



O'REILLY®

What Is Portainer?

An Introduction to Container Management for Developers

Fabian Peter

REPORT

What Is Portainer?

An Introduction to Container Management for Developers

Fabian Peter



Beijing • Boston • Farnham • Sebastopol • Tokyo

What Is Portainer?

by Fabian Peter

Copyright © 2023 O'Reilly Media Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: John Devins
- Development Editor: Jill Leonard
- Production Editor: Katherine Tozer
- Copyeditor: nSight, Inc.
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- February 2023: First Edition

Revision History for the First Edition

- 2023-02-16: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *What Is Portainer?* the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14209-4

[LSI]

Introduction

My first encounter with Portainer was back in 2018, when I managed a fleet of Docker Swarm clusters for a team of developers that was running an application composed of containerized microservices on top of it. Naturally, the team regularly needed to interface with its containers, and doing it through Docker's CLI meant it was necessary to give the team access to the underlying Swarm nodes. It also required a lot of understanding of how Docker works to get access to logs or debug network issues between containers. But I thought this level of understanding shouldn't be expected of application developers. So I started researching alternative ways to manage such an environment and make it easy to give people access to resources in a human-friendly but controllable way.

When my research ended, it led to two web-based solutions that seemed a good fit: Portainer and SwarmProm. I decided to give Portainer a try because of its simple installation procedure and the vibrant community it already had, which gave me a good feeling about its stability and feature completeness. So I set up a central Portainer instance running in Docker, connected the Docker Swarm clusters through Portainer's Edge Agent, and configured appropriate access levels for the development team. In less than an hour, I had a web-based management UI that gave the developers access to their respective deployments, making their microservice architecture visible and digestible even for nonexperts. With a few clicks, the team was able to read logs, access

container shells, and restart failed containers. Portainer not only enabled them to better understand and manage their own services—it also allowed them to make meaningful adjustments to the system without requiring my intervention or deep knowledge about the inner workings of Docker.

Portainer's UI connects all the dots for my team's developers, giving them access to information about networks, volumes and their contents, and even the status of the underlying nodes. The developers were now in the position to independently own their deployments and interface with them whenever they needed.

I have used Portainer in every container environment since then. It's a great tool to give developers and operators access to containerized architectures regardless of their personal expertise. Portainer allows administrators to define different access levels for different people and thus enables them to keep their container infrastructure secure and stable while still allowing everyone to do their jobs independently. It speeds up their learning process, eases management, and gives people confidence to own their application deployments.

Portainer comes in two editions: the Community Edition, which is open source and freely available on GitHub; and the Business Edition, which bundles additional, enterprise-ready features on top of the Community Edition.

Modern software development teams are often cross-functional and operate

with a “you build it, you run it” mentality. This report will outline how Portainer can help developers, operators, and service managers alike to deploy and operate containerized applications with that mindset, providing self-service and meaningful insights to all stakeholders.

Since Docker Swarm has mostly been superseded by Kubernetes in recent years, you will learn about the role Kubernetes plays in modern application development, the challenges developers face with cloud native software architecture, and how Portainer enables development teams to manage their application stack by incorporating industry best practices like GitOps, role-based access control, and hybrid-cloud environments.

It’s hard to understand containerized systems from log messages or YAML manifests alone, and it’s even harder if you’re not familiar with the inner workings of these systems. Portainer makes this visible and establishes connections between the components of your application that you can follow by simply clicking on them in a modern UI. It can help you to get a deep understanding of what you’ve built and how it works together by providing human-friendly access to these resources.

This report is for you if you know the pain of managing containers with CLI tools or pure GitOps mechanics and want to learn more about different approaches to it. In this report, we cover what Portainer is and how it fits into today’s technology landscape. If you want to jump right in and put it to use, you should head over to its [official documentation](#) since it covers the ins and

outs of the tool itself, describes how to set it up correctly, and gives great instructions on how to get started managing containers with Portainer.

Chapter 1. Kubernetes Is the Operating System of the Cloud

Before diving deeper into the specifics of Portainer, I want to establish a bit of context in terms of the ecosystem Portainer comes from and its broader reasons for existence. As mentioned, I first used Portainer to manage a few Docker Swarm clusters. Over the years, the industry has backed away from Docker Swarm and toward Kubernetes as its container orchestrator of choice. My clients have followed this trend. But my initial need to provide secure access to containerized applications for teams of developers persisted—and it still does today. Luckily, Portainer quickly added support for Kubernetes as a container orchestrator.

While the underlying mechanics are fairly different between Docker Swarm and Kubernetes, Portainer managed to provide the same user experience for both systems. That made it easy for the teams I already worked with to make a smooth transition from one platform to the other. To understand why one would go with one orchestrator over another and why Portainer started to focus mainly on Kubernetes, we need to know a few things about Kubernetes itself and its own *raison d'être*.

In 2014, Kubernetes was developed and released by Google as an improved variant of container orchestration, building upon the knowledge Google acquired over years of running distributed systems at scale.

NOTE

Kubernetes was donated to the [Cloud Native Computing Foundation \(CNCF\)](#) in 2015 with its first stable release and has been developed as a community project under the umbrella of the CNCF since then. Its underlying architecture and design principles stemmed from [Borg](#) and [Omega](#), Google's internal cluster management system. Many of the developers that worked on Borg and Omega at Google work on Kubernetes now.

Given its lineage and being open source, Kubernetes (or K8s for short) soon became the dominating container orchestration engine in the cloud native ecosystem. It was built to handle huge distributed workloads, and the community efforts evolving around it made sure the software was keeping up with the requirements of the big cloud providers and individual enterprises looking to run containers in production. One of the first companies to incorporate Kubernetes into their commercial tech stack was Red Hat, which used it as the foundation for its OpenShift Container Platform.

In order to understand what makes Kubernetes the operating system of the cloud, we need to have a better understanding of the history of containers and how they changed the way we build and deploy software today.

A Few Words on Containers

Simply put, containers are packages of software that contain all of the necessary elements to run in any environment.

With the advent of Docker in 2013, the way we perceive and use containers has changed. Before it, software containers used to work a lot like virtual machines (VMs)—carrying a whole operating system, running multiple long-running applications (i.e., services), and giving you access through Secure Shell (SSH). The main difference between containers and VMs was that VMs emulated the full hardware stack, isolating the system fully from the underlying operating system, while containers worked in a chroot environment, being dependent on the kernel and resources of the surrounding operating system.

Docker established a new paradigm by stripping away pretty much everything but the actual application you want to run and its most important dependencies. While before you had to run a full web application stack, including the web server, database, and anything else you needed in a single container, with Docker you could split each of those services into their own containers and have them interface with each other over the network.

If done correctly, a Docker container only runs a single process, and if that process is being terminated, the container is too. Unlike virtual machines or “legacy” containers, this way of doing things comes closer to how process management in an operating system works. It requires a different kind of planning and resource management when you need multiple processes to work together, but it also allows you to properly isolate workloads and reduce the risk surface of applications to a minimum.

With containers only running a single process and its most important dependencies, Docker also enables you to make those applications portable between different operating systems since a container is no longer dependent on its underlying or surrounding operating system. Everything is perfectly isolated and only interfaces with the outside world over the network or by specifically mounting directories from the host machine into the container. As long as the server you want to run the container on supports the Docker Engine, you can rest assured that your container will run perfectly fine on it.

