CONFLUENT

Event-Driven Microservices

Table of Contents

Introduction

Software Is (Still) Eating the World

Today, every organization needs a digital strategy to survive. Customers of all organizations are hungry for the best user experience. Gathering data, analyzing that data, and making relevant offers in real time are keys to driving a good customer user experience. Business stakeholders understand that within their organization, all of the data is at hand, but unfortunately, data is either siloed or stuck in a legacy system, which can be complex or risky to touch.

The Rise of Microservices

New technology emerges at an ever-quickening pace. To benefit from the new tools, system integrations need to be decoupled and polyglot. There is a lot of value in picking the right tool for the job. HTTP and REST APIs represent a standard and widely-adopted protocol. Every programming language has its own toolset for writing web services efficiently. Better operations, security, and automation tools have emerged. More than ever before, system admins and operators are comfortable with running different stacks in production, and many modern applications are thus composed of microservices implemented in many different ways.

In the past, enterprise architects had to be very restrictive about the tools used within an organization: tool choice was much more impactful to operations. Running a J2EE server has nothing to do with running an ESB cluster. Through standardization and automation, organizations are beginning to embrace a paradigm where each individual business unit may have a different technology stack. The price to pay is cheaper than it has ever been.

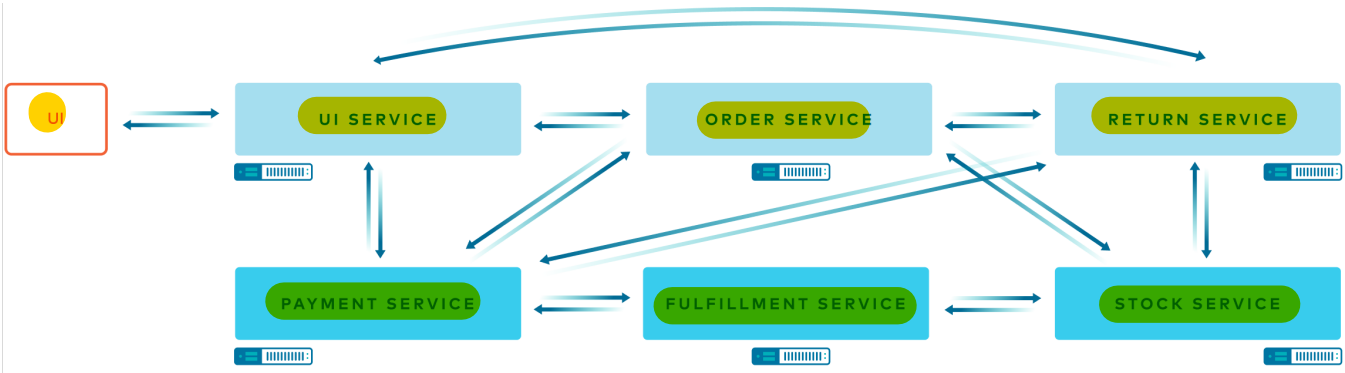
In the mid-2000s, companies including Amazon and Netflix adopted a holacratic organizational structure and spun off autonomous teams, which managed their services as products. Each team is free to choose the tools that meet its own requirements, as long as the team respected its contracts and Service Level Agreements (SLAs).

A Natural Evolution Towards Event-Driven Microservices

Challenges of Scaling Organizations

In these organized but chaotic environments, dependencies between services become organic and impossible to capture as they constantly evolve. For an enterprise that wasn't born in the cloud and employs decades' worth of legacy software, there is a danger of microservices increasing management complexity instead of organizational efficiency.

Such a system reaches scalability limits when microservices use synchronous communication. While Service A waits for Service B to finish its call to Service C, resources are blocking, waiting on every node to complete the chaining requests. This increases pressure on the system as a whole.



However, a fault or slowdown in any of these microservices will negatively impact the overall user experience.

Rethinking the Business As Events

Over the last few years, we have seen a revolution. Organizations must become real-time; to become real-time, they must be event-driven. At the heart of an event-driven approach is the event itself.

A sequence of related events represents a behavior. An item is added to and then removed from a shopping cart; a credit is applied to an account; a new user account is created; an error recurs every 24 hours; users always click through a website in a particular order. A sequence of related events is commonly called a "stream." Streams can come from an IoT device, a user visiting a website, changes to a database, or many other sources. When thinking of business complexity, we start with the event rather than the highly-coupled concept of a command: "an event X has occurred," rather than "command Y should be executed." Where we previously mined behaviors from log files in batch processing operations, we can now infer them from events in a more timely fashion. This thinking underpins event-driven streaming systems. We think of streams and events much like a database table and rows. Streams and events are the basic building blocks of a data platform. In an event-driven streaming architecture, we use a "key" to create a logical grouping of events as a stream, similar to what we do with a database table. Streams represent the core data model, and stream processors are

the connecting nodes that enable us to create flows, resulting in a streaming data topology. Stream processor patterns enable filtering, projections, joins, aggregations, materialized views, and other streaming functionality. Unlike in the table-and-rows model, events are front and center.

Is a Message Bus Enough for Event-Driven Microservices?

Traditional messaging offering Publisher and Subscriber semantics was designed with ACID (Atomicity, Consistency, Isolation, Durability) capabilities in mind. As defined by the CAP theorem, atomicity is a tradeoff that we need to make when seeking scalability and availability.

Traditional messaging middleware is not only non-distributed and therefore not scalable, but it also lacks persistent storage as a core characteristic. With a traditional messaging server, messages are persisted in queues until the server attempts a delivery, which is not guaranteed. This poses challenges when we need to add a new service and want it to consume historical events.

One workaround would be to expose an endpoint on the producer side, and use this endpoint to regenerate the desired events. But even with such a feature, we cannot guarantee that these events will be identical to the ones previously consumed. This is an anti-pattern: it would couple the two services together with an API, rather than with a streaming contract. An event-driven microservice must be able to replay historical events and messages that occurred prior to the existence of the service. A core feature of Apache Kafka® is that it persists messages and is able to natively serve past events to new consumers.

Bridging the Legacy With This New Paradigm

As organizations increase their usage of microservices, they develop improved architectures. But the elephant in the room remains. What about the data? How can we extract data from the monolith and serve it as events within a streaming platform, such as Confluent Platform? Kafka is flexible, provides a number of abstractions, and can act as an authoritative system of record. Kafka Connect provides an abstraction layer for linking data sources to Kafka in a plug-and-play fashion. Confluent, with its partner ecosystem, distributes and supports over 100 connectors, allowing you to move data both ways between Kafka topics and mainframes, SQL databases, MongoDB, Elasticsearch, Cassandra, and many others. A monolithic application dependent on these systems can use Kafka Connect to evolve into native event-driven microservices.

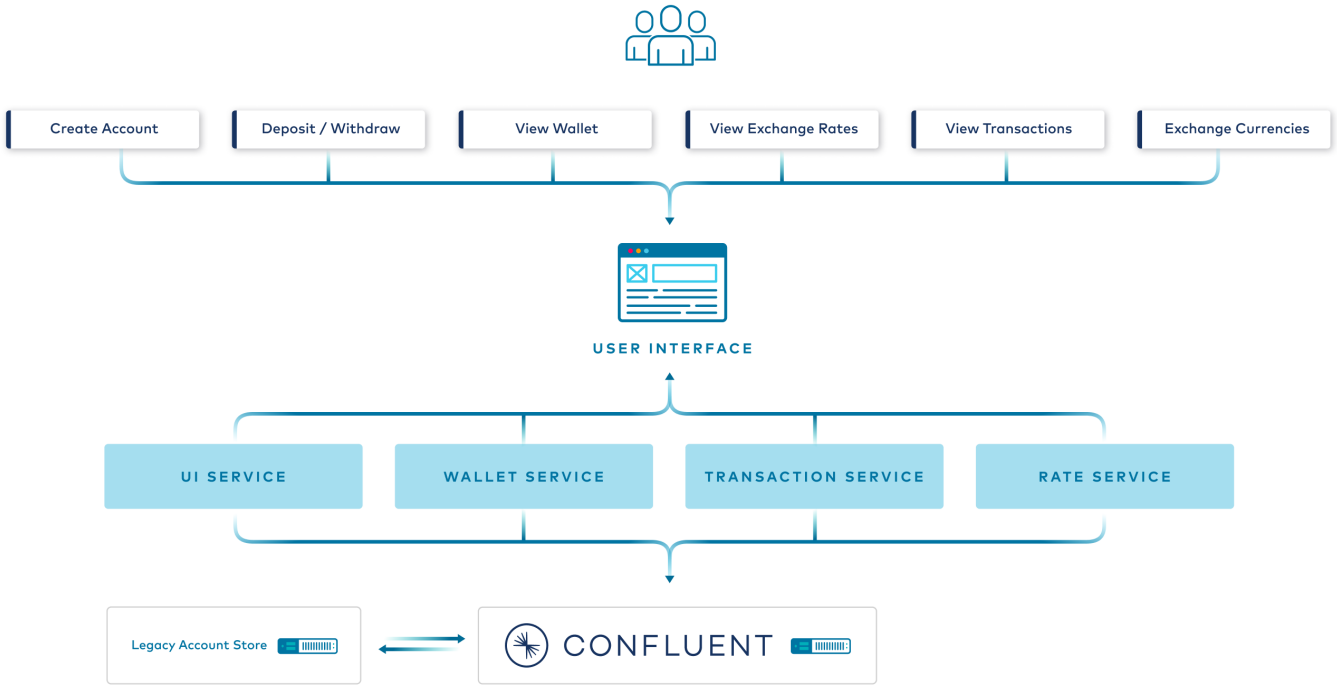
Designing for Event-Driven Microservices

