O'REILLY®

Modernize Applications with Apache Kafka

Discover the Next Generation of Messaging for Your Applications

Jennifer Vargas
& Richard Stroop

REPORT

Red Hat

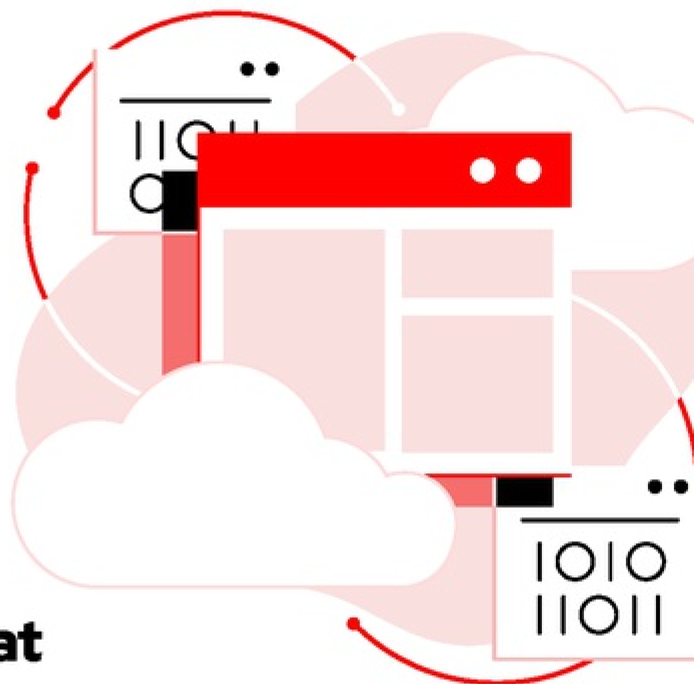
Connect data streams

Say goodbye to isolated data

Connect data in real-time across applications, APIs, devices, and edge computing using cloud-native technologies. Red Hat® OpenShift® Streams for Apache Kafka provides:

- Kubernetes-native application development, connectivity, and data streaming.
- A unified experience across different clouds.
- An ecosystem to streamline real-time data initiatives.

Start your data streaming journey



Modernize Applications with Apache Kafka

Discover the Next Generation of Messaging for Your Applications

Jennifer Vargas and Richard Stroop

Beijing • Boston • Farnham • Sebastopol • Tokyo

Modernize Applications with Apache Kafka

by Jennifer Vargas and Richard Stroop

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: Aaron Black
- Development Editor: Gary O'Brien
- Production Editor: Clare Laylock
- Copyeditor: nSight, Inc.
- Proofreader: Clare Laylock
- Interior Designer: David Futato

- Cover Designer: Randy Comer
- Illustrator: Kate Dullea

- April 2023: First Edition

Revision History for the First Edition

- 2023-04-14: First Release

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Modernize Applications with Apache Kafka*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

This work is part of a collaboration between O'Reilly and Red Hat. See our statement of editorial independence.

978-1-098-14335-0

[LSI]

Acknowledgments

We would like to extend our sincere gratitude to Duncan Doyle for his contributions to this report. He played a key role in supporting the structure and writing of everything we produced. We also want to thank other members of the Red Hat team that made this possible: Coco Jaenicke, Eric Johnson, and Monica Hockelberg.

Chapter 1. The Path to Application Modernization

Application modernization consists of taking existing legacy applications and systems and then transforming their platform infrastructure, internal architecture, and/or features to improve agility, decrease time to market, create better experiences for end users, and improve application performance along with scalability. Most of the modernization efforts today are centered around migrating legacy monolithic applications to cloud native applications, allowing for easier maintenance and optimized resource utilization.

Cloud native application development is a methodology that aims to design and build applications and systems that fully use the cloud computing model.

Cloud infrastructure’s dynamic and on-demand scalability nature promotes an architectural model in which applications consist of a series of discrete, small, independent, and loosely coupled units called microservices. These microservices can be individually built, deployed, and operated, enabling agile development and easier operation of these individual components. Modern application platforms are often designed to support cloud native application development.

Modern application platforms that provide a scalable and comprehensive environment for automated application development and deployment tend to

accelerate the adoption of application modernization. These platforms provide newer frameworks, tools, and services that promote the development of applications using cloud native patterns, while facilitating developer collaboration, technology adoption, increased productivity, and business agility.

Many legacy applications and systems play a critical role in business operations, and migrating them can present a big challenge to organizations. Migrating applications can hinder the ability to add new functionality to existing or new applications, which can reduce the team’s ability to respond quickly to market changes. Other concerns during this process include security, data management, and cost management.

Organizations can benefit from adopting a phased approach when modernizing legacy monolithic applications into simpler microservices. A phased approach requires understanding the application modernization goals of your organization, evaluating and understanding the existing monolithic legacy applications, and determining the optimal modernization approach for each existing legacy application. There are a variety of techniques available for modernizing applications, and not all of them require breaking down your monolithic application. No matter how you accomplish your modernization goals, one thing is certain: your applications will need to be able to communicate in a modern way or they will remain bound by their current limits of connectivity.

Reasons for Modernizing Applications

Application modernization involves evaluating and transforming any legacy applications and their underlying platforms, as well as understanding and adapting the development processes and methodologies used within the organization. The goals are to enable innovation, increase business agility, and deliver better security at every level of the stack.

Legacy monolithic applications are often hard to maintain, secure, update, and scale. Application modernization can provide applications with the improved elasticity, availability, and scalability required for modern digital experiences.

Building agility into an organization starts with making application development processes easier to adapt and change. Agility in the application can be achieved by modularizing codebases into smaller microservices, increasing the number of automated tests to improve application quality, and implementing reliable and repeatable automated build and deployment pipelines. A consistent, reliable, repeatable, and predictable application development and delivery process allows organizations to work faster and more efficiently. *Faster time to market* and the flexibility to easily adapt to change are some of the most important benefits of application modernization.

Microservice applications are *simpler to scale horizontally*, allowing organizations to dynamically scale when application usage increases and

reduce utilization rather than reduce usage. This could lead to potential improvements in resource utilization and optimization. *Customer experiences* can also benefit as microservices deliver better system performance, better response times, and fewer disruptions and interruptions on frontend applications.

The use of cloud computing and cloud resources has been increasingly growing as more organizations decide to move some or all of their workloads to the cloud. This leaves organizations with a hybrid IT landscape where workloads are distributed across private data centers using traditional deployment models, private clouds, and one or multiple public clouds. Defining clear, standardized, loosely coupled, manageable, and *governed* APIs across these applications is key when building distributed systems and applications that might span the hybrid cloud.

Time is one of the most important resources a developer needs to take care of, while *developer productivity* is an increasingly important metric for a development team. Cloud native development and containerization have emerged as the technologies that can accelerate time to market and improve developer productivity. This is because they allow developer teams to spend more time on designing, developing, and testing their applications instead of spending time on getting access to required infrastructure. Automated provisioning and immediate access to infrastructure are now available.

Making data available close to the workload that requires data is key when

building high-throughput, low-latency applications. In addition, efficient data distribution across the hybrid clouds can greatly *reduce the operational costs* of your IT landscape from inter-cloud data transfer. Therefore, it is crucial to be able to transfer data between clouds efficiently and reliably.