So, What Is Kubernetes?

That level of portability created new possibilities in terms of scalability. It's very common to run an application on multiple servers and place a load balancer in front of it to achieve higher stability or better performance. We call this *horizontal scaling*. Before, the main mechanic to scale an application was to *vertically scale* the server it was running on—i.e., giving more resources to a single machine. That way of scaling has its limitations in terms of hardware and availability since upgrading machines typically results in downtime.

With Docker, it became very easy to run the same container on different machines and achieve horizontal scaling. [Docker Swarm](#) was the first approach to incorporate fast up- and downscaling over multiple clustered machines and an integrated load-balancing mechanism into the Docker

Engine itself. You could easily upgrade a single container to a [service](#) and increase its replicas, having Docker Swarm manage the placement of the replicas on the nodes in the cluster and handle the routing of traffic through its integrated overlay network.

Kubernetes took the concepts of containerized software and workload orchestration to a whole new level, providing well-designed abstractions for handling compute, network, and storage over distributed servers. While Docker Swarm exposes a human-friendly interface to orchestrate containers, it still lacks many advanced features. There's no production-grade support for distributed storage mechanisms apart from NFS (Network File System) or SMB (Server Message Block), and it has no concept of network policies or support for custom ingress implementations for advanced handling of incoming traffic. Kubernetes—stemming from Google's experience in running distributed systems at scale—delivers all those things and more by providing generalized building blocks instead of specific implementations.

In Kubernetes, everything is handled by APIs whose implementations are decoupled from the core logic of the API server. As a result, the developer community was able to build implementations for specific components like network, storage, or ingress independently from the core logic that keeps everything running. This works a lot like modern operating systems (OS) in that most of the integral parts of the system have been compartmentalized to specify clear interfaces for their domains without providing the actual logic to do anything, giving end users the opportunity to use their own tooling to

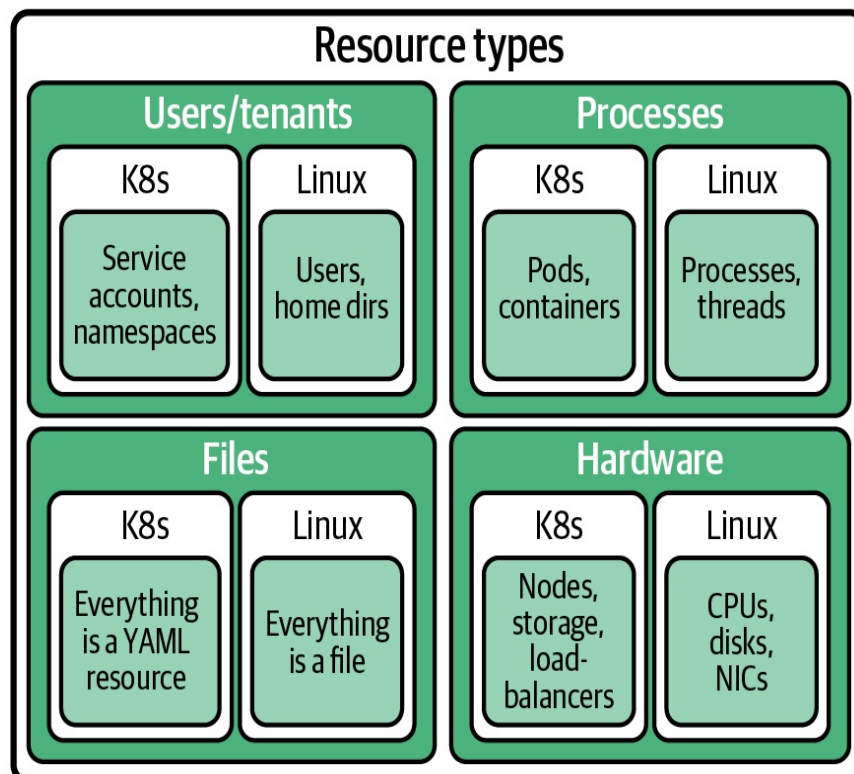
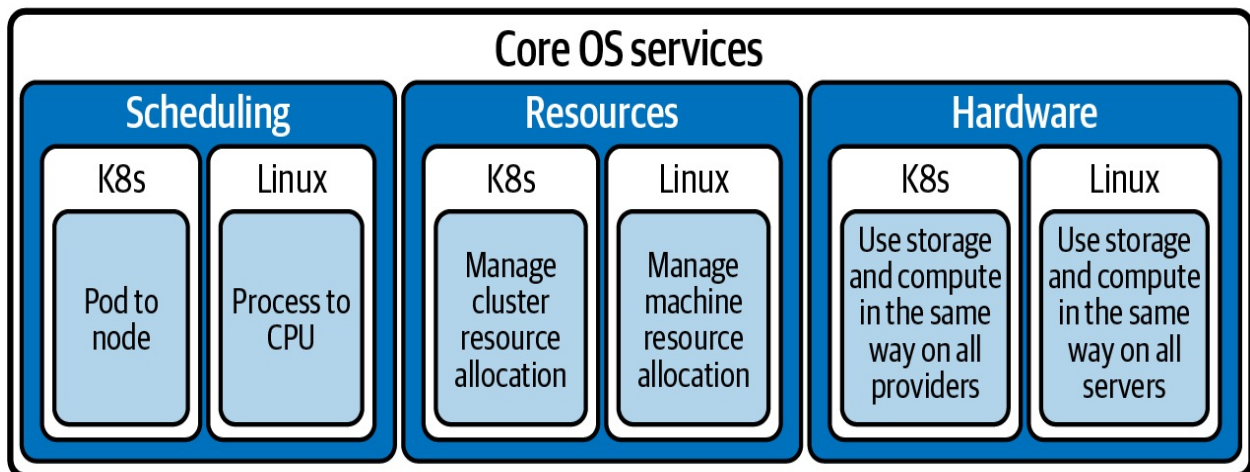
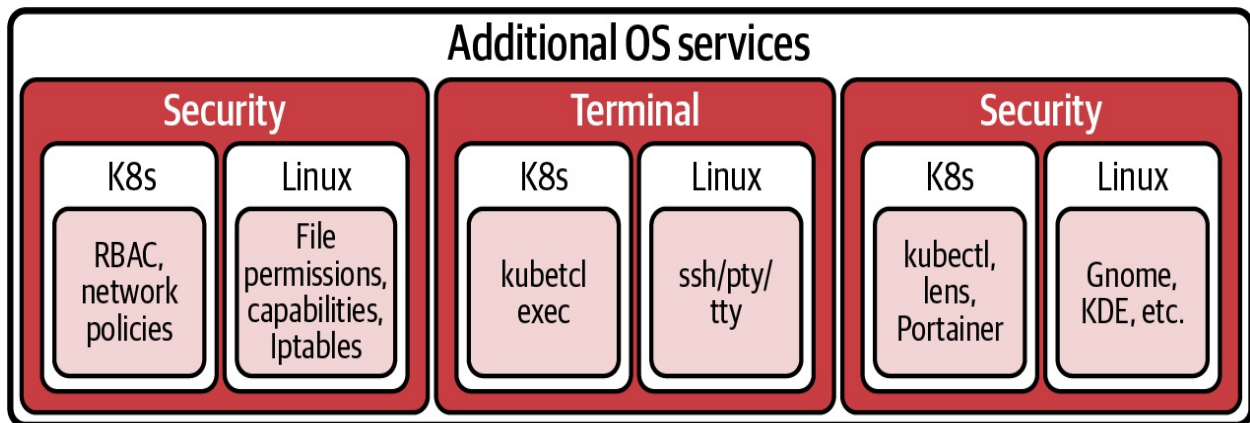
handle the domain-specific necessities.