General Consideration for Data Model

When designing an event-driven microservices architecture, it is tempting to look at the data first. But the data is irrelevant until we have identified the events and information required for our use case. The data being presented to a user only makes sense in a specific context. To build a generic and centralized view of the world would require coupling, which is already established as an antipattern. Deriving source events and processing them in real time is the key to building event-driven microservices.

Example: Building a Currency Exchange Platform

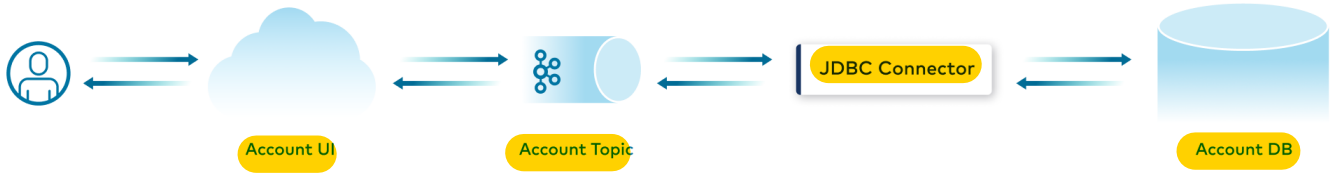
Let's take a functional use case, decompose the events and services, and work our way through each of the design considerations. A reference implementation of this platform is available online for readers to download and run. This platform allows a user to deposit funds in certain currencies and exchange them in real time, based on continuously updated exchange rates. While this application serves as a reference example, you can also walk step by step through the process of building a similar application in a tutorial.



Features of the Exchange Platform

Create an Account

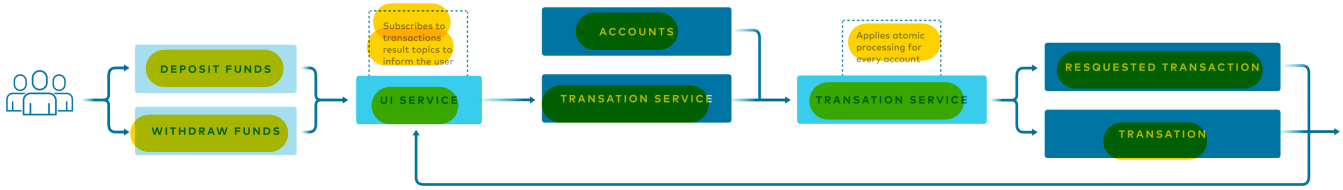
In the real world, systems are not typically written from scratch. Instead, they require integration, so our design simulates that. Our approach will be to expose account information as messages to a Kafka topic. In the reference implementation, we achieve this using the Confluent JDBC connector for Kafka Connect. Account creation is handled by the UI Service, which publishes requests to a sink topic. The JDBC connector synchronizes this sink topic with our legacy account datastore. The stream of user information can now be made available to any other downstream service that has an interest in user accounts, allowing us to build composable services.



Deposit and Withdraw Funds

Users can use the platform to deposit funds, using a randomly-generated credit card, to a simulated

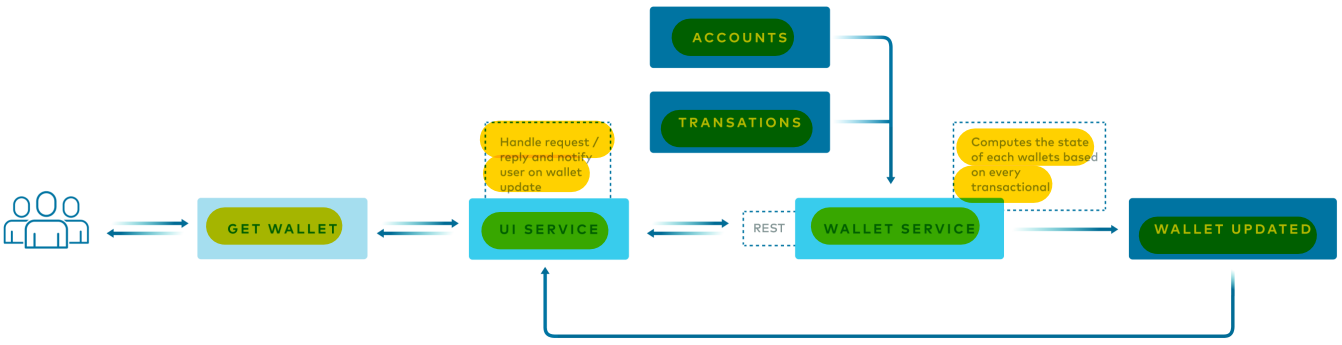
bank account number. Deposit and withdrawal operations each result in a transaction request being sent to Kafka. The Transaction Service will apply transactional semantics to validate transactions. In this scenario, we need to validate that the account has enough funds in a given currency before executing the operation. Using key partitioning and Exactly-Once Semantics, Kafka can guarantee that all the instructions for a distinct account will be processed transactionally and atomically.



When submitting an operation, we would naturally expect a request/reply semantic. The user only "wants" to submit a transaction. This simple "want" is an event in itself, and other similar "wants" may form other interesting events later. The "want" does not mean that the user has sufficient funds for the transaction. Because request processing is transactional and the outcome is uncertain, we shouldn't have to stop everything and wait for a response. What if the transaction result is an event itself? Consumers of the Transaction Service can subscribe to separate Success and Failed topics, which dictate the response to apply when a "success" or "failure" event occurs. As long as the backing transaction processor is transactional, users can flood the processor and it will continue to return coherent transaction responses.

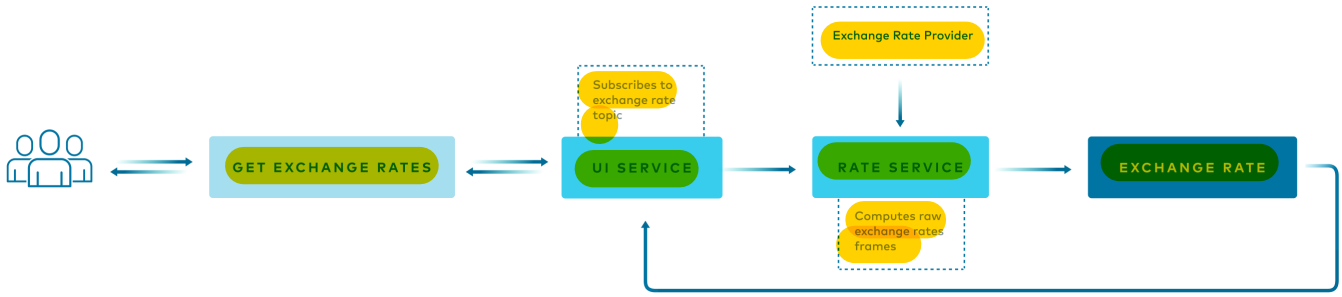
View Wallet

Keeping a complete snapshot of a user's portfolio of currencies is the job of a dedicated service, the Wallet Service. This service gives the user an up-to-date view of their complete wallet, so it needs to subscribe to all accounts and all successful transactions. The consumer wants to be informed when the wallet snapshot is updated; the best way to achieve that is to have the Wallet Service publish an event whenever it processes a new transaction. To stay up to date, the UI Service can use polling or lightweight notifications.