Using different clouds—be they public and/or private—provides an organization with operational flexibility. Cloud infrastructure is often used for bursty workloads, seasonal workloads, or workloads that require a lot of compute power for a short period of time (e.g., training machine learning models). The dynamic nature of the cloud, which gives you the *flexibility* to spin up, spin down, and compute on demand, is ideal for these kinds of workloads. Other more stable, predictable workloads can benefit from potentially lower operational costs when running in a private cloud or data center. A common pattern in hybrid and multicloud scenarios is that different workloads run on different public clouds (based on the services available) or on private clouds for cost, security, and data regulation reasons.

IT development teams and developers are also under pressure to design applications with security in mind. Application modernization plays a role in *security and compliance* for the organization. Application modernization not only brings changes to the code of the application but also propels adoption of modern platforms and technologies. Newer technologies and modern application platforms, such as container orchestration platforms, put some emphasis on security and compliance. Container orchestration platforms allow scanning for security vulnerabilities and defects, automating the

application delivery process for eliminating errors during deployment, and promoting secure designed applications from scratch. Finally, application modernization promotes the use of newer security and compliance standards that take into account the security and vulnerability challenges of maintaining cloud native applications.

Application Modernization Techniques

The path to modernizing applications can differ from application to application. There is no need to break down every monolithic application into microservice applications. Organizations could choose to minimally update legacy applications, leave them untouched, or replace them with new software.

Application modernization requires an organization to understand the current state of its legacy applications, their dependencies with other systems, and the software requirements to run them and maintain them. It is also important to evaluate the pace at which you can add new functionality and capabilities to the existing applications. These factors will help determine which applications to consider first.

The next step is to define the organization’s goals for each specific application. You may want to improve your team’s agility when delivering new capabilities or reduce costs from software licenses and reduce vendor

lock-in.

The pace of modernization for applications requires understanding which applications are critical to the business and deciding on how to offset the old before adopting the new. There are different modernization techniques to help organizations decide how to modernize. We will quickly discuss the five recognized approaches to application modernization:

Retention

For those applications that don't require change to support the business, the easiest thing you can do is to retain what you have and postpone modernizing the application for the foreseeable future.

Retirement

For those applications that are no longer in use or whose functionality can be easily covered by other applications, the best course of action is to retire them and repurpose their computing resources.

Rehost

This approach takes an application as is and hosts it on new infrastructure such as cloud infrastructure or a Kubernetes platform. The goal is for applications, existing integrations, and dependencies to remain unchanged.

Replatform

This approach requires redeploying an application in a modern runtime environment such as running in a container on a Kubernetes platform. This

requires a bit of application alteration without changing its architecture.

Refactor
This approach rebuilds your legacy monolithic application as microservices to support faster release cycles and optimize application performance. Refactoring involves rebuilding and re-architecting your application.

It is important to recognize the complexity of each modernization technique. Each technique has a different level of effort (time and cost), and its impact will differ on the business and the application itself. Figure 1-1 provides general guidance for understanding which technique could bring the most value when thinking of modernizing your application, but more value will also require more effort.

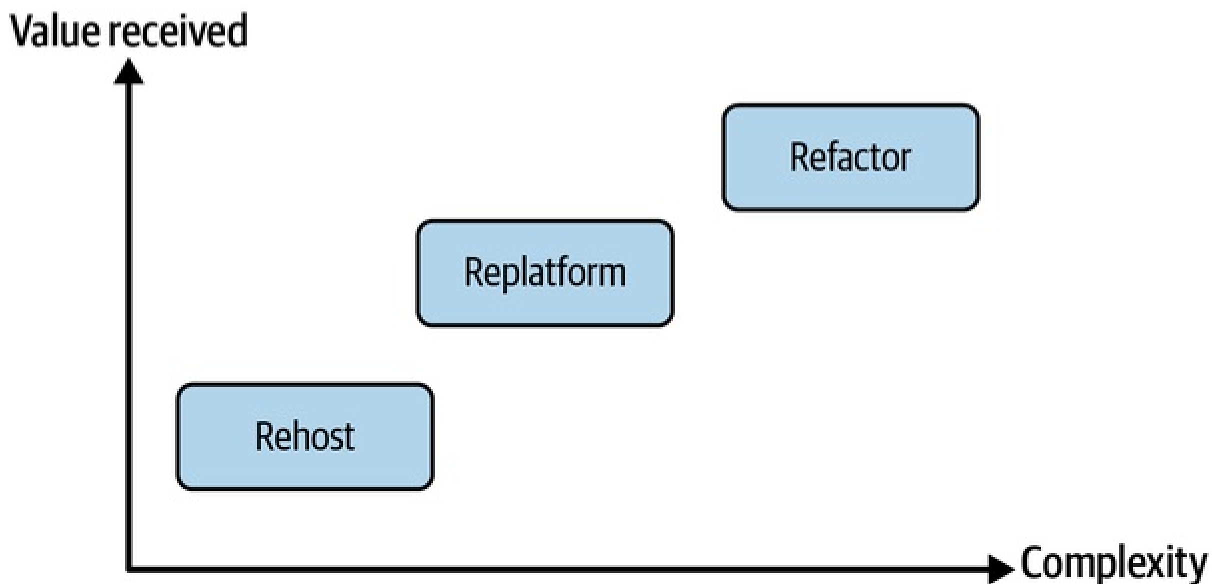


Figure 1-1. Complexity versus value received for application modernization techniques

For the rest of this book, we will be focusing on application modernization from the perspective of replatforming and refactoring since we see the most value there. The complexity can be mitigated with technologies and techniques that we will discuss further. But first, let’s explore the challenges that often arise when taking on these approaches.

Challenges of Modernizing a Monolithic Application

Migrating a monolithic application to a microservices application requires looking into things like APIs, latency, and parallel processing.

Properly defining APIs for both monolithic and microservice applications is an important process during migration. Explicit, well-defined APIs can create clear boundaries between different components of an application and make it easier when migrating to a microservices architecture. We recommend using an API-first approach during application modernization to facilitate communication with monolithic applications and other systems. APIs can be maintained during application modernization while gradually switching to microservices and even when testing new functionality in the new microservices.

When moving from a monolith to a microservices architecture, operations that need to be called many times in a row, known as *chatty APIs*, are a

concern. As most communication within a monolith is done in-memory, there is not much overhead. There are really no penalties when creating a chatty API because an in-memory method call does not require serialization and deserialization of request and response parameters, and there is no network overhead, no security, and no encryption, etc. When moving to a microservices architecture, these chatty APIs can become problematic because communication between microservices includes the overhead of all the things listed earlier.

Another challenge of application modernization is *latency*. Latency will increase when remote calls replace in-memory calls. Legacy monolithic applications use in-memory calls, while loosely coupled microservices use remote calls. One reason latency increases is the unavailability of services. Therefore, organizations need to make sure the new microservice applications and any existing legacy applications are able to handle failures and timeouts without affecting the user experience.