A good example for this is the networking stack in Linux. While most distributions use Linux's own implementation of the TCP stack, it's absolutely fine and sometimes desirable to replace it with something homegrown.

NOTE

When most people think of an OS, they think of Microsoft Windows or Linux, which are specific implementations of what an OS actually is: low-level system software that manages computer hardware and software resources and provides common services for computer programs.

Looking at Kubernetes from a high level (see [Figure 1-1](#)), it does exactly this: handling memory allocation for processes, providing ways for processes to communicate, and providing interfaces to the underlying hardware. The difference between a classical OS and Kubernetes is that Kubernetes does this for a cluster of machines instead of a single one. With that said, a container becomes a single process, and Kubernetes works like the [kernel](#) of an OS, making sure that the process gets what it needs to work properly.



The Road to Microservices

Modern operating systems profit a lot from being able to share resources efficiently between processes. Pretty much every OS comes with some kind of package manager or service registry, allowing software developers to reuse components and access a central index of “what’s there.” One good example of this is how [apt \(advanced packaging tool\)](#) works in debian-based operating systems. Instead of including every necessary library or tool in their own code, developers can reference shared libraries and programs that already exist in the OS, making it very easy to run software without causing bloat. Much of how this works today stems from the [UNIX philosophy](#): *do one thing and do it well*.

NOTE

The Unix philosophy is about building simple and modular code that is highly extensible and can easily be repurposed or composed with other components to build bigger solutions with a clear separation of concerns. In other words: keep it simple; do one thing and do it well.

But how does this translate to Kubernetes? Well, instead of processes, software developers often refer to the software running in containers as

microservices. Before the rise of containers, software development had a very monolithic character—there was a single, often huge code base everybody worked on, and software was mostly running on a single machine, making it necessary to handle interprocess communication within the application itself or rely on the underlying operating system to provide the necessary resources.

Today, most of the software that is powering the web as we know it is designed to have a clear API and do one thing—and do it well. Instead of writing huge and complex software monoliths, developers started to further compartmentalize their programs into single, independent microservices that talk to each other through their respective APIs over the network. These services are often stateless, meaning they can easily be scaled to multiple replicas for increased throughput or availability.

This new way of building software made it necessary to have a higher-level system that was able to orchestrate these microservices, schedule them on different servers, allocate resources to them, and provide ways for them to communicate with each other. That's what Kubernetes is doing for them: like an OS, it provides access to shared resources, enables them to know about each other, and makes sure to keep everything running smoothly.

If you've ever worked with containers, this is nothing new for you. We commonly refer to this way of building software as *cloud native architecture*. The cloud was one of the biggest driving factors of technological innovation

in the past decade and enabled many new patterns of building and running software, but it also introduced new challenges for software developers. In the following chapter, I will highlight some of these challenges and how Kubernetes and Portainer evolved as solutions to common problems in cloud native architecture.

Chapter 2. Challenges of Cloud Native Architecture

Cloud native architecture is a design methodology that utilizes cloud or cloud-like services to allow dynamic application development techniques that take a modular approach to building, running, and updating software through a suite of microservices rather than a monolithic application infrastructure. In the beginning, developers were limited to services provided by actual cloud providers. Meanwhile, cloud native technologies have evolved and are available for on-premises systems and even developer workstations to make them more accessible and reduce friction in the development lifecycle.

Containers have become the biggest enabler of this trend, allowing developers to build and package software on their workstations while being assured that their containers will run in Docker or Kubernetes on any cloud native infrastructure. Gone are the times of “it works on my machine.”

My personal cloud native journey already began when I discovered Docker but really accelerated once I started working with Kubernetes. As outlined in the previous chapter, operating Kubernetes and successfully running applications on top of it requires a different understanding of how things work together. The new level of abstraction developers and operators have to wrap their heads around is also important for the understanding of Portainer’s features and their reasoning behind specific implementations. Since

everything is far more “distributed” and “decoupled” in cloud native architecture, tools like Portainer evolved to provide simpler interfaces for end users to be able to get their work done and not drown in the complexity of this new world. I had a personal “aha” moment when I discovered that Portainer can act as a proxy for the Kubernetes API server, allowing me to define fine-grained access controls within Portainer to limit the access my developer teams had on their respective clusters. This is so much easier than setting up RBAC in Kubernetes itself, which is something I’ve never wasted a thought on when using Docker before since Docker just didn’t have any meaningful access control. While it’s great to be able to properly authorize every single action in a system, it also poses many new challenges when working in bigger teams with complex technologies: there’s always something that doesn’t work as expected. Having a tool to provide meaningful insight and workflows for setting things up not only saves me a lot of time: I can rely on best practices and the smarts of others who’ve used it before me and implemented things “as they should be,” as opposed to me fiddling around with settings I don’t fully grasp. This chapter is dedicated to explaining the cloud native ecosystem, how it changed the way we build software today, and the challenges we face on a daily basis that tools like Portainer try to make easier for us.

The available toolchain for developers evolved with the cloud native ecosystem and so did the learning curve a developer had to master to be able to make proper use of cloud native architecture. This led to new specializations like DevOps or platform engineers, whose task it is to provide

fine-tuned workflows and toolkits to keep the development and operations of software in sync. It also required rethinking a lot of the development lifecycle, introducing more automation to the process, and integrating microservices written in different languages by different developers before they could be used effectively to work together as an application.

One of the biggest challenges developers face today is the complexity of that ecosystem with all its new technologies and paradigms. It's not enough to "just know Python." You need to know Docker, continuous integration (CI), testing frameworks, and deployment techniques too. To properly integrate independently developed software, a huge range of tests and checks are required. Previously these tests could be done locally as developers had the full monolith right at their fingertips. But today they make use of dedicated CI platforms like Jenkins or GitLab to asynchronously build, test, and integrate their microservices in an automated way.

Continuous delivery (CD) involves the same workflow established over time for the deployment of these microservices into environments like Kubernetes. CD is a methodology that ensures the delivery of the output of CI pipelines to special test or even production environments whenever they successfully produce usable artifacts. Oftentimes, developers push their code to a branch in a Git repository and wait for a range of automations to succeed before they know if the newly written code works as expected and can be merged into the stable branch, which then automatically gets deployed to production systems. Companies like Meta or Uber do this several thousand times a day and thus

are able to ship features and fixes to their customers almost in real time, and this practice has become a standard in modern software development.

This requires a lot of orchestration between all involved parties and technologies. Most of this will be automated over time, but it also raises the bar for what's deemed "state-of-the-art" software development. The amount of tooling you need to develop modern applications increased tenfold with the rise of cloud native architecture, and so did the amount of knowledge a development team needs to bring to the table to successfully do it.