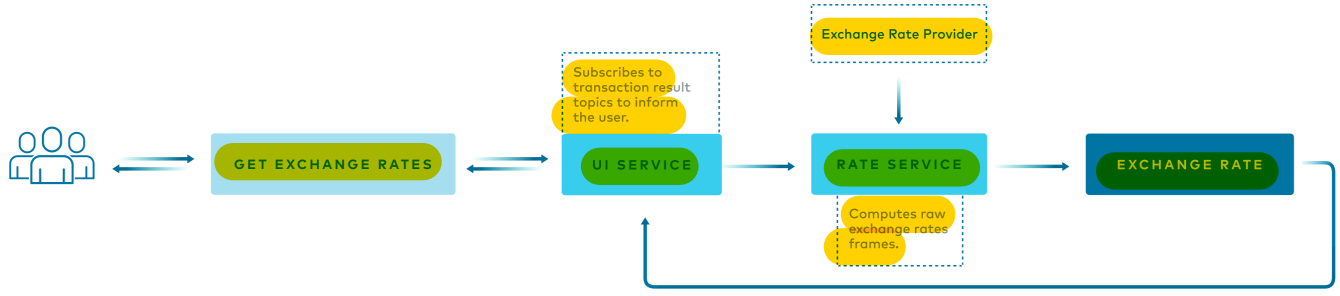
View Exchange Rates

The platform's main function is to perform currency exchanges. For this, we need a provider that can supply exchange rate quotes. The reference implementation provided with this project uses a public web socket endpoint from a cryptocurrency trading third party. The Rate Service will compute notifications from this provider and publish them to a named dedicated topic. The example implementation also exposes a web socket endpoint allowing the UI Service to display quotes to the user.

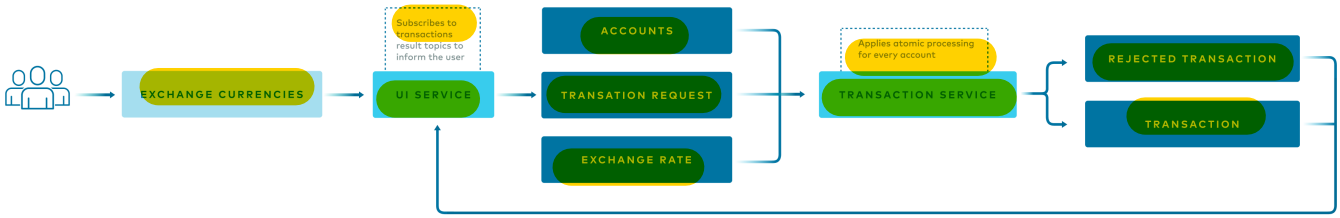


Exchange Currencies

The event flow for exchanging currencies is very similar to the event flows for depositing and withdrawing funds. Each translates into a transaction, and each requires the same atomic processing. The only difference when exchanging currencies is that we need to have the current exchange rates. We can ensure this by subscribing to the exchange rate topic, and then computing the amount while processing the transaction. The existing infrastructure will notify the user of the transaction result.



Summarizing Events and Their Respective Topic



Everything Is a Stream

Something that should stand out in this design is that each service has separate concerns; it knows nothing about the services upstream or downstream. We extract data from external providers' static data sources and translate that data into events. Business-oriented microservices then apply real-time event processing, joining and aggregating events. None of these microservices is concerned with extracting data from any specific datastore.

This allows us to define a clear separation of concerns and a pluggable architecture. For example, integration engineers can focus on the best implementation for extracting data from a mainframe, then publish the data to a Kafka topic for consumption by business developers working on business-related microservices. Allowing engineers to focus on the best implementations of their own individual microservices ensures an end-to-end well-architected system.

As we'll see in the next section, there are multiple ways to ingest data into Kafka. Your event-driven journey might lead you to swap one way for another based on various tradeoffs.

The Four Layers of Abstraction for Kafka Developers

Native Client

Kafka's clients are core to building any event-driven application. Layers of abstraction build upon simple client operations. In Apache Kafka, the communication between clients and servers is done using a simple, high-performance, language-agnostic TCP protocol. This protocol is versioned and maintains backward compatibility with older versions.

> Apache Kafka provides a Java client, which can also be used with other JVM languages, including Scala and Clojure

> Confluent provides Kafka integrations with C/C++, Python, Go, and .NET

> Additional community-supported clients work with different languages

Kafka Connect

Apache Kafka® also includes Kafka Connect, an application runtime and development framework for Java. Kafka Connect is supported by Confluent, and it removes a lot of the boilerplate code required by the Kafka Producer and Consumer APIs; it provides Sink and Source APIs that are specifically intended for moving data between Kafka and external datastores. Kafka Connect provides standardized connectivity to a variety of different data sources, and each connector hides the individual complexity of the data source from the developer or user. This enables not only out-of-the-box integration with specific data sources, but also decoupled integration between multiple sources and sinks, with Apache Kafka at the center.

Kafka Streams

Kafka Streams is a lightweight client library for stateful stream processing. It's useful for microservices where the input, output, and state data are stored in a Kafka cluster. Kafka Streams combines the simplicity of writing and deploying standard JVM-based client-side applications with the benefits of Kafka's inherent distributed, parallel nature. Here's a simple example of a Kafka Streams appliction which filters events based on a field in the event. A consumer reads events from **payments** and then writes, with a producer, to **fraudulent_payments** if any of the **payments** events has a **fraudProbability** higher than **0.8** (as determined earlier by a user-defined aggregate function, or UDAF).

```
object FraudFilteringApplication extends App {
  val config = new java.util.Properties
  config.put(StreamsConfig.APPLICATION_ID_CONFIG, "fraud-filtering-app")
  config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-
broker1:9092,kafka-broker2:9092")
  val builder: StreamsBuilder = new StreamsBuilder()
  val fraudulentPayments: KStream[String, Payment] = builder
    .stream[String, Payment]("payments-kafka-topic")
    .filter((_ ,payment) => payment.fraudProbability > 0.8)
  val streams: KafkaStreams = new KafkaStreams(builder.build(), config)
  streams.start()
}
```

ksqlDB

ksqlDB is a streaming SQL engine that enables real-time stream processing with Kafka. It provides an easy-to-use, powerful SQL interface, eliminating the need to write code in a programming language such as Java or Python.

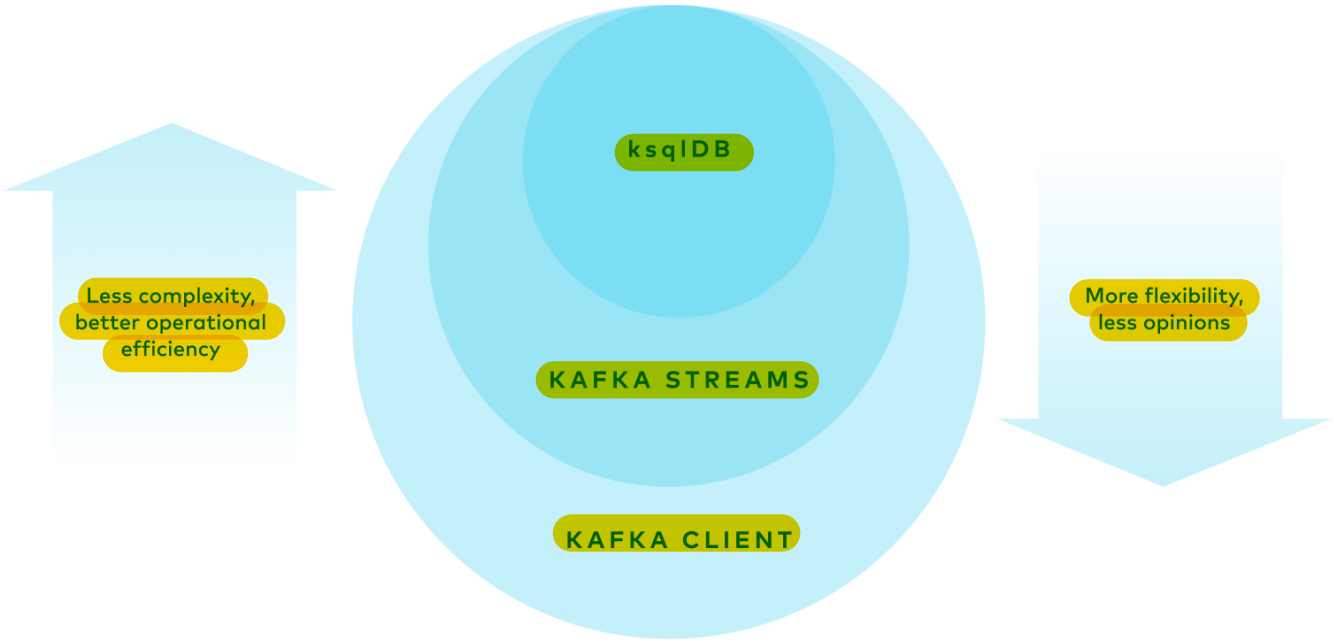
Scalable, elastic, and fault-tolerant, ksqlDB supports a wide range of streaming operations, including data filtering, transformations, aggregations, joins, windowing, and sessionization. ksqlDB is built on Kafka Streams, so a ksqlDB application communicates with a Kafka cluster just like any other Kafka Streams application. Additionally, ksqlDB supports direct integration with Kafka Connect; a single query can source a stream from a connector, process it, and pass it to one or more sink connectors.