Parallel processing is another challenge when migrating a monolithic application to microservice applications. Normally, a monolith's functionality and components will be refactored into various smaller microservice applications. When doing application modernization, you need to be aware that components and services can have multiple instances running simultaneously and in parallel. Often, monoliths are designed assuming a single instance of a given component, requiring messages and requests to be executed in a certain order. For those monoliths to enable parallel processing,

further architectural considerations are required. On the other hand, microservice applications introduce multiple service instances processing messages and requests in parallel. When applications make use of parallel processing, the order of execution of operations is often not guaranteed. For this reason, it's often said that it's harder to move from one instance of a service to two than it is to move from two instances to many. Parallel processing concerns can be offset by using an asynchronous architecture and communication pattern.

The challenges of modernizing applications are often offset by adopting newer technologies, like container orchestration platforms, application framework patterns, and modern messaging architectures. But adopting new technologies requires application developers to spend extra time learning those technologies, thus decreasing the time spent on maintaining existing legacy applications. Organizations can opt for hiring developers with previous experience on the technologies, but often employees have to be trained on new technologies before they can be effective when creating new applications. No matter what level of experience someone has, modernizing applications will inherently come with new challenges based on the maturity of the whole company. By following common patterns while modernizing, these challenges can be somewhat mitigated.

Microservices Architecture and

Communication Patterns

Modernizing applications gives organizations a chance to design the applications of the future, especially when considering multicloud and hybrid-cloud deployments, artificial intelligence/machine learning (AI/ML), and emerging technologies. The introduction of cloud native application development drives IT developer teams to identify software and hardware requirements, development processes, and emerging technologies that will allow the business to respond fast to market changes. Cloud native application development is an approach for building and updating applications that are responsive, scalable, and fault-tolerant in a fast manner while improving quality and reducing risk.

Cloud native application development is a pattern that is used when creating applications that considers the latency, unavailability, and concurrent nature of the services being deployed while architecting them. Even if the application is never deployed to a cloud, these considerations allow it to function better in a distributed environment. Adopting this pattern and others, such as circuit-breakers and asynchronous communication, allow applications to feel responsive in uncertain environments.

As discussed in the last section, one of the challenges when modernizing monolithic applications to microservices is defining the communication layers and architectural patterns to use between them. Microservice

applications are independent, loosely coupled, and distributed, requiring communication patterns and protocols that can support microservice-to-microservice communication. Loose coupling in software systems is a design principle focused on minimizing dependencies between components. When there are fewer dependencies, the components can evolve freely without impacting other components in the system, thus making development teams more autonomous. Loose coupling allows organizations to deal with requirements of scalability, flexibility, and fault tolerance in addition to improving the resiliency of the systems.

There are various forms of coupling between components in a system:

Temporal

Temporal coupling exists when components in a system are coupled by time. A simple example of this is calling a RESTful service over HTTP. As HTTP is a synchronous protocol, the client expects the service it calls to be available at the moment the call is made. Hence, there is a temporal coupling between client and service, as they need to be available at the same time.

Location

Location coupling exists when components in a system directly communicate with each other on a static address. The component initiating the interaction requires the other component to be available on the address it knows, limiting the option of moving that component to another location

(e.g., server, container, cloud).

API

When clients communicate with services, they do so by calling an API or sending a message to an API. This creates a coupling between the clients and services and limits the changes that can be made to the API without impacting the clients.

Data type/format

Components in a software system communicate with each other by transmitting data. For two components to understand each other, they need to send the data in the data format and data type that the other component expects and can process. Changes in the data format or data type on either side of the communication channel can cause communication failures.

Microservice communication protocols and patterns can heavily impact the overall architecture of your service landscape. In principle, there are two communication styles: synchronous and asynchronous. Synchronous communication means that when a client sends a request to a service, the client waits for the service's response before it continues processing. In asynchronous communication, the client sends a request but does not block and wait for the response. It will continue processing and either does not expect a response at all from the given service or receives a response at a later point in time.

Asynchronous communication can allow for the design of more loosely coupled architectures. It is recommended to consider different communication patterns in the target architecture. This might mean that the microservice needs to support both a synchronous and asynchronous communication style during the phased migration, or that the monolith needs to be adapted to support asynchronous communication without impacting the performance and user experience of the application.

Chapter 2. A Modern Messaging Infrastructure: The Apache Kafka Ecosystem

The needs of modern applications across hybrid-cloud and multicloud environments require that organizations look at technologies, architectural patterns, and development approaches that will support the scalability, efficiency, reliability, and throughput required in today's world. Increasing data volumes, the evolution of data ingestion, and the demand for real-time availability of data changed the way IT operations and development teams design and implement modern applications.

As we discussed in the previous chapter, microservice architectures became the preferred method for modernizing legacy monolithic applications. Communication between microservices and different systems requires considering traditional and modern messaging technologies.

Apache Kafka has emerged as one of the preferred modern open source technologies for messaging and streaming data architectures. This technology is recognized for supporting loose coupling and asynchronous event-driven communication in distributed systems, providing the capability to deliver data in near real time across applications and systems. It is a complex tool with a vast ecosystem to support it. Fortunately, there are ways to automate

Kafka’s deployment and simplify its integration into your modernized applications.

The Business Value of the Apache Kafka Ecosystem

Apache Kafka is a distributed streams processing platform that uses the publish/subscribe method to move data between producers and consumers, which can be microservices, cloud native, or traditional applications and other systems. The publish/subscribe messaging pattern is mostly used when you want to decouple systems in terms of location and availability in architectures, and when you need an event or a message to trigger one or multiple actions across disparate applications and systems.

Apache Kafka’s approach to the publish/subscribe method stands out because of its ability to send and receive messages at a very fast rate, stream vast amounts of data, horizontally scale as the number of requests increases, and retain the data even after messages have been consumed. All these capabilities allow for using Kafka in innovative use cases that couldn’t be solved using traditional messaging or processing solutions.

Apache Kafka has evolved over the years to include an ever-growing set of features and components developed, delivered, and maintained by various open source communities and vendors to support large and complex

enterprise implementations. The ecosystem of services, providers, and community projects has quickly created solutions that simplify connectivity to and from data sources, enable new ways to ingest data from various data sources, ensure data governance and discoverability, and support processing and analyzing large data volumes in real time. The power of Kafka is often augmented by the adoption of container and application platforms, cloud infrastructure, and other managed services built for supporting application development and delivery.

Apache Kafka and Its Components

The Apache Kafka product ecosystem consists of a real-time streaming broker, schema registry, connectors, and processing capabilities. The combination of all these capabilities allows organizations to deliver a complete streams processing platform that supports data streaming, data storage, and data processing.

Apache Kafka is a publish/subscribe system, often referred to as pub/sub. Pub/sub systems are characterized by senders pushing messages to a central point for classification and subscribers receiving messages of interest from the central point. These systems require a broker that acts as the central point where messages are published. Subscribers and publishers communicate through topics, which organize the messages and replicate them across brokers.

A typical Apache Kafka implementation contains multiple brokers, which provide the high availability and fault tolerance characteristics of the platform. Topics are hosted on the brokers, and each topic is split into one or more partitions. These topic partitions are distributed across the brokers in the cluster, effectively allowing consumers to consume and process records on a topic in parallel, increasing the throughput of the consumer (Figure 2-1).