In the rest of this chapter, you will learn about a few specific challenges that arose from the increased complexity of the software development ecosystem to lay ground for the next chapter in which you'll learn about the motivations to build and use Portainer.

Distributed Development

Distributed development means two things:

- Developers are working from remote locations, often in different time zones.
- Components of an application (the so-called microservices) are being developed by different developers or teams of developers.

These things often come together in modern organizations, making it

necessary to have systems in place to coordinate asynchronously.

On the one hand, the distributed nature of development these days offers a few perks when it comes to the velocity of development and stability of applications:

- Microservices can be developed and released to production 24 hours a day, decreasing the time to market for features or fixes.
- Applications can be supported 24 hours a day, reducing the risk surface of software-as-a-service (SaaS) business models.

On the other hand, the amount of coordination necessary to keep everybody in sync and applications healthy has increased a lot. Instead of a quick chat with a colleague over the boundaries of a change one wants to implement, developers have to follow predetermined processes and navigate their changes through automated control-systems (like CI/CD) before they can request qualified feedback from their peers or get a feeling for the quality and impact their code has.

This can add lots of friction to the development process until all the bits and pieces have been tuned to work together. Tools like [Portainer](#), [GitLab](#), and [Jenkins](#) have been built to give developers a common place to work through these processes and give them visual feedback and a good developer experience (DX) when dealing with changes to complex systems in a distributed environment.

Enterprise Requirements

In the world of software development, there's basically two gears companies can shift in:

Startup speed

Startups naturally need to be fast, adjust to change quickly, and often pivot in terms of business and technology. As a result, their main focus is to keep developer productivity and development velocity as high as possible to be able to scale on disrupting markets and industries.

Enterprise quality

Enterprises have a slower pace in everything. With established ways of doing business and usually enormous headcounts, the focus lies on keeping things in good shape and under control to serve the business's needs and keep it safe from disruptions.

A typical result of a startup building disruptive technologies are enterprises that want to use them. And with their interest come their requirements in terms of security, stability, and integrations for their existing tech stack.

This, again, puts a lot of pressure on developers as typically enterprises are not prepared or willing to adjust their infrastructure to the needs of modern software. Tools like Portainer, [Cilium](#), or [Keycloak](#) try to build bridges between these worlds, offering interfaces and solutions that serve startups and

enterprises alike. This helps to keep DX and velocity high while delivering increasingly complex solutions that often serve no direct purpose for the people developing them.

Hybrid Environments

We already learned that the cloud is the main driver of cloud native architecture. Amazon Web Services (AWS) was the first major public cloud provider, showing that a shared usage model for IT infrastructure is not only possible but beneficial for all involved parties. Many companies, including Google with Google Cloud Platform (GCP) and Microsoft with Azure Cloud, followed their lead and launched offerings that were competitive with AWS. Customers could now escape Amazon's [walled garden](#), shifting parts of their infrastructure to other providers, creating hybrid environments.

Along with generic services like virtual machines or managed Kubernetes clusters, these cloud providers also offer specialized services like managed databases, managed message queues, or big data tooling, usually wrapped inside their own business logic and APIs. This introduces additional complexity to the tech stack of software developers as they have to deal with each provider's specific implementation of the underlying technology (e.g., [PostgreSQL](#), [MySQL](#), or [Elasticsearch](#)). Moving between cloud providers, while generally possible, is a complex undertaking that requires thoughtful design of the software integrating with their services and usually a lot of

manual effort.

But being able to run software on any cloud provider is absolutely worth the effort as it enables companies to react to pricing changes or changing requirements. Technologies like Kubernetes try to account for the need to be flexible in that it serves as a generic abstraction layer on top of cloud services, so developers can focus more on their actual mission and less on wrangling infrastructure. Tools like Portainer, [Rancher](#), and [OpenShift](#) enable them to run their workloads in hybrid environments that even include private cloud infrastructure, which is very popular with enterprises.

Integrating Legacy Systems

As outlined in [“Enterprise Requirements”](#), sometimes modern technologies have to integrate with legacy systems. A very good example for this? Car manufacturers: they have a huge need for stable software and thus seek to implement technologies like Kubernetes to operate it. On the other hand, they have to deal with lots of industrial-grade systems like a programmable logic controller (PLC), which is a widely used standard for programming shop floor equipment. These legacy systems typically don’t provide modern APIs that developers can interface with, and they are always highly shielded from unwanted access.

It’s a very complex endeavor to bridge the gap between these target systems

and modern development workflows, which are heavily relying on public cloud providers, CI/CD systems, and a high degree of developer autonomy. Many approaches to modern software development simply do not work when you have to deal with a microcontroller sitting inside a robot that can only be controlled by means of physical access to the machine.

From personal experience, I can say that there are not many ways to provide a good DX in these environments. It's hard to implement cloud native architecture in places that have not seen a cloud in ages, but tools like [Node-RED](#), Portainer, and of course Kubernetes attempt to converge their modern approaches with these legacy systems to offer developers ways to overcome their challenges while not giving up too much of the goodness of cloud native architecture.

Security

Security is a basic human need and, as such, very important for all software because it is mainly used by humans. Working with digital technologies, security is really hard to get right, and over the past few years, there has been a push to [shift left](#) in terms of security in software development. If you think of the software development lifecycle as a process line running from left to right, testing and security implications are typically something that comes after design and implementation. After all, you mostly check for leaks in your architecture after the house has been built, don't you? Shifting left means to

incorporate security far earlier in the process, before even building anything.

Unfortunately, the complexity of cloud native architecture makes it difficult to catch every defect before it hits a production system. As such, control systems like CI/CD and automated indexing and vulnerability scanning of software running in Kubernetes clusters have become the norm. But the risk surface is increasing proportionally to the compartmentalization of software into smaller components. Mixed with the distributed nature of development and hybrid deployments, where not all parts of an infrastructure can be deemed equal, the learning curve for keeping software secure is constantly growing, putting individual developers under increasing pressure.

Software like Portainer, Cilium, and [KubeSanity](#) try to account for that by providing developers with tools to make defects and vulnerabilities visible early on in the development process and give them a means to deal with those quickly and reliably.

Of course, there are more challenges to cloud native architecture than the ones I mentioned in this chapter. My goal was to highlight some of the biggest ones and describe ways to deal with them. One of those ways is to add Portainer to your tech stack. In [Chapter 3](#), you will learn how exactly Portainer can help you to build software in a cloud native way, demonstrating how you can solve exactly the aforementioned challenges with it.

Chapter 3. How Portainer Works

In the previous chapters, I laid out the foundations of modern software development and the challenges developers face. In this chapter I will briefly outline the most important functionality Portainer offers with regard to that problem space, and you will learn how Portainer helps you to manage the aforementioned challenges. If you want to go deeper on specific topics, many advanced resources are available, including Portainer's [official documentation](#), YouTube, and GitHub.

Portainer's purpose is to enable you and your team to easily and securely manage containerized workloads in different environments. It abstracts away as much of the complexity as possible while enabling you to dive deeper and interface with the container orchestration engine. In this chapter you will learn the basics of how Portainer works, and how it enables you to manage hundreds of environments with hundreds of people and securely deploy your applications while keeping your sanity.

Managing Different Container Environments

In Portainer, a single container orchestration system is called an *environment*. Portainer allows you to manage many of these environments to which you

can deploy your application containers (see [Figure 3-1](#)).