The commands shown below create a Kafka Stream that is managed and scheduled by a ksqlDB cluster. This specific example implements the same use case as the Kafka Streams example.

```
CREATE STREAM fraudulent_payments AS
 SELECT * FROM payments
 WHERE fraudProbability > 0.8;
```

Native Client, Kafka Streams or ksqlDB? Picking the Right One

In the Apache Kafka ecosystem, there are many choices for stream processing. ksqlDB imposes the smallest burden for developing and running stream processing applications in production---no need to compile a binary and a container, to find a server or a PaaS and build deployment scripts. A simple ksqlDB command defines and deploy a streaming application. But the tradeoff is less flexibility with available operations.



Ensuring That Transactions are Atomic

Apache Kafka 0.11 introduced a new feature, Exactly-Once Semantics (EOS). These semantics build on idempotency and atomicity. Here's an explanation from the excellent blog post by Neha Narkhede:

> *"This is why the exactly-once guarantees provided by Kafka's Streams API are the strongest guarantees offered by any stream processing system so far. It offers end-to-end exactly-once guarantees for a stream processing application that extends from the data read from Kafka, any state materialized to Kafka by the Streams app, to the final output written back to Kafka. Stream processing systems that only rely on external data systems to materialize state support weaker guarantees for exactly-once stream processing. Even when they use Kafka as a source for stream processing and need to recover from a failure, they can only rewind their Kafka offset to re-consume and reprocess messages, but cannot rollback the associated state in an external system, leading to incorrect results when the state update is not idempotent."*

Guaranteed deliveries are only half of the missing ingredients for transaction handling. Kafka 0.11 also introduced a transaction API to allow control over offset commitment. Developers can define a scope of processing, and messages within the scope will be reprocessed if a failure occurs. With properly managed offset commitment, a process that reads and writes can guarantee that each event from its source topic will not be committed until processing has completed. The blog post Transaction in Apache Kafka goes deeper into the mechanics of transactions.

With these primitives, we can guarantee that our transaction processing, performed by a native Java

client, Kafka Streams, or KSQL, will be atomic and distributed by accounts. If a user floods the system with transaction requests, the requests will all be processed atomically, recovered if a network issue causes a processing failure, and computed as new events in Transaction and Rejected Transaction topics.

How Big Should a Microservice Be?

There are no straight answers to this question. If we define a microservice as a single independent system process, we would have to rely on multiple criteria to define how to group business functions. In theory, we could package all four of our microservices into a single artifact; from a Kafka perspective, they would still represent individual consumers and producers, each with a distinct consumer group. Or we could take the opposite approach, and package each microservice as a distinct binary.

As an illustration of the microservices concept, the example application takes this extreme approach, since we are implementing each individual Kafka client using different languages. When deciding the optimal size of a microservice in reality, we should consider several factors.

Team Structure

A general rule of thumb is that each business team working on an application should be able to work on a dedicated codebase, and have a distinct delivery artifact. Your first filter for defining the scope of a microservice is *1 team = 1 microservice*, at the very least. If you find that having multiple teams with different agendas and priorities, all delivering code as part of the same artifact, causes friction and slows the delivery velocity, you should consider adopting an event-driven architecture.

Rate of Delivery

Each individual business domain of your application will have a different rate of delivery. For our currency exchange platform, the Exchange Rate Service is a good example of a service that will likely have a slow delivery cadence. The Wallet Service, on the other hand, is closer to our core business domain, and it might be delivered weekly or monthly as we add new customer-driven features. Grouping these services into the same artifact means that your rolling upgrade will have an operational effect on both services. In most cases, this might be acceptable, but there might be a threshold where the difference leads you to separate them.

Risk Tolerance

Every financial institution will acknowledge that its transactional banking mainframe is probably its most critical component---imagine a bank being unable to handle debit and credit operations for its

customers. In the currency exchange platform, our Transaction Service is certainly critical. With a monolithic design, a deployment of Wallet Service revision could accidentally include deployment of a development version of the Transaction Service. That could result in a broken system or lead to bad transaction calculations being applied to the accounts. In this case, isolating the Transaction Service in its own dedicated deployment unit would make more sense.

Scalability

Our currency exchange platform allows the user to see a valuation of their Wallet in real time. This means that the Wallet Service will continuously recompute the total value based on the streaming exchange rates. Naturally, the Wallet Service will handle many more events than our Transaction Service, which is only triggered by user actions. An operational tool like Confluent Control Center can give you insights when the Wallet Service begins to lag on topic consumption. That would be a sign that the service needs to be scaled, whether vertically or horizontally. Separating this service ensures that you can scale it individually, without wasting unneeded resources on the other services.

Optimal Usage of Frameworks

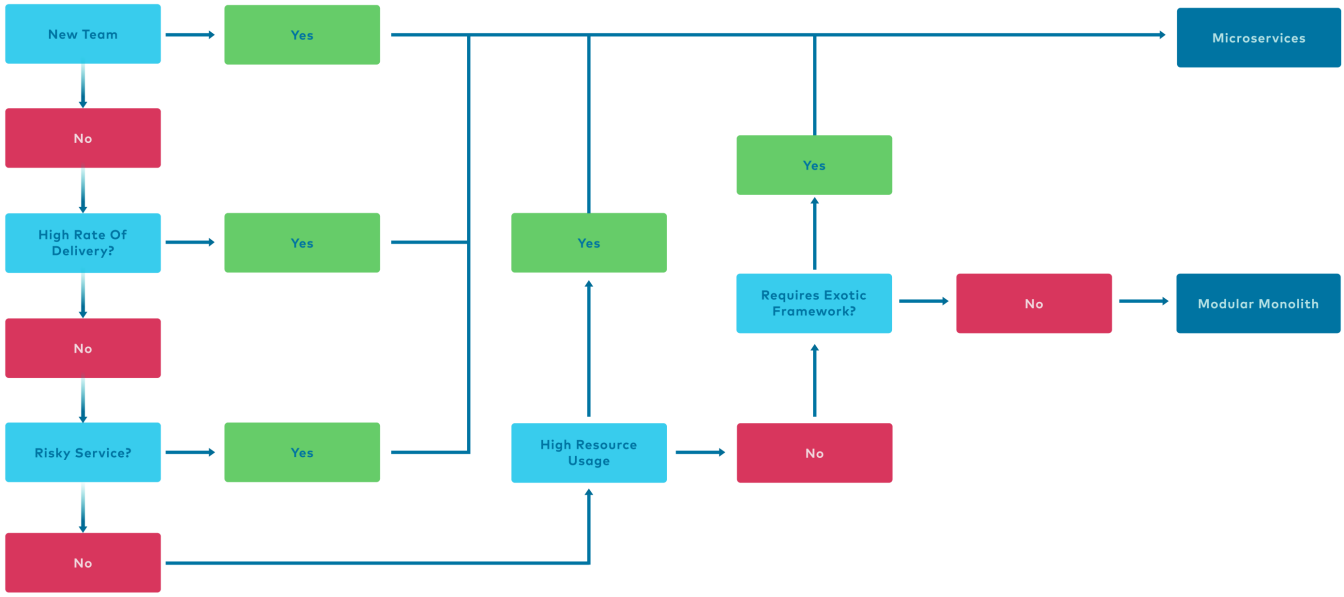
The choice of language and framework is a tradeoff discussion that will depend on your team's context. Some developers in the team might have a strong Java background. Similarly, you might have a requirement for event streams to be ingested to fuel ML training models written by data scientists in Python. Many different requirements can lead you to choose a specific framework or language, but no framework will meet every requirement. Adopting different stacks will allow you to achieve a natural separation of services.