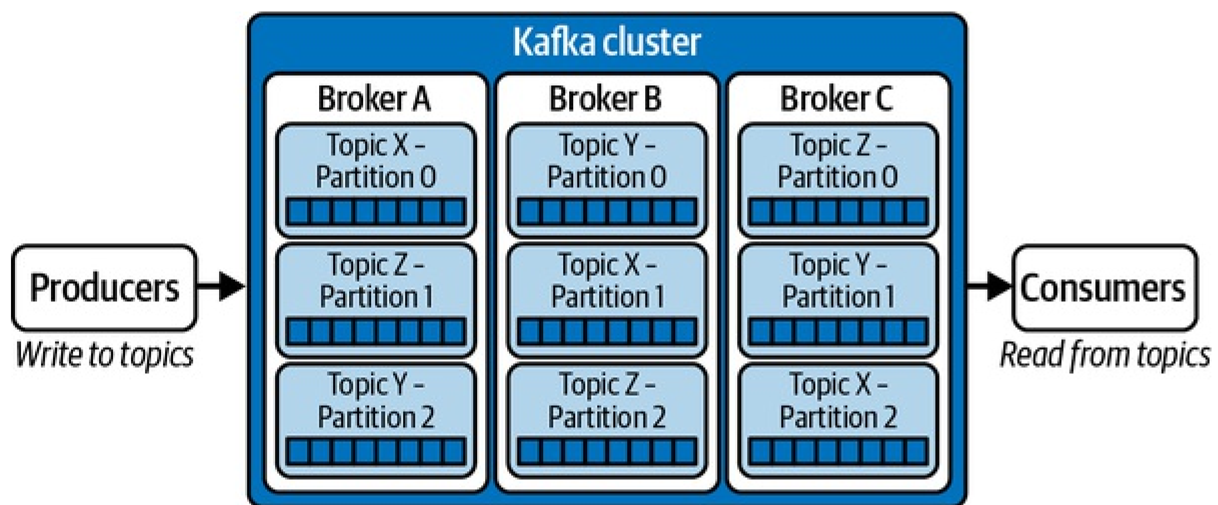


Figure 2-1. Typical Apache Kafka configuration, including brokers, topics, and partitions

A topic is a collection of messages or events that you want to group together for processing by a consumer or subscriber. A partition contains a subset of the messages written to a topic, which can be distributed for you by the broker. By distributing messages in a topic across multiple partitions, Kafka enables the parallel consumption and processing of messages. This can improve the overall performance of the system. Second, partitions are replicated across multiple brokers to ensure high availability of the data and to make the system fault-tolerant. Consumers can read from leaders or

replicas of the data to further speed up the consumption of data.

Value of the Apache Kafka product ecosystem

One thing that is important to understand is that Apache Kafka by itself does not deliver the full value. You require additional components to complete the product ecosystem. First, you need a practical way to connect multiple data sources to Kafka for getting information in and out.

Prebuilt connectors allow communication among popular source systems of both new and existing applications, allowing organizations to move at their own pace. Legacy applications can be retired safely once newer microservice applications are deployed, and organizations can still use Kafka as the data backbone for the larger architecture.

Schema design and registry can help with ensuring data consistency and governance. The messages and data streamed with Apache Kafka need to use the right data structure format. Industry-standard data formats or schemas are available for supporting communication between Kafka topics and producers/consumers. It is also important to have a place to store existing schemas, register new schemas, and provide compatibility and validation checks of the schema format used in a message. Figure 2-2 shows a logic diagram of the basic components for the Apache Kafka product ecosystem. The full ecosystem can allow organizations to deliver on multiple benefits and use cases when using Kafka as the preferred communication technology.

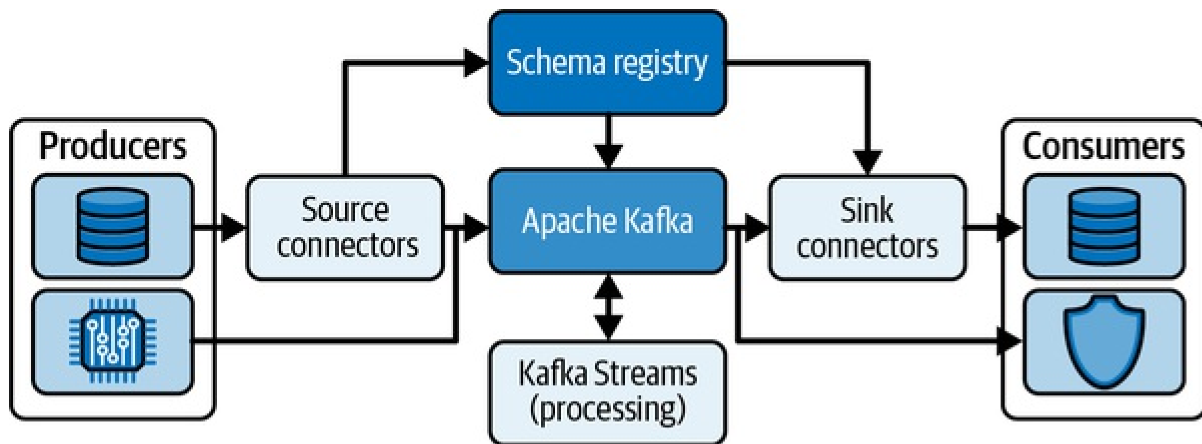


Figure 2-2. Basic components of an Apache Kafka product ecosystem

Connectors

The value of Apache Kafka is maximized when helping organizations take advantage of all the data across the business. Getting data from multiple sources to Kafka and delivering to sinks in a timely, reliable, and secure manner is one of the most important tasks. The open source community has built Kafka Connect to provide a pluggable way for accessing and extracting data from and to other applications and Kafka.

Kafka Connect is not the only way to integrate applications to Apache Kafka. Other projects like Apache Camel and Debezium provide prebuilt connectors that support integration to a variety of data sources, systems, and applications. Apache Camel is an open source integration framework that implements enterprise integration patterns, as well as a rich set of connectors from and to various systems. Debezium is an open source platform for Change Data Capture (CDC), allowing you to capture changes in relational database systems and emitting these changes as events to a streaming

platform like Apache Kafka.

The value of Kafka Connect, Apache Camel connectors, and Debezium is to facilitate connectivity between systems without requiring in-depth knowledge of the APIs of source and target systems when connecting to Apache Kafka. This ensures faster time to market, less time writing code, and less code maintenance. An enterprise-grade implementation of these open source projects can provide an easy way to deploy and delete connectors quickly, support error handling, and guarantee delivery of messages and streaming data.

Change Data Capture

A vast amount of legacy enterprise applications use relational database management systems (RDBMs) and/or NoSQL datastores to store their data and keep state. With the rise of service-oriented architectures and later microservice architectures, which advocate a datastore per service approach, a lot of enterprise data is scattered across various datastores throughout the organization. Unlocking this data and making it available for stream processing increases the data's value as it enables organizations to use this existing data to implement new and innovative use cases.

Adapting and enhancing the legacy applications and/or commercial off-the-shelf (COTS) applications and systems, unlocking their data, and making it available as streams of events is often impractical or even infeasible.

Change Data Capture captures the data and events not at the application API level but at the database level. CDC allows us to directly connect to a datastore (usually a database), and it can identify and capture the changes to the data, emitting those changes as events to a downstream system. Debezium is an example of a CDC solution that emits these captured changes to an Apache Kafka topic, making these data changes available to downstream consumers.