It supports the following types of environments:

Docker

Standalone [Docker hosts](#) as well as [Docker Swarm clusters](#)

Kubernetes

Standard [Kubernetes clusters](#)

ACI

[Azure Container Instances](#)

Nomad

[HashiCorp Nomad](#) clusters

KaaS

Managed Kubernetes clusters offered by a cloud provider



Select your environment(s)

You can onboard different types of environments, select all that apply.



Docker

Connect to Docker
Standalone / Swarm via
URL/IP, API or Socket



Kubernetes

Connect to a
kubernetes
environment via URL/IP



ACI

Connect to ACI
environment via API



Nomad

Connect to HashiCorp
Nomad environment
via API



KaaS

Provision a Kubernetes
environment with a
cloud provider

Start Wizard

Figure 3-1. Portainer's Environment Wizard

For each environment type, Portainer offers different options to connect to the actual environment:

Via API

Docker, Kubernetes, ACI, and Nomad environments can be accessed directly through their API and the respective credentials such as a [Kubeconfig](#).

Via Agent

The [Portainer Agent](#) can be deployed to Docker or Kubernetes and works as a proxy for the respective environment's API. The agent's port needs to be exposed so Portainer can connect to the agent to manage the environment.

Via Edge Agent

The [Portainer Edge Agent](#) is a special variant of the Portainer Agent that doesn't need to expose ports for external access. Instead, it connects to the configured Portainer instance and establishes a reverse tunnel that Portainer can then use to manage the environment.

Via Socket

If Portainer runs inside Docker, it can use the Docker Socket (usually `/var/run/docker.sock`) to manage the environment.

In the case of managed Kubernetes environments, Portainer allows you to create a Kubernetes cluster at one of the following cloud providers right from within Portainer: [Civo](#), [Linode](#), [DigitalOcean](#), [Google Cloud](#), [Amazon Web Services](#), and [Microsoft Azure](#).

Once you add an environment, you can onboard your team and implement fine-grained access control policies to give everyone the ability to do their work while keeping the environment secure from unwanted access.

Environments can be organized in so-called *groups*, which can be used to limit user access to the respective environments. One option is to group environments by purpose (e.g., by development or production).

Onboarding Users and Managing Access Rights

Next to its internal authentication mechanism, Portainer offers integrations with industry-standard authentication protocols like LDAP, Microsoft Active Directory, or OAuth, as shown in [Figure 3-2](#). This allows you to import and sync your existing user base from external systems and reduces management overhead in large enterprise organizations.

Using one of the advanced authentication mechanisms comes with a few advantages in terms of user provisioning:

Single sign-on

Users will be logged in automatically if they are already logged in to the authentication provider (this is only available with OAuth).

Automatic user provisioning

If users log in to Portainer for the first time, they will be created automatically (this is only available with OAuth).

Automatic team membership

Depending on their group membership in the authentication provider, users will be assigned to teams with the same name inside Portainer automatically.

Automatic admin mappings

Users who are members of a specific group in the authentication provider will automatically be promoted to Portainer administrators.

Authentication

Configuration

Session lifetime 

1 year



 Changing from default is only recommended if you have additional layers of authentication in front of Portainer.

Authentication method



Internal
Internal authentication mechanism



LDAP
LDAP authentication



Microsoft Active Directory
AD authentication



OAuth
OAuth authentication

Figure 3-2. Portainer's authentication options

In Portainer you can organize users in so-called *teams* and assign *roles* to users and teams for environments or groups of environments. This enables you to derive access and permissions of users for environments from their group memberships in an authentication provider if you're using LDAP/AD or OAuth to provision users.

Portainer comes with five preconfigured roles:

Environment administrator

Has full control over an environment but lacks the ability to make changes to the infrastructure that underpins it.

Operator

Can control resources deployed within a given environment.

Helpdesk

Has read-only access to resources deployed within an environment without the ability to make changes to resources.

Standard user

Only controls resources they deploy. If the user is a member of a team, they control the resources that users of that team deploy.

Read-only user

Has read-only access to resources they are allowed to see (e.g., resources created by members of their team or public resources).

By integrating with industry-standard authentication providers like LDAP, Active Directory, and OAuth, Portainer checks many boxes in highly regulated enterprise environments and allows you to control environment access for users from a centralized location.

Deploy Containers

Once your environments and users are set up, you can start deploying your applications. [Figure 3-3](#) shows a few of the options available when deploying using a manifest. Portainer coined the term “ClickOps” for its very special user experience, as it started out by simplifying the CLI-heavy workflow of Docker’s early days by providing easy-to-use forms in a streamlined web GUI to make it easier for nontechnical users to achieve their goals.

'Advanced deployment'

<> Deploy

📄 Logs

Build method



Repository

Use a git repository



Web editor

Use our Web editor



URL

Specify a URL to a file



Template

Use an Edge stack template

Deployment type



Kubernetes

Kubernetes manifest format



Compose

docker-compose format

Figure 3-3. Portainer's Wizard to deploy docker-compose stacks to Kubernetes

Today there are two options available:

Use a form-based wizard

This is the classical way to deploy containerized workloads in Portainer. You will be guided by a form-based wizard that abstracts away Docker's complexity and gives you clear instructions on what to do.

Use a manifest

Over time, Portainer evolved into the enterprise space and became more attractive to experienced engineers who didn't want to click through forms but instead wanted a more technical approach to application deployment. As such, you now have additional ways to deploy your containers that better fit the [infrastructure as code movement](#) of the recent years.

If you use a manifest, you can:

Deploy from repository

This is the [GitOps](#) way of deploying applications. You can connect a Git repository that Portainer constantly monitors. Changes to the repository will be deployed to the environment automatically.

Deploy from web editor

Next to the form-based wizard, this is the most convenient way to “get going quick.” You can paste a deployment's YAML manifests into the

web editor directly.

Deploy from URL

Deploying from URL is the middle ground between a deployment from a repository and the web editor. It's a one-off deployment, where Portainer reads YAML manifests from an external URL and deploys them to the environment. In contrast to the repository approach, Portainer does not continuously monitor the URL for changes.

Deploy from template

This makes use of Portainer's internal template system with which you can define customized and reusable templates right from within Portainer.

Paired with its fine-grained access-control mechanisms and its ability to manage a variety of container orchestrators, Portainer is a great choice for organizations that handle diverse container workloads and have teams with mixed skill sets when it comes to container management. It's up to each team how it deploys its applications, and Portainer makes sure that the final result is a stack of running containers in the right environment.

Chapter 4. Portainer Use Cases

In [Chapter 3](#) you learned about the most important features of Portainer with regard to the challenges of cloud native architecture. In this chapter I will outline specific use cases built upon the aforementioned features that you can incorporate into your own infrastructure and workflows. You will learn how to give your team tiered access to resources in your managed Kubernetes clusters, deploy complex applications to them, and manage them with external tooling like [kubectrl](#), [k9s](#), or [Lens](#).

Managing Access to Kubernetes with RBAC

RBAC is short for [role-based access control](#), a widely used mechanism to restrict systems access to authorized users. This is done by defining roles with specific privileges and assigning them to users or groups of users, as shown in [Figure 4-1](#).