When Nothing Justifies Separating Services

If you can't apply any of the above reasons to decouple your business logic into separate processes, then congratulations! This means that all of your business logic and services can be bundled into a single artifact. Your CI / CD pipeline and runbook will be simplified, and your colleagues from production operations will be thankful.

Microservices are not a panacea. If your application is well-designed and properly architected, you can always come back later, slice out a business module, and run it as a distinct microservice to realize those benefits. Because Kafka is your integration medium, the other modules of your application will be unaffected. Thank you, Modular Monolith!

The decision tree below walks through the process of deciding whether a microservices architecture is right for your needs.



Adopting Event-Driven Microservices

Confluent as Your Streaming Platform

Confluent delivers the industry-leading event streaming platform, on premises and in the cloud. An enterprise requires development frameworks, operational tooling, security mechanisms, and support to operate with agility and resilience.

What Successful Service Teams Have in Common

Autonomy

A streaming platform acts as a central nervous system for an entire organization. As integration engineers work on getting data into and out of Kafka, microservice teams can leverage all-inclusive data streams and focus on writing business logic that generates value. Microservices built on event streams free the development teams from dependency friction, allowing each team to ship code at its own optimal pace.

Going fast doesn't always mean you can break things. The next element highlights how breaking the

soft link between producers and consumers impacts business operations and affects your ability to scale.

Contracts

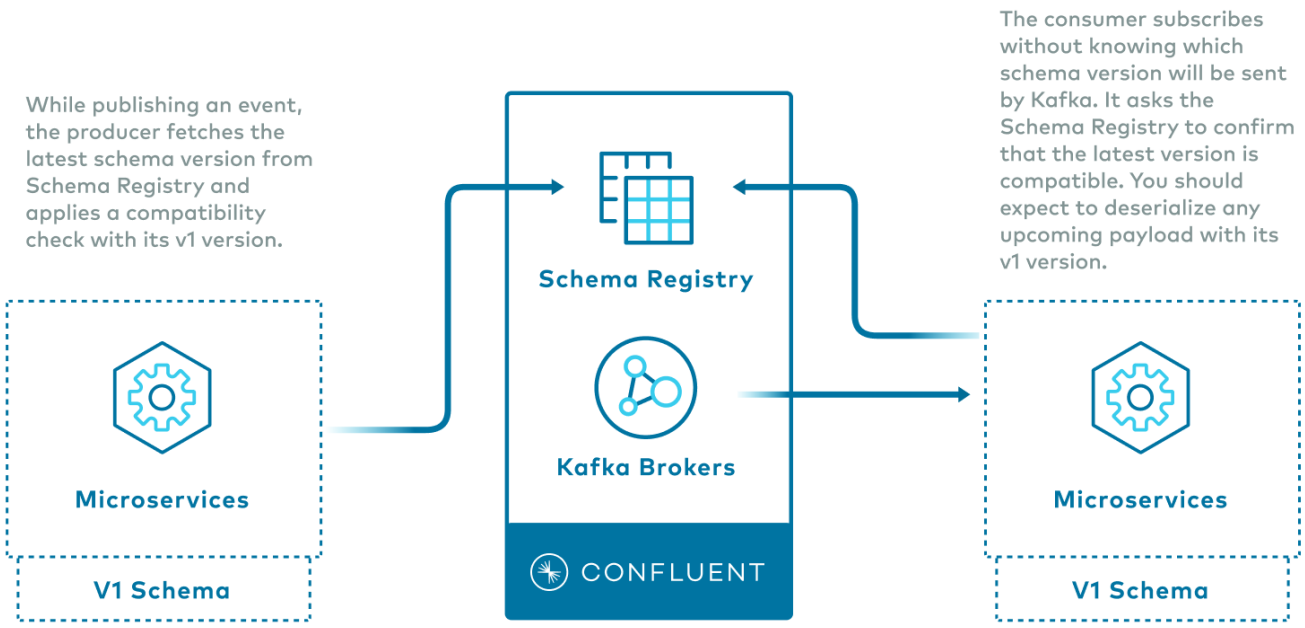
As a Kafka producer, your microservice has an obligation to deliver data in a specific format to its consumers. Confluent offers a Schema Registry, which uses Avro data serialization and schema evolution rules to require a producer to respect schema compatibility. The Schema Registry allows you to upgrade producers or consumers with new data formats, while preserving existing contract agreements until all communicating services have been upgraded. This makes the Confluent Schema Registry especially important over time; by design, microservices evolve independently of each other.

By default, schemas created in the Confluent Schema Registry are defined with the *BACKWARD* compatibility type. Let's go through the appropriate release workflow for keeping data compatible between different microservice versions. The Schema Registry documentation includes a dedicated section about compatibility types. The following summary focuses on the BACKWARD type.

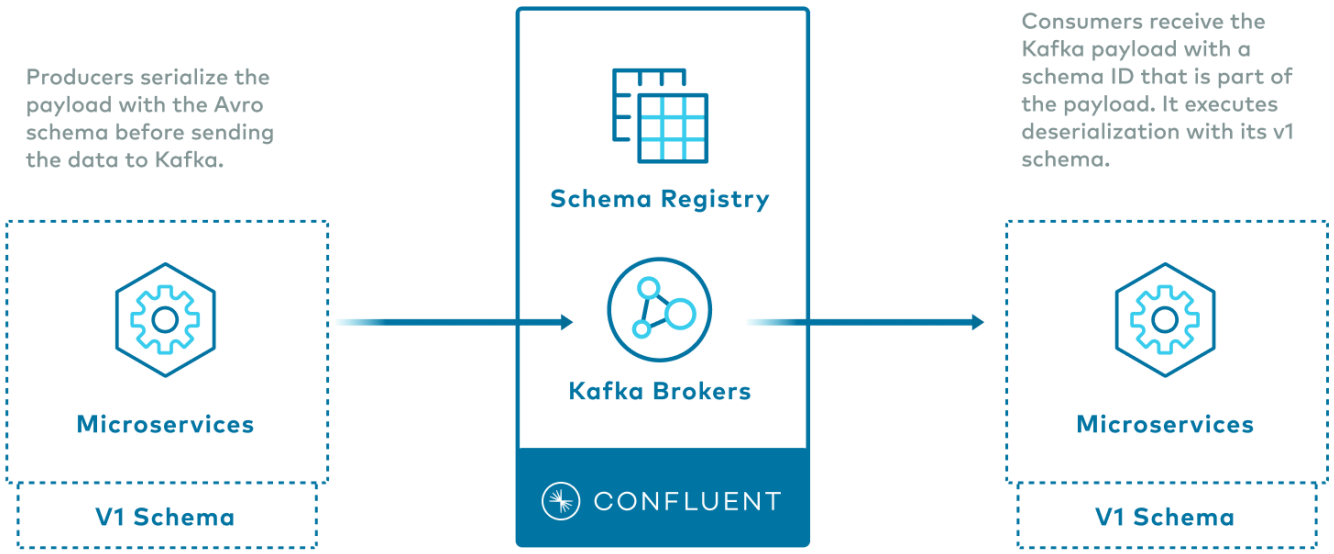
The Schemas and Their Versions

Version	Avro Definition
v1	\{"namespace": "example.avro", "type": "record", "name": "user", "fields": [\{"name": "name", "type": "string"}, \{"name": "favorite_number", "type": "int"}] }
v2	\{"namespace": "example.avro", "type": "record", "name": "user", "fields": [\{"name": "name", "type": "string"}, \{"name": "favorite_number", "type": "int"}, **\{"name": "favorite_color", "type": "string", "default": "green"}**] }