The use cases for CDC are vast. The technology can be used to update caches, search indexes, and derived datastores, or to make the data available for real-time stream processing and analytics. Making the data available in Apache Kafka topics as a stream of events allows any Kafka consumer to access, ingest, and process the data that was previously locked inside a database.

CDC plays an important role in application modernization, as it allows developers to tap into and use existing data, making the data available as a stream of events, without the need to modify the original system that owns the data. This enables the exploration of new use cases and innovative solutions while keeping existing systems, applications, and processes intact.

Schema definitions, governance, and control

The Apache Kafka broker focuses on handling and fetching data without worrying about the structure of the data it is transferring. Because Kafka

enables decoupling, any subscriber and publisher of data requires a base contract to understand the type of data that is being streamed or transferred between them. Schemas in Apache Kafka define the structure of the data format or contract. Contracts define the type of data that is being published to a topic. These contracts allow data to be serialized and deserialized from compact binary formats, allow data sharing between publishers and subscribers, and prevent data conflicts.

Standard formats have been created for schema contracts to allow serializing and deserializing messages. The most used formats for serialization/deserialization are Avro, JSON, or Protobuf. Serializing messages and events to compact (binary) formats reduces the amount of data that needs to be transferred to and from Apache Kafka, improving the performance and throughput of the system. Smaller messages also reduce the amount of storage needed by the brokers to persist and retain the messages on the topics.

A schema registry can support an Apache Kafka implementation by providing a place to store existing schemas, register new schemas, and provide compatibility and validation checks when a new version of a schema is introduced. This validation guarantees backward compatibility and prevents breaking message consumers. It can greatly improve the productivity of multiple development teams by providing a common repository for easy sharing and discovery. It also provides the necessary tools to govern and manage schema and data evolution as part of a software

development lifecycle.

Validation of schema happens when a publisher checks the schema format against the schema registry. If the schema exists in the repository, the message is safely transferred to an Apache Kafka topic for subscribers to read. If a schema is unavailable in the repository, there are two options: you can update an outdated schema, or you can create a new schema in the registry. Schemas can also be checked for backward and forward compatibility.

Processing data using Kafka Streams

Another important piece in the Apache Kafka ecosystem is the processing capability. As mentioned earlier, Apache Kafka does not need to know what the messages contain when streaming between publishers and subscribers. Regardless, many Apache Kafka implementations can benefit greatly from supporting data processing and storage before a particular subscriber consumes it. Some examples of stream processing use cases are filtering events and messages, combining events from multiple topics to create new higher-value data, and rekeying messages to route them to specific topic partitions.

Kafka Streams is an external component developed to support processing data streams using minimal code through a domain-specific language (DSL). Kafka Streams was designed as a simple and lightweight client library that

provides fault tolerance, parallel processing, and scalability.

This stream processing technology offers two ways to define topology: the DSL and the API. Kafka Streams DSL provides common data transformation operations (map, filter, join, and aggregate), while the Kafka Streams API allows developers to define and connect custom processors and to interact with state stores. Another option for processing that can complement Apache Kafka is Apache Flink. Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments and to perform computations at in-memory speed and at any scale.

Replication with MirrorMaker

Any messaging infrastructure requires access to the organization's data, regardless of where that data is created and where it needs to be consumed. The Apache Kafka ecosystem provides various tools and technologies, like MirrorMaker, to enable data replication across disparate Kafka clusters. MirrorMaker provides various replication options, including bidirectional replication and replication of a subset of topics to a target Kafka cluster. These replication options allow you to optimize your data replication across the hybrid cloud in terms of both data location and availability, as well as cloud infrastructure and inter-cloud data transfer costs.

When building applications in a hybrid-cloud environment, Apache Kafka can give you the tools and technology to distribute data across your environments effectively and efficiently, providing you the flexibility to deploy your workloads in a way that maximizes your operational goals.

There is significant value in understanding the components offered by Apache Kafka and its product ecosystem. Kafka enables a variety of use cases that promote the adoption of modern frameworks and technologies geared toward supporting modern digital requirements. This is further improved when paired with container orchestration platforms, since it allows delivering on more audacious goals, such as automated delivery, automated operations, faster development, and resource optimization and utilization. But with a powerful messaging platform comes a large amount of responsibility and challenges that must be addressed.

Challenges of Using Apache Kafka

Setting up and managing an Apache Kafka cluster is not simple. Although it often works well out of the box for simple demos, the complexity involved with running realistic workloads quickly becomes apparent.

The first issue is that your broker, topics, and clients all have to be tuned. The broker sets how many replicas have to be in sync in order for it to let producers send more messages. It also controls how often data is actually

flushed to disk, which has a large impact on performance. The topics have to be tuned to have enough partitions to send data to all of the concurrent consumers, as well as to set limits on the amount of data and time that messages are stored. And unlike with traditional brokers, the clients have to be aware of commits and offsets that take excessive time to sync if done after every message. The clients can be tuned to increase these intervals and batch messages to save time on overhead traffic if there is room to handle some message loss. These are only a few examples of the things that can and should be tuned when preparing for a production deployment.

Once the broker is configured and tuned to be running how you want, you still have to set up monitoring to make sure it stays that way. This can be done with log scraping or extra tools provided by the platforms where Apache Kafka is deployed. There are even some tools that can be purchased to help with monitoring. But many companies opt to just integrate Kafka into the systems they already have for monitoring and alerting. Integration or setting up a new tool takes time and then has to be watched regularly.

Eventually, if your business succeeds and grows, so too will your Kafka clusters and the amount of data that they store. Although a standard installation of Kafka can handle a large amount of traffic, eventually you may need to add another broker to the cluster. This requires rebalancing the consumers as well or there will be no benefit to the added hardware. If the backing storage starts to fill up, it can be increased but requires restarting and replicating large amounts of data over the brokers. Also, because Kafka is

actively being developed and improved, there will be security fixes and new updates that should be applied. In order to avoid downtime, you can perform a rolling upgrade of the brokers, but you have to be aware of what versions the clients support and wait for all of the data to sync back up before moving to the next broker. One of the newest updates even changes the required dependencies for configuration management and could require a complete rebuild of your cluster.

Finally, in order to secure the traffic to a broker, you need to set up certificates and handle distributing them to clients. You also have to set up an access-control list (ACL) for topics based on users that can either be defined in Kafka or managed externally. Most companies will already have an identity management tool that then has to be integrated with Kafka in order to give the correct access to topics. This is time-consuming to configure and maintain but must be done to protect the valuable data in the brokers.

Apache Kafka Made Easier with Kubernetes

All of the challenges discussed in the previous section can be accomplished more easily with the use of a modern application platform. Tuning becomes simpler because everything can be managed within a single configuration file. Monitoring is often built into the platform. Scaling and upgrading can be automated with tools like Kubernetes Operators. Certificates can be stored or

distributed to clients on the platform without manual steps. Users can be configured with the same DevOps approach used by the infrastructure or integrated with the identity management tools running on the platform already. And if the same platform is used for the Apache Kafka brokers and the applications that you are modernizing, they can benefit from being colocated.