Portainer's built-in roles map to cluster roles and namespace roles inside a Kubernetes cluster. Each role gives access to certain resources and the respective actions (or “verbs,” as they are called in Kubernetes) like *get*, *create*, or *delete* that are available for these resources inside the cluster.

You can learn more about the Portainer roles mapped to Kubernetes roles in

the [official documentation](#).

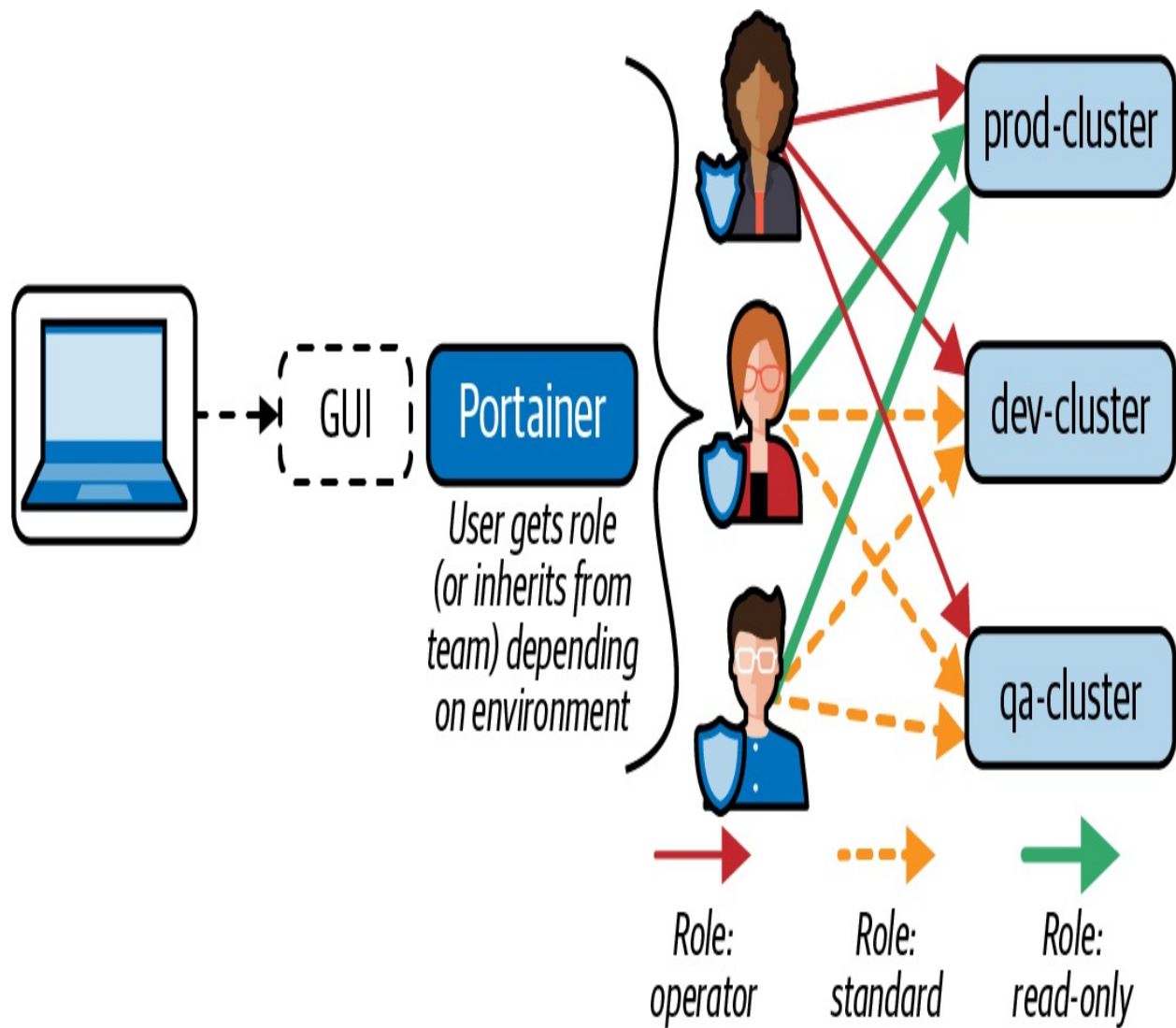


Figure 4-1. Overview of Portainer's RBAC features

Roles can be assigned to teams or users in each environment, as shown in [Figure 4-2](#). Users of a team inherit the roles of their team. Depending on your role, you gain a certain level of access to an environment and its resources (e.g., containers, ConfigMaps, etc.). A few examples include:

- View or edit Namespaces.
- View, create, or delete containers, Secrets, or ConfigMaps.
- Create or edit storage volumes or configurations.

Create access

⚠ Adding user access will require the affected user(s) to logout and login for the changes to be taken into account.

Select user(s) and/or
team(s)

 DevOps Team ▼

Role

Operator ▼

+ Create access

Figure 4-2. Assign roles to teams or users

Using RBAC in combination with a centralized authentication provider and group sync, you can easily grant multiple, distributed developers access to resources in Kubernetes, depending on their role in your organization or the team they are part of.

Using Portainer as a Kubernetes API Proxy

Portainer can work as an API proxy for Kubernetes, as shown in [Figure 4-3](#). As opposed to connecting through the web UI, as [Figure 4-1](#) shows, with the API proxy you can connect external tooling like kubectl, k9s, or Lens that needs proper configuration in the form of a Kubeconfig file to work with a Kubernetes cluster.