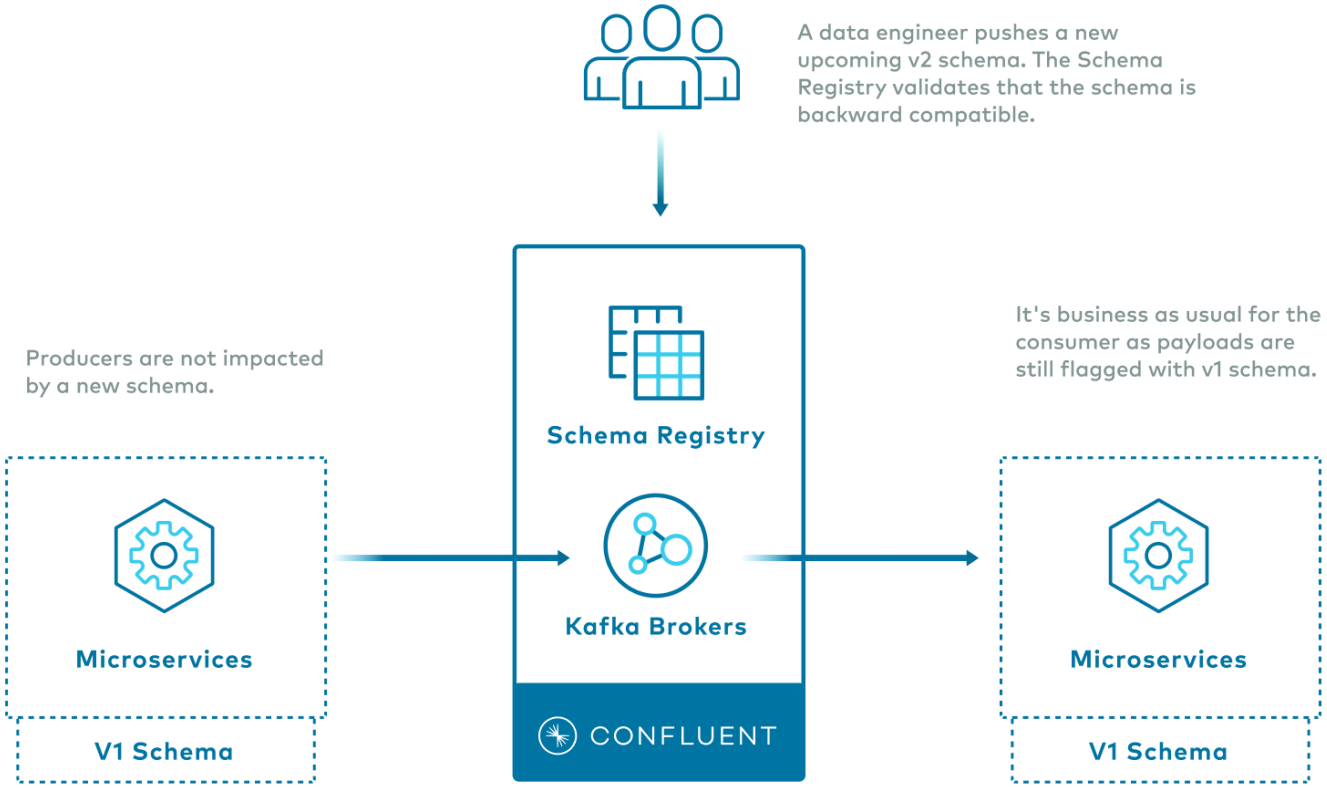
Running Microservices with a v1 Schema version



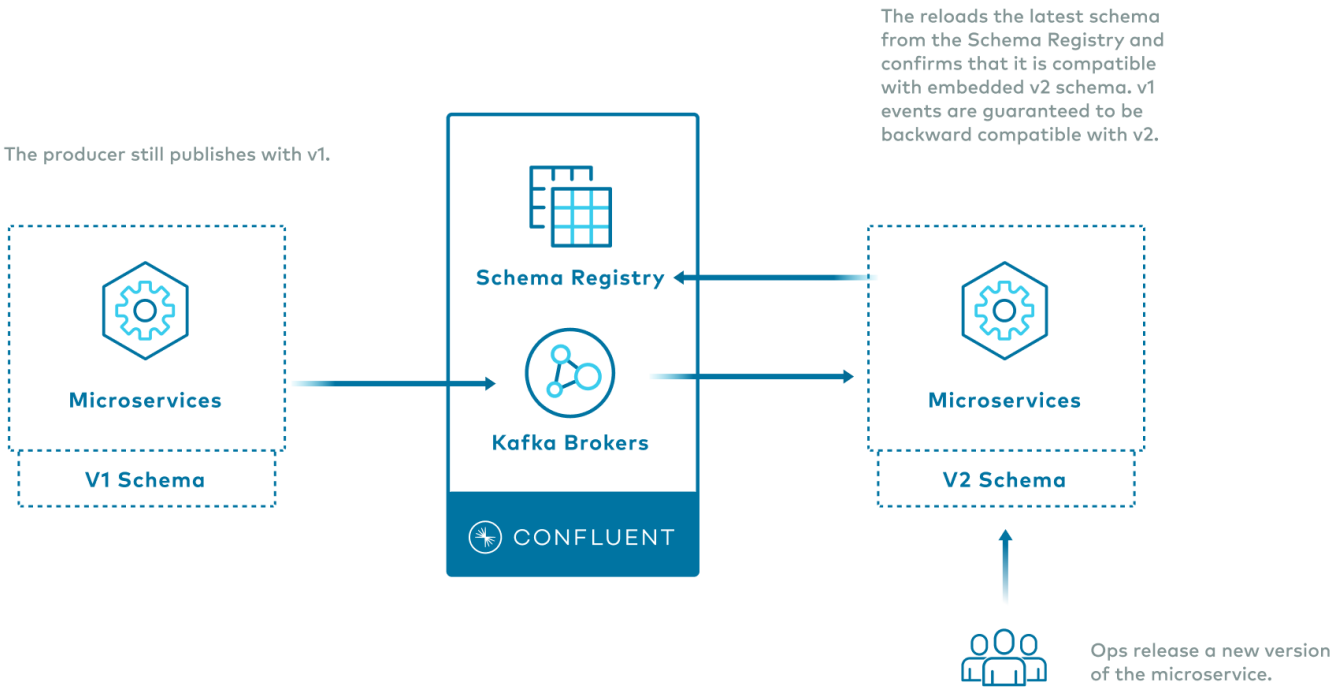
Producers and Consumers Communicate Using the v1 schema



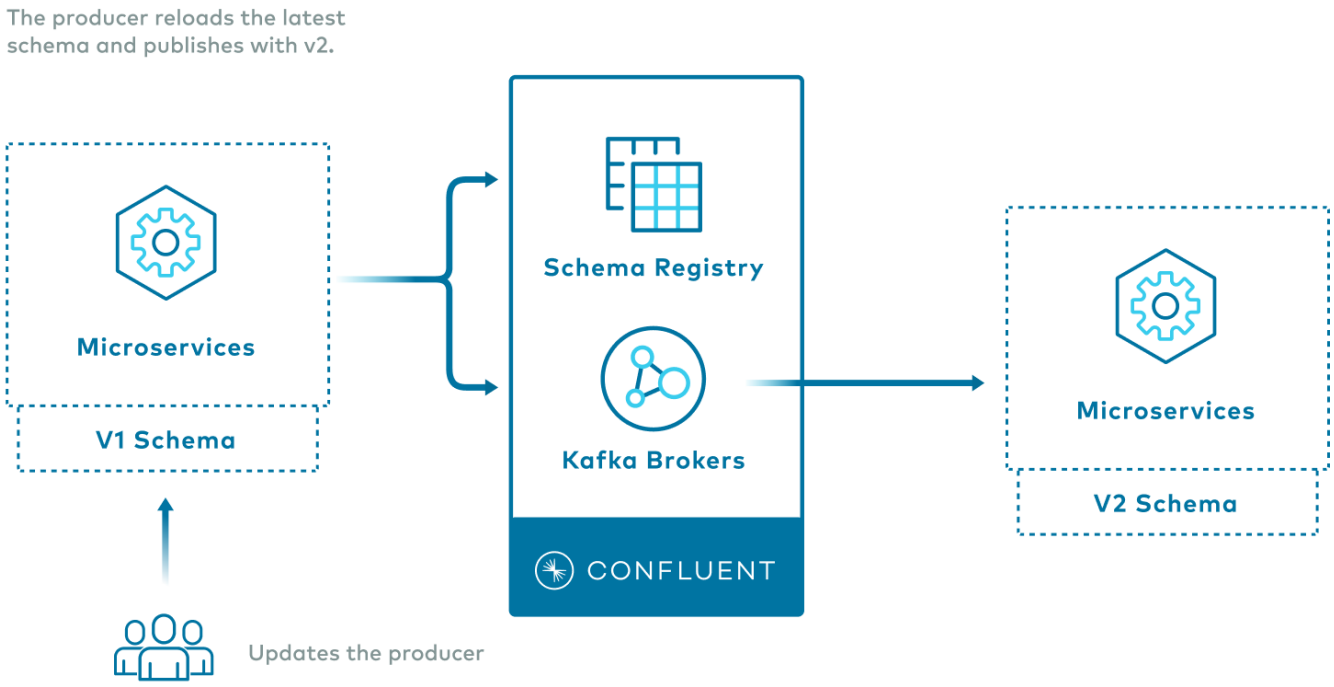
Data Engineers Publish a New v2 Schema in Schema Registry