Choosing the right application platform can help organizations and IT development teams to stay agile and flexible while modernizing applications and deploying application services like Kafka. The right application platform is able to:

- Abstract the complexity of accessing and managing infrastructure
- Provide developers with freedom and flexibility when choosing their preferred development tools and technologies
- Deliver a consistent developer experience in the data center and in private and public clouds
- Provide the tools needed for automating application's build, test, and delivery
- Offer a set of messaging and integration technologies that work together seamlessly
- Provide a place for managing existing legacy and new microservice applications

The right application platform supports IT development teams in reducing the

time it takes to deliver an application and to deploy the tools needed to support the applications. These tools include application services such as integration, streaming, messaging, identity management, application monitoring, and data services. Container orchestration platforms also allow consolidation of hardware, simpler horizontal scaling, built-in high availability, and a way to ensure consistency between environments. All of this leads to time savings that allow development teams and IT organizations to spend more time on innovation.

Modern cloud orchestration, cloud computing, and cloud services platforms are designed to facilitate reliable, repeatable, predictable, and consistent deployment and provisioning of both applications and services. Apart from cloud platforms, automation tools like Ansible, Terraform, Tekton, and ArgoCD allow organizations to define the deployment of their cloud infrastructure, cloud services, and applications in a software-defined manner, enabling the adoption of methodologies like GitOps. There is a vast array of plugins available for these tools to deploy and configure cloud provider infrastructure (e.g., AWS, Azure, GCP), cloud services (e.g., databases, messaging services, monitoring systems), containers, and applications. A fully automated system can provision and deploy a complete application and its infrastructure in a matter of minutes, saving even more time to focus on the business problems that matter to an organization.

One thing that is very important for an Apache Kafka implementation in Kubernetes is keeping a map of dependencies between producers, Kafka

brokers, and consumer applications. When updating producer applications, discrepancies between topics are caused if processes like performance tests, compatibility tests, etc., are not automated and dependency maps are not in place. It is important to ensure you are deploying Kafka with automation in mind, if you want to maintain developer freedom and flexibility and are looking to keep a healthy pace for delivering applications.

The implementation of Apache Kafka standalone requires skilled IT professionals dedicated to configuring and maintaining the platform. Kafka's implementation on Kubernetes reduces the complexity of managing the platform thanks to its ability to efficiently optimize resources based on the streaming requirements (i.e., containers and clusters, based on the streaming requirements) and its ability to automate operational tasks in the form of operator and controller components. IT professionals can create and destroy Kafka environments for development or testing purposes as they see fit, and autoscaling can support those use cases where software and hardware requirements are variable.

The Strimzi project provides a way for running on Kubernetes and providing an automated approach for managing and maintaining complex implementations. Kafka implementations require a large amount of knowledge over a wide selection of services for managing the infrastructure and for keeping track not only of the performance of the platform but also of things like scalability, fault tolerance, performance, and storage. DevOps can address a few of these challenges with some of the functionality provided by

Kubernetes, but in some cases, adopting a managed approach can be more beneficial.

Cloud computing, and in particular managed cloud services, is another way to ensure IT development teams are focused on innovation and not operational tasks. Managing and administering container orchestration platforms and application platforms is a time-consuming task that requires having highly trained personnel dedicated 24/7 to keeping the lights on. When delegating these tasks to service providers, the IT developer teams gain the flexibility required to dedicate more time to innovation and less to management.

Managed application platforms provide self-serve access to development environments, allowing developers to get started quickly. Self-service access ensures developers won't need to wait for IT teams to provide them with a place to develop, test, and iterate. There is a lot of freedom when developers get to choose the best and preferred technology, framework, and environment to design an application. Productivity will increase if developers are allowed to focus on the application and the business logic of an application.

Managed Apache Kafka can provide and deliver the same benefits during application development as those covered for cloud computing and even more. Managed Kafka reduces the complexity of managing and maintaining a streaming platform since it takes care of configuration, monitoring, and scaling of the platform. It supports data distribution rebalancing and is tuned to provide a high-throughput, low-latency platform for optimal performance.

Finally, sizing for storage, retention, throughput, and latency are monitored for cost efficiency and better scalability.

Providing developers with access to an application platform that provides fully managed support of an Apache Kafka implementation allows them to move faster during application development. Developers don’t need to worry about tying up the integration between Kafka and the platform, and it also provides them with an infrastructure that scales when necessary and is flexible enough to maintain its existing legacy applications and new microservice applications.

Chapter 3. The Impact of Application Modernization and Apache Kafka Adoption on Business Agility

Previously, we discussed how a phased approach to modernization provides organizations with the flexibility required to modernize their application landscape at their preferred pace. Apache Kafka is a technology that enables the execution of such a phased approach. Streaming pipelines can be put in place to serve as the communication channel between legacy monoliths and new microservice applications. Kafka allows monolithic applications to expose data and events in standard formats, making it available for consumption by other systems. This establishes a common and immutable data pipeline between applications and services, which can help decrease data conflicts, reduce complex dependencies, and allow for optimization and scalability.

When focusing on applications that need to be refactored or replatformed, there is always some amount of application development that needs to take place. In this chapter, we will discuss how introducing Apache Kafka to your architecture can reduce the effort and increase the effect of your modernization efforts.

Benefits of Using Apache Kafka during

Application Development

The number of use cases supported with the Apache Kafka ecosystem is large and diverse, making it a preferred messaging solution by many organizations. Kafka lets organizations view and analyze a business in real time and react quickly to continuously changing market situations.

Apache Kafka with its product ecosystem has been recognized for supporting complex architectural patterns, such as real-time streams processing, event-driven architectures, and AI/ML streaming analytics deployments.

Real-Time Streams Processing

Real-time streams processing refers to the architectural paradigm where data is ingested and processed in near real time. The paradigm is often used to support modernization efforts where batch processing is still in use and real-time support is required.

Batch data processing is a standard method used by many organizations for collecting and storing high volumes of data over a period of time before processing. Delivering batch data requires an infrastructure that allows moving and dumping large amounts of data at once. For many organizations, batch processing will happen once a day or once a week, depending on the use case. The method has been very successful for scenarios where time was not an issue, including, for example, bank reports, billing, order fulfillment,

and so on.

Implementing a real-time streams processing architectural pattern can bring new technical challenges, including scalability, fault tolerance, and message delivery guarantees, as well as procedural challenges such as data sequence, consistency, quality, and correctness. Modern technologies have evolved to support both new and existing data stream processing requirements. Apache Kafka is one of the technologies that supports modernization of batch data processing architectures while providing a way to handle many of the challenges introduced by moving to a real-time architecture. Its publish/subscribe mechanism allows for low overhead, high-rate message delivery, and higher scalability.

Apache Kafka can support the analysis of continuous streams of data, eliminating the need for aggregating batches of historical data. This could lead to a cost-saving scenario since any data that is not needed can be deleted rather than stored in large data warehouses.

The Apache Kafka ecosystem provides various components required to implement an architecture that promotes real-time streams processing.

- Prebuilt connectors provide a standardized way to connect a variety of data sources to Kafka, effectively unlocking their data by streaming it to topics and making it available to the vast array of Kafka consumers.
- Schema registry offers a service for designing and registering schemas that

improve data quality, and it also enables data governance when sending data in serialized format.
- The Kafka Streams framework provides application libraries that allow developers to write stream processing applications that consume, analyze, and process continuous streams of data from multiple sources.