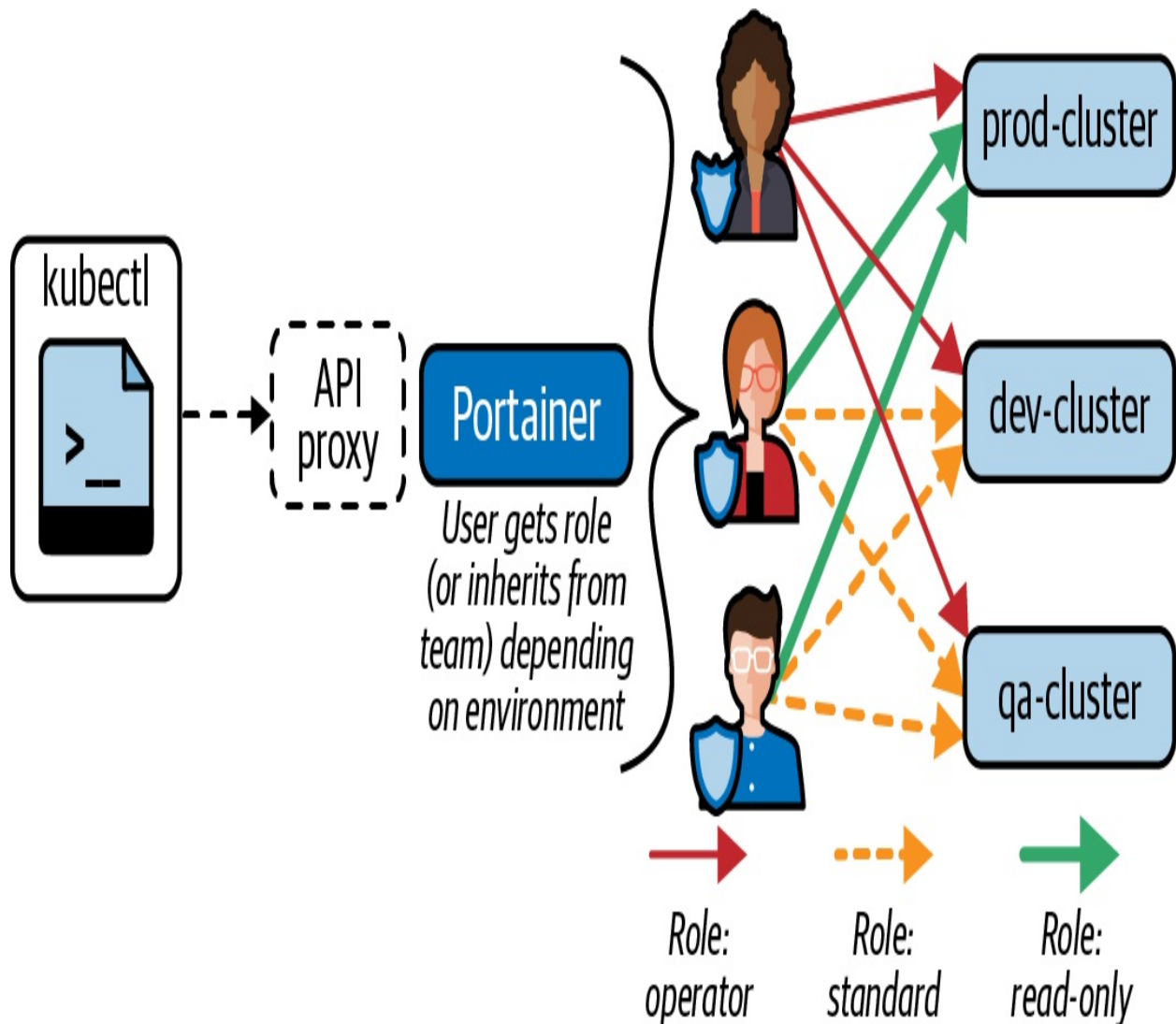


Figure 4-3. Overview of Portainer as an API proxy for Kubernetes

Portainer allows you to download a special Kubeconfig file that has the Portainer instance configured as the API endpoint for each environment that is available to you.

With this Kubeconfig file, you authenticate against Portainer with a special user account that inherits your user's roles and thus privileges on the respective clusters, so working with external tooling gives you the same level

of access as working directly from within the Portainer GUI. Some people, like me, prefer to use specialized tooling when interfacing with Kubernetes clusters, which makes this feature a game changer in enterprise environments.

Without Portainer, cluster administrators would need to carefully create accounts and roles inside a cluster themselves when users need to access it with CLI tooling that needs a Kubeconfig file. And from personal experience, I can tell you that this is something you don't want to deal with regularly. Having Portainer as a proxy, taking care of the necessary security layers, removes the burden of keeping the cluster secure from unwanted access from the shoulders of the cluster administrators and shifts it to a centrally managed control plane with well-designed RBAC rules.

Building Container Platforms for Cloud Native Teams

[Figure 4-4](#) shows the environment overview, where you can review accumulated information about an environment at a central spot. The overview allows you to jump directly to the referenced objects.

Working with cloud native architecture patterns involves quite a few services apart from the actual workloads you might build. These include:

- A container registry

- A service catalog
- A tool for continuous deployment
- A tool for dealing with TLS certificates
- Databases

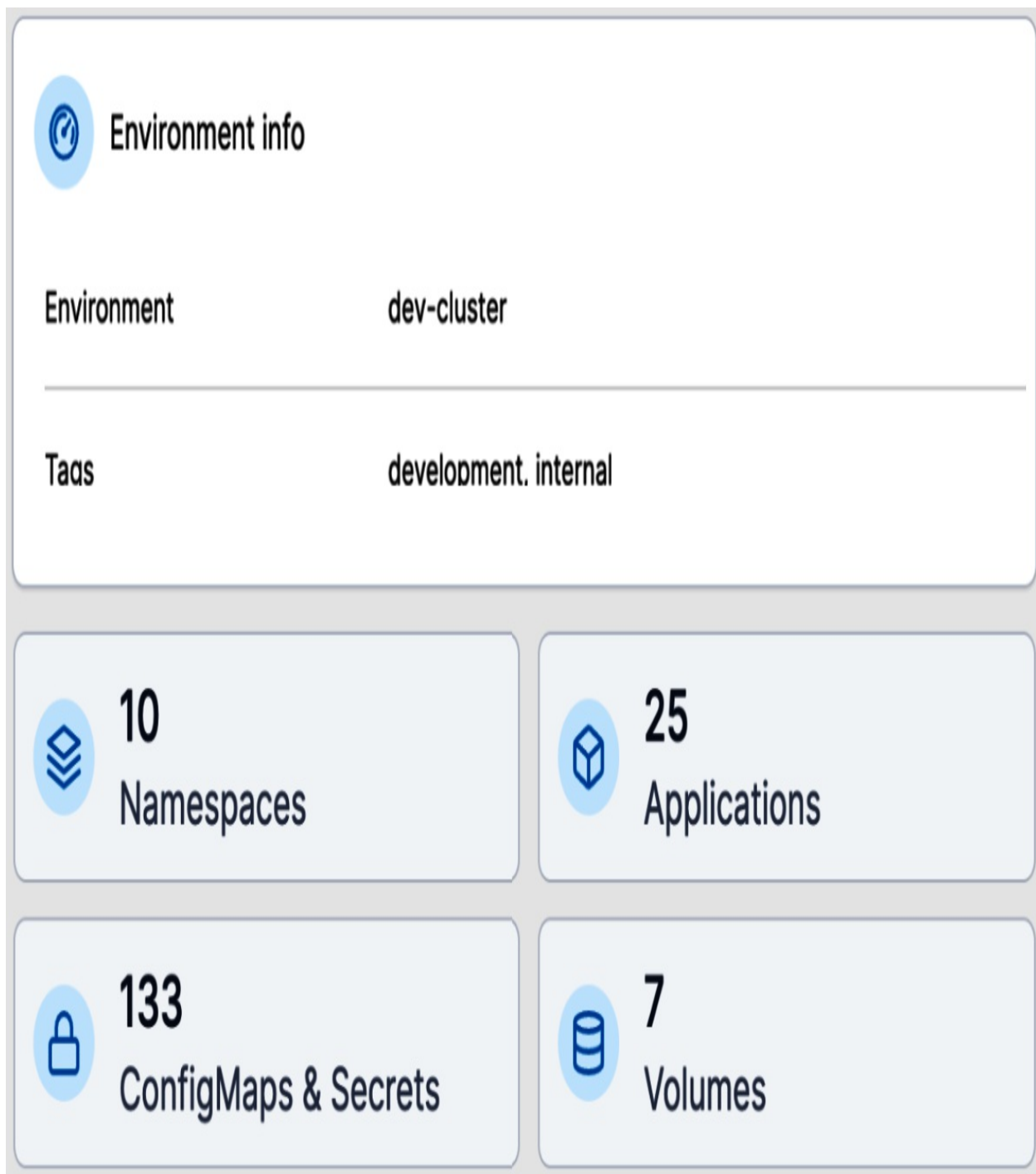


Figure 4-4. Environment overview










Portainer makes it easy to deploy and manage all the applications a team working with cloud native architecture patterns needs. Environment operators

can provide helper tools like a container registry or a service catalog, while standard users (i.e., developers) can deploy their own workloads to an environment, making use of the helper tools without themselves being able to manipulate them (see [Figure 4-5](#)).