Ops Releases a New Version of the Consumer Microservice

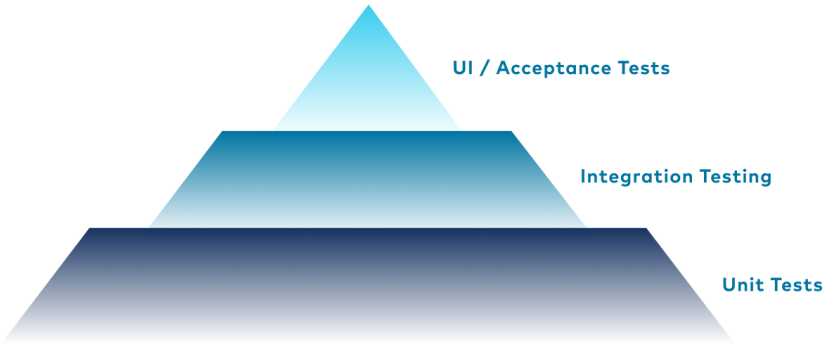


Ops Releases a New Version of the Producer Microservice



Testing

There are different schools of thought regarding naming and testing strategies, but most agree on the principles of the Test Pyramid. You'll want the majority of your tests to be isolated, execute quickly, and cover as much as possible of your code base. These unit tests are the first line of defense for validating that your business logic behaves as expected. The objective is to fail as fast as possible on regressions. As you climb the pyramid, you test more layers of the software, at the price of slower execution and more dependency requirements.



Continuous Integration

Continuous Integration is the phase where an automated toolchain tests your code, with all of its dependencies. These tests don't need to validate every possible aspect of your business functionality, but simply ensure connectivity and verify that behaviors work as expected. They are naturally slower to execute, as they are closer to end-to-end testing. These tests should be limited in number so that they don't take too much time.

As we saw earlier, with a traditional microservice architecture that doesn't leverage Kafka, your integration tests would require a stable version and pre-defined datasets for every microservice that communicates with your application. As the owner of the Wallet Service, for instance, you would depend on the Transaction Service team to provide you an environment where you can test your own new version. The same would apply for the Exchange Rate Service---your application also depends on that. What used to be centralized is now distributed.

This is where event-driven microservices, with Kafka and Confluent, shine. With contracts and using a streaming platform such as Kafka, the Wallet Service only needs a Kafka environment and its own personal datasource. Your test infrastructure can replay events from the Transaction topic, or mock them through new publications. This decouples the Wallet Service team entirely from the Transaction Service team and increases your autonomy as a team to deliver your software faster.

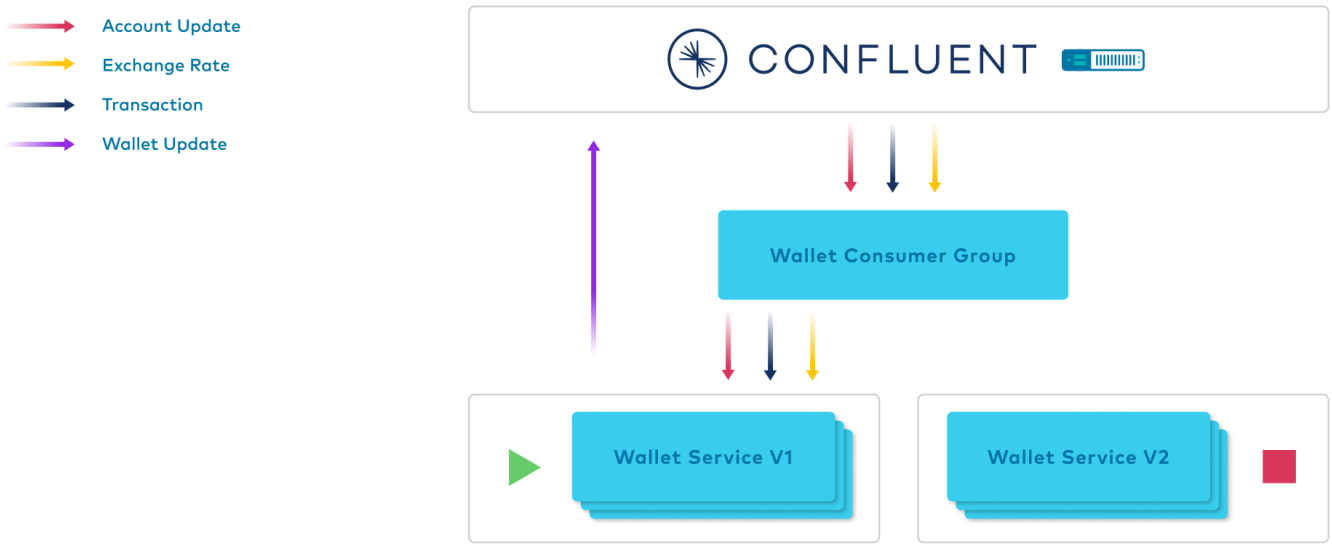
Continuous Delivery

All of the previous aspects come together in Continuous Delivery. By leveraging all of them, you can gain confidence in delivering changes to your system as they are committed, tested, and robust. Some organizations include fully automated acceptance tests as part of their CI / CD pipelines.

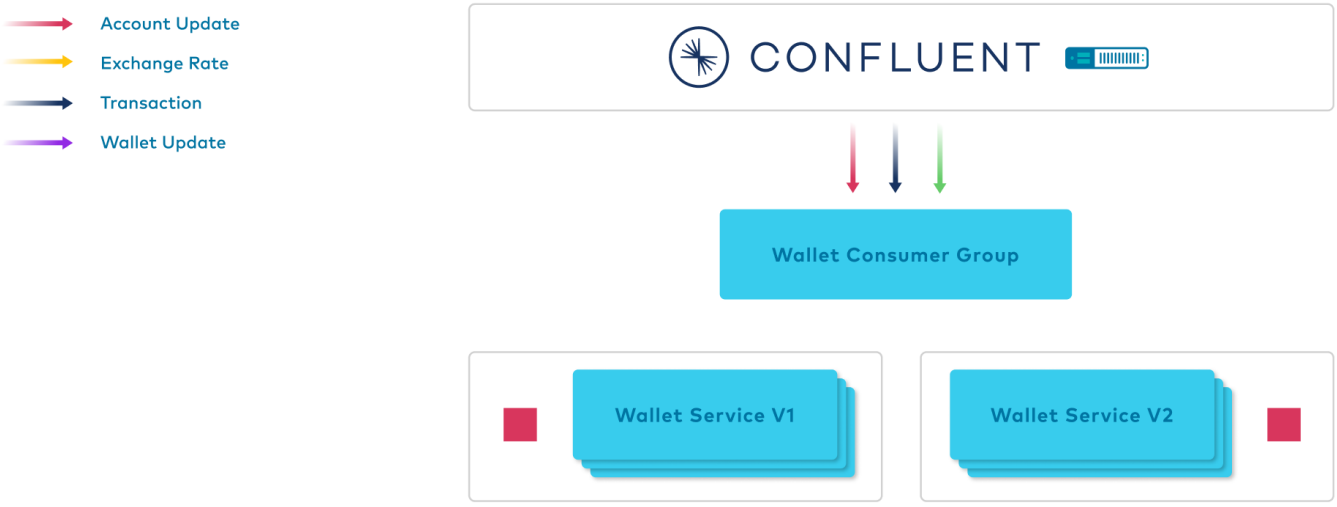
With an artifact validated and ready for production deployment, you next need to identify an upgrade strategy. Kafka enables different deployment models for different contexts.