As mentioned earlier, modernizing an existing application or architecture cannot happen overnight. Streams processing and data platforms allow organizations to unlock data from disparate sources; make data available in a standardized, distributed, and high-throughput manner; and enable organizations to meet their real-time processing requirements. This gives organizations the flexibility and agility needed to gradually transform their application architectures from batch-oriented to stream processing–oriented, unlocking the potential of real-time data processing.

Event-Driven Architecture

Event-driven architecture (EDA) is a software architectural pattern for application design based on streaming of real-time events or notifications. An event-driven system focuses on capturing, processing, and persisting events that travel unidirectionally from source to destination. This paradigm has re-emerged as an ideal solution for modern use cases that involve big-data movement, real-time responses, or the need for better integration scalability, flexibility, and durability.

Event-driven communication is often identified as an alternative approach to traditional client-server messaging. Traditional messaging architectures are characterized by offering a centralized approach with intelligent brokers that make routing and persistence decisions. This broker-centric approach focuses on low-volume messaging, allowing for slower persistent storage until delivery, unique support per message, and guaranteed message delivery.

Traditional messaging communication technologies are well-suited for complex IT infrastructures that require using granular messaging configurations to send messages from one system to many. They are also especially useful when sending information that is highly sensitive. Modern use cases that require a real-time approach, faster processing, larger scalability, high availability, and improved fault tolerance might benefit from an event-driven approach rather than traditional messaging patterns.

Before diving into additional benefits of EDA, let’s define what an event is. An event is any significant change in the state of a system within a business process. The source of an event can be from internal or external inputs. Events are often time-sensitive or mission-critical for the business organization, and they require dedicated attention. Examples of events are online orders, inventory checks, billing generation, invoice submissions, changes in temperature, the hiring of a new employee, and so forth. Events are usually characterized by being immutable, persisted, and always consumable.

The key piece of an EDA is the event broker. Event brokers are middleware components that receive published events and deliver them to subscribed consumers. Publishers are not required to know who will collect and process the event. Only consumers that care about the event data will ingest it. Apache Kafka is designed to act as an event broker.

Apache Kafka’s broker configuration allows for loosely coupled, asynchronous communication. Consumer applications and the event producers have no system availability dependencies. This means that producers do not expect an immediate response from the application consumer, and multiple received events can be logged or queued until the consumer becomes available to process it. This ability allows organizations to move large volumes of messaging data at a very high rate, moving closer to real-time message delivery.

Another benefit of EDA is parallel processing and horizontal scalability. As explained in the previous chapter, each broker in the Apache Kafka architecture can contain multiple topics, and each topic can be broken into multiple partitions. This configuration allows consumers to access topics and multiple partitions in parallel, which improves scalability. Horizontally scaling requires you to add more brokers to an existing Kafka cluster. The cluster will require more nodes to balance out the load and ultimately allow the cluster to serve more requests.

The capacity to add more brokers and topics makes EDA a highly

composable architecture. Developers can split apart and recombine new microservices to deliver new functionality, all from the same events.

AI/ML Streaming Analytics

Streaming analytics deployments are based on using a streams processing framework or other analytics techniques like artificial intelligence and machine learning to predict behaviors, identify outliers, or offer recommendations.

Delivering AI-driven applications requires designing and creating ML algorithms that can support prediction and detection of events. ML algorithms are trained with historical data and are then deployed to be used with real-time data. Training the models requires ingesting large amounts of data, which can be complex and time-consuming. Streaming analytics architectures can support AI/ML applications to improve their incremental learning by providing streams of data.

One of the challenges of AI/ML modeling is that it requires a cyclic approach, which means collecting historical data for model training and ingesting real-time data for actionable insights. AI/ML modeling requires high computational power, tons of storage, and a development environment where data scientists can run and train models alongside developers who can design intelligent applications.

Apache Kafka supports data ingestion and streaming during AI/ML modeling, and once models are deployed, the same infrastructure can be used for prediction and detection of events. Figure 3-1 shows how streams processing technologies could support AI/ML modeling and inference.

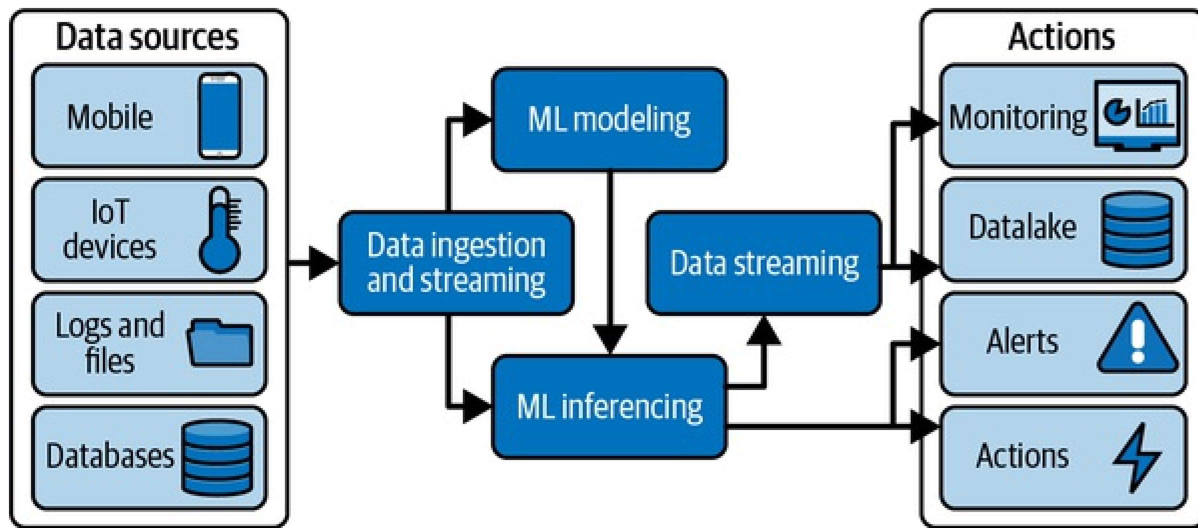


Figure 3-1. AI/ML modeling and inference

Apache Kafka and its components are able to connect to any data source and stream vast amounts of data in near real time. Kafka allows the storage of data for consumers to retrieve when necessary. Historical stored data can be valuable when training AI/ML models.

The Apache Kafka ecosystem is a logical fit with AI/ML modeling and inference because it provides a simple way for connecting to multiple data sources without the need for creating complex integrations. The combination of the Kafka broker and Kafka Streams allows for supporting data transformation and processing before data is used for model training.

One thing to keep in mind is that choosing an AI/ML platform that can be highly integrated with a modern application platform will make the complete process easier for every team. A modern application platform will support AI/ML operations teams to provide environments for open collaboration between developers and data scientists when cleaning, ingesting, and using data. Modern application platforms also support IT operations teams when delivering AI/ML models as intelligent applications. And finally, having the data in the brokers closer to the running models will improve performance during modeling and serving.

How Apache Kafka Can Support Business Outcomes

Apache Kafka provides support for many modern use cases in today's organizations, providing a variety of benefits from the business-level perspective. When looking at IT scenarios, there are two real-world examples we would like to highlight to show some of the ways Kafka can benefit an organization during a modernization effort.

