [89](#)  
Trunk-based development  
  about [172](#)  
  benefits [173](#), [174](#)  
  CI/CD Pipelines, configuring [11](#)  
  concepts, utilizing [10](#)  
  features [173](#)  
  reasons [10](#)

## U

User Acceptance Testing (UAT)  
  about [85](#)  
  challenges [86](#)  
  consistency [85](#)  
  Cypress, using [86-88](#)  
  dependency, managing [85](#), [86](#)  
  interoperability [85](#)

## W

WORKSPACE [22](#), [42](#)