Namespaces

[Remove](#)[+ Add with form](#)[+ Create from manifest](#)

 System resources are hidden, this can be changed in the table settings.

| <input type="checkbox"/> Name  | Status  | Quota  | Created  | Actions |
|---|--|---|---|---|
| <input type="checkbox"/> argo-workflows | Active | - | 2022-08-31 21:35:48 |  Manage access |
| <input type="checkbox"/> cert-manager | Active | - | 2022-08-11 13:27:04 |  Manage access |
| <input checked="" type="checkbox"/> default | Active | - | 2022-07-28 14:49:08 | - |
| <input type="checkbox"/> external-dns | Active | - | 2022-08-04 14:27:33 |  Manage access |
| <input type="checkbox"/> harbor | Active | - | 2022-08-11 13:23:26 |  Manage access |
| <input type="checkbox"/> ingress | Active | - | 2022-08-11 15:26:52 |  Manage access |

Items per page

10 

Migrating Legacy Workloads to Kubernetes

Portainer has a long history with Docker, and many people and organizations use it to manage their Docker containers or [docker-compose stacks](#). One challenge that arose with the advent of Kubernetes was its increased complexity compared to Docker or docker-compose. Many organizations can't tackle this complexity due to lack of experience or budget.

In Portainer, you can simply import existing docker-compose stacks, and Portainer gladly converts them for you to Kubernetes manifests with the help of [Kompose](#). In most cases, this conversion works seamlessly and thus enables you to safely move between Docker and Kubernetes environments right from within Portainer.

Portainer offers a range of tutorials outlining the process of migrating your legacy applications to Kubernetes:

- [Deploy a compose-based app to Kubernetes using Kompose](#).
- [“How to Transition from Docker/Swarm to Kubernetes with Portainer”](#).
- [Deploy applications to Kubernetes](#).

This functionality helps to bridge the gap between single-node Docker

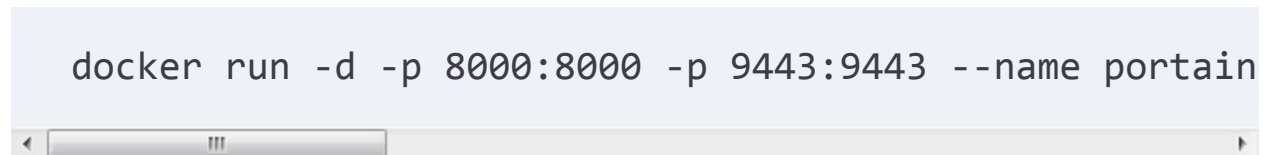
environments and Kubernetes clusters. You don't need to understand Kubernetes's YAML manifests to migrate your applications to Kubernetes, but you can start by feeding Portainer an existing Docker Compose stack and then dive deeper into Kubernetes's configuration language by analyzing and adapting the translated manifests.

Replace Docker Desktop for Container Management

Docker Desktop offers a dedicated user interface for the management of containers, images, and volumes, but it comes with a few strings attached. For one, that user interface is only available to Windows and macOS users but not for Linux users. Additionally, Docker Inc.'s licensing changed a lot over the years, rendering Docker Desktop unusable for large enterprises due to its license constraints.

Portainer can be installed on any Docker engine with a single command and provides a common user interface to manage containers on all operating systems:

```
docker run -d -p 8000:8000 -p 9443:9443 --name portainer
```

A screenshot of a terminal window with a light blue background. The command 'docker run -d -p 8000:8000 -p 9443:9443 --name portainer' is displayed in a monospaced font. Below the command bar is a horizontal scrollbar with a small handle in the center.

On top, Portainer also works with alternative container engines like [Podman](#)

or [Lima](#) and provides the same workflows and user experience for any of them.

This makes Portainer the ideal replacement for Docker Desktop, especially in environments with licensing constraints or alternative container engines.

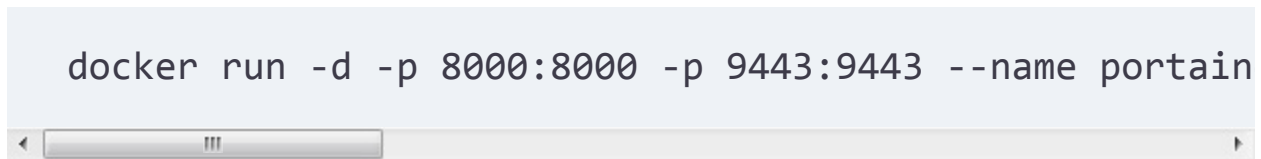
Wrap-up

In the first two chapters, I outlined why a tool like Portainer is actually needed: while the software development ecosystem has evolved a lot in the last few years, the problems developers face have also evolved. Portainer is a great fit for an agile and distributed development environment and has many additional features in addition to the ones I covered in Chapters [3](#) and [4](#). I encourage you to catch up on them through [Portainer's documentation](#), YouTube, or another of the many articles and blog posts on the topic.

Depending on your own use cases, you might find Portainer valuable for totally different reasons: for example, as the container dashboard for your homelab, as a multitenant control plane for a managed services offering, or to quickly start containers from [preconfigured templates](#). My personal background is in infrastructure, software delivery, and distributed development environments, so I naturally picked use cases that I'm familiar with. However, getting started with Portainer and exploring what it can do for you is only a single command away, and I encourage you to get your hands

dirty right now:

```
docker run -d -p 8000:8000 -p 9443:9443 --name portainer
```

A screenshot of a terminal window with a light blue header bar. The terminal shows the command 'docker run -d -p 8000:8000 -p 9443:9443 --name portainer' in a monospaced font. Below the command bar is a scrollbar with a grey track and a white slider.

Once it's running, you can access it on <https://localhost:9443> in your browser.

I tried to avoid step-by-step instructions on how to use Portainer in this report because this has been covered numerous times by the sources mentioned. I aimed to provide context as to why the tool exists and how it can help to solve higher-order problems you might encounter in your daily work. Given its popularity these days, it made sense to focus on Kubernetes for this report, but Portainer makes it equally easy to work with Docker or Docker Swarm environments, which are still very prevalent in our industry, albeit predominantly in on-premises environments. Since Portainer isn't a SaaS solution but can easily be deployed to on-premises infrastructure, it's worth a look even if Kubernetes is not part of your tech stack (yet). After all, Portainer is about containers at large.

As we all know, the level of abstraction in technology is constantly increasing, but now you're equipped with a unique understanding of the problem space and a viable solution for some of the challenges you will encounter when developing and deploying software in a cloud native environment. I hope you found value in reading this report and have a clearer

perspective on today's software development ecosystem—and of course Portainer.

About the Author

Fabian Peter is founder and chief executive officer at cloud native consulting and service provider ayedo. Fabian has been in the industry for more than 10 years, founded the Cloud Native Meetup Saar and currently serves as a Portainer Ambassador. Fabian's main expertise is system and platform architecture, product development, and bootstrapping digital businesses. When not building software, Fabian enjoys playing and listening to music and taking long walks with his dogs.