Connecting Distributed Systems for a Better User Experience

A large bank in the United States was already in the process of modernizing its applications to keep up with its competition when it started to feel the

growing pains that come with it. The bank had split its processes into distributed systems that allowed for more flexibility and scalability within a given line of business, but then the integration between the systems began to drift apart. The bank had settled on traditional messaging to connect different applications but struggled to define the contracts with which the applications would communicate. Each team had its own definition of what a customer was, and each team used the term "payment" to mean something different. Communication then required a large amount of transformation between each team, and no two interactions were the same. On top of all the work required to communicate between the disparate systems, each team was still storing its own version of information in a database for only its process. These sources of truth were often out of sync and slow to share information with each other, which made the customer experience terrible. A customer would call in to report suspected fraud and make it through a game of 20 questions with an automated system, only to be passed to a person that had no record of what information was captured by the automated system. After the call was done, if the customer logged in to the website, there was no record of their report or the action being taken on their behalf. The process broke down even further as real paperwork got involved and batch processes that required sending or receiving mail had no feedback into the system until it was done. The customer experience was terrible, and fixing the problem was taking too much time, while customers were leaving. These problems could have been solved using the bank's traditional messaging broker with more effort, but a new architecture was eventually proposed that brought in the Apache Kafka

ecosystem to help.

Kafka introduced reliability and fault tolerance to the bank’s messaging architecture and improved system availability for the applications. Having a reliable and centralized event broker created a single source of truth from which every channel could communicate. A unified definition of a customer and types of payments were created and stored in topics that could be accessed by any process to get relevant real-time information. This effort required more application development, but because the bank had already started to modernize, it was well-prepared to quickly adapt by using a modern platform. It also helped that the different teams no longer had to interact with a seemingly infinite number of other processes and only had to consider adding their data to the agreed-upon stream. This soon led to a system that stayed synced in real time and even started to incorporate many of the bank’s batch processes into the event bus.

The customer experience dramatically improved and even led to new innovations. Because all of the information about a customer was available as events, the bank was able to invest into AI to help predict what a customer would need help with and save them time no matter what channel they were connecting on. When a customer logged in to the website, they could be asked if they wanted to finish completing a fraud claim. If a customer called in to make a payment, they could be transferred to the right department without having to fight a phone menu.

Apache Kafka was a great fit for the bank's modernization efforts, as it worked well in the bank's distributed system, enabled an event-driven architecture, and gave the bank a real-time streaming platform that was reliable enough to use as a source of truth. The bank was already using a container orchestration platform, so deploying the brokers right next to the bank's applications was swift and simple. Most of the work was in rewriting each team's consumers to be aware of the topics and offsets. This effort was well worth it as the bank achieved its goal in an amount of time that was impressive for a bank.

Saving Money by Moving to an Event-Based Architecture

A commercial services company had recently moved all of its tracking to a new customer relationship management (CRM) system, and had a large number of old applications that were hitting the CRM API at a high rate for similar information, which was costing the company large sums of money. On top of that, the apps were old enough that they were no longer supported and needed to be upgraded to a newer runtime. In order to lower the number of API calls to the CRM, a single consumer was set up to read all of the customer data and write it into a central database. The company then set up CDC to take the database events and put them into Apache Kafka topics to be consumed by all of the other applications. This seemed like an elegant solution to the company's problem, but it was not without issues.

First, the Kafka cluster took months to set up. The team responsible for managing the infrastructure was not familiar with Kafka and struggled to get it provisioned. Second, all of the applications were deployed manually by the infrastructure team, and there were hundreds that had to be updated. Third, the data coming out of the database was huge, and not all of it was needed. Unfortunately, to save space, the team responsible just started taking out fields it didn't think were needed without agreeing upon a schema. This meant that almost every day the data structure changed: items were removed for being too large or added back because some other team complained about needing the data, only for them to be removed again when the team realized some of the data was duplicated in other fields. Finally, the new applications were functioning slower than their older counterparts, despite using a supposedly high-performance messaging broker and a newer runtime.

In order to solve these problems, the first change that was made was to start using a container management platform for the applications and infrastructure. This allowed for the database and Kafka brokers to be installed with Kubernetes Operators, which simplified the configuration into YAML files that could be version controlled in a GitOps pipeline. This gave everyone visibility into how the infrastructure was configured and allowed for controlled changes in a timely manner. Developers that were more familiar with Kafka were able to make suggestions to the configuration as code without needing access to the secured environment running the infrastructure. This, coupled with the fact that the applications were running closer to the brokers, instantly improved the performance back to reasonable

levels. Further tuning of the commit intervals on the consumers eventually yielded the large performance improvement that the company was looking for.

The same GitOps approach used for the infrastructure was adopted for the applications, allowing them to be deployed and updated en masse. Despite the new ability to change rapidly, the developers were still not happy having to constantly update their schemas to get the data from Kafka. They adopted a schema registry to automatically deserialize the newest format of the data and eventually started publishing smaller pieces of the database capture to different topics based on what type of event it was. This still made all of the necessary data available to everyone without overloading a single topic or consumer.

In the end, the company saved money by reducing the number of calls to its CRM system, lowering the amount of extraneous storage it had, and removing the need to pay for extended support on old software. The company achieved all of this with minor changes to the actual code base of its applications by relying on improvements to the infrastructure to aid its modernization. These improvements were not without their own issues, but by adopting GitOps practices and some much-needed tuning, the company was able to save money and increase its performance.

Conclusion

Modernizing applications can look very different based on your goals and how you decide to implement them. Often those goals can be accomplished more quickly with the help of modern messaging technologies such as Apache Kafka. The Kafka ecosystem works well in the cloud or as part of any distributed system. It comes with an ecosystem of tools to lower the amount of effort required to integrate with older applications. It scales well to handle large amounts of data as more and more applications start to use its topics. Kafka can secure communication and replicate messages to prevent data loss. Finally, it makes it easier to implement loose coupling in modern architectures.

Decreasing coupling between services and components in a system allows them to evolve more freely and easily, enabling faster time to value when developing complex software systems. The promotion of loose coupling also promotes the use of well-defined APIs and data types, discoverable endpoints, and an asynchronous, event-driven communication style. This, in turn, enables teams to more easily discover available functionality and data to implement new features, which enables new use cases and brings value to customers and organizations.

Delivering a better customer experience, improving the user experience quickly, and providing support across hybrid environments prepare

organizations for responding and reacting to quickly changing market conditions. In today’s digital world, responding fast to market changes to impact business outcomes is a must. Application modernization is not easy, but technologies like Kubernetes and Apache Kafka are designed to support the challenges faced by organizations when scaling their architectures and applications for such goals.

About the Authors

Jennifer Vargas is a marketer—with previous experience in consulting and sales—who enjoys solving business and technical challenges that seem disconnected at first. Her areas of expertise are AI/ML, IoT, Integration, and Mobile Solutions.

Richard Stroop is an integration domain specialist for Red Hat, Inc., where he has worked for ten years herding Apache Camels alongside many other technologies. He received his master's degree in computer engineering from Virginia Tech, where he began his focus on messaging systems. Throughout his career, he has been fascinated with how disparate systems and people communicate. He has helped large companies in almost every sector solve complex integration problems, all while keeping a positive and enthusiastic attitude. He leads the Integration Community of Practice to promote and adopt open source integration technologies in all organizations within Red Hat. He currently lives in the mountains of Virginia, where he enjoys playing hide-and-seek with his kids and numerous board games with his friends.