While a REST-based architecture includes an HTTP load balancer to redirect traffic between an old version and a new version, Kafka provides the same abstraction mechanism through consumer groups. When subscribing to a topic, a consumer is linked to a consumer group, and Kafka tracks the offsets consumed for the group. You can stop an old version of your microservice and start the new version—both are part of the same consumer group. To roll back the Blue / Green deployment, you can stop the new version and restart the old version. This strategy involves a small amount of downtime but provides guarantees for how microservices will process events during the operation. With a fully event-driven architecture, this downtime will not have any effect other than a slight delay in downstream event processing. Event-driven systems are resilient by nature and handle back pressure from unavailable systems.

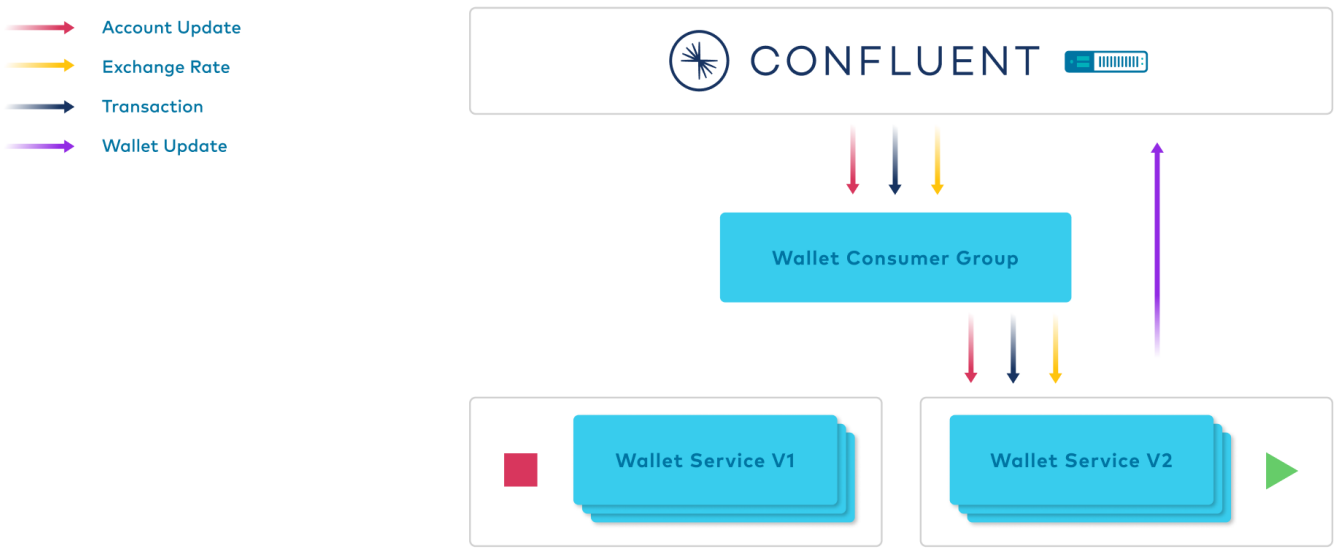
Deploying an Inactive New Version



Stopping the Existing Version to Ensure No concurrent Processing

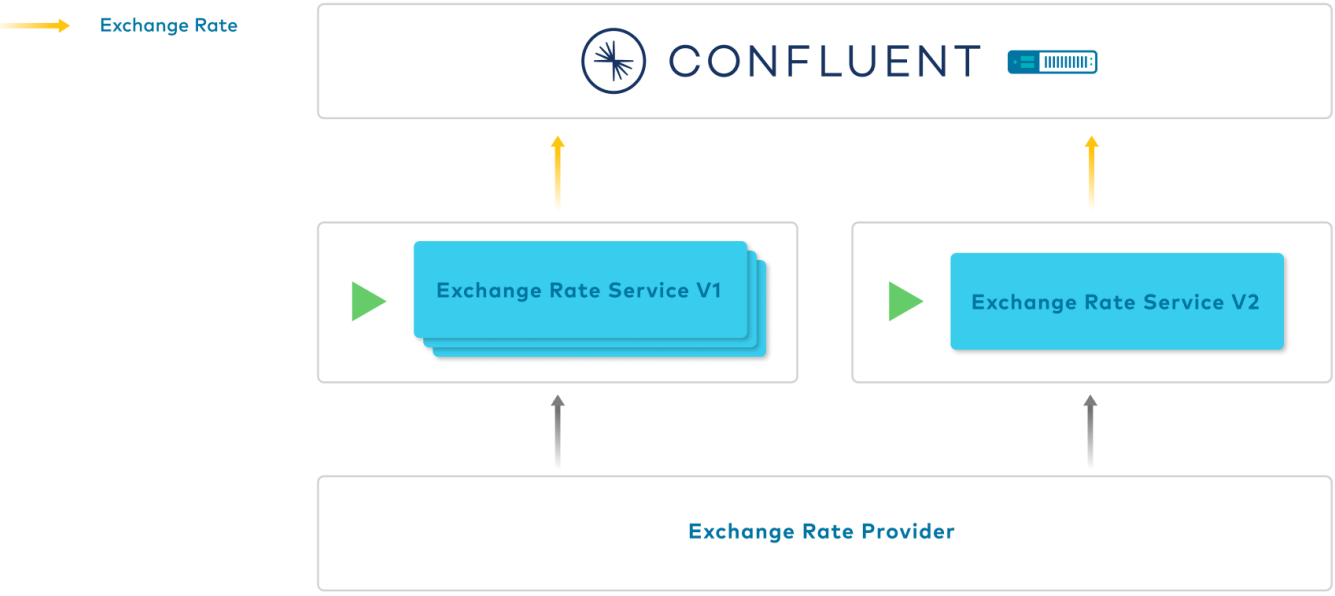


Starting the New Version After the Existing Version Has Properly Shut Down

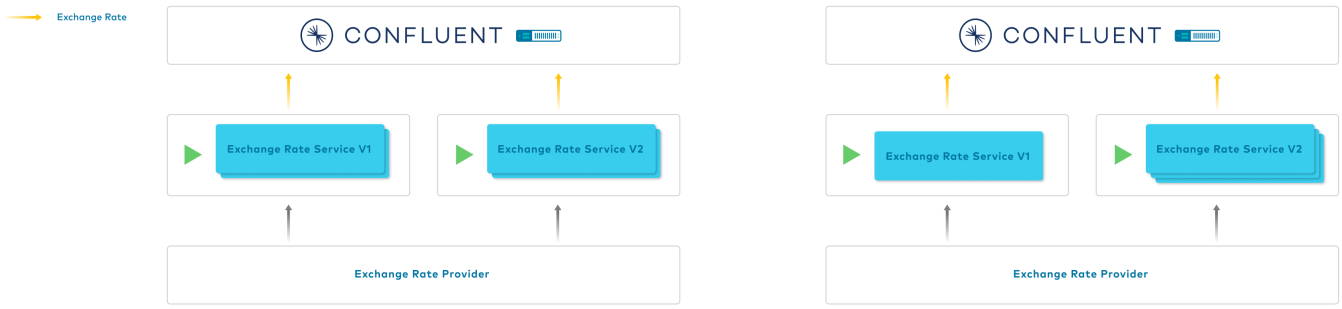


In some specific use cases, it might be unacceptable to cease event processing for the duration of the deployment operation. Our exchange rate service might have that constraint---we certainly would prefer that our customers don't lose currency rate updates. In a canary-style deployment, both versions of the microservice run in parallel, and we progressively reduce the number of instances for the old version while increasing the count for the new version. The rollback procedure is fairly straightforward: just scale down the new version and scale the old version back up to the original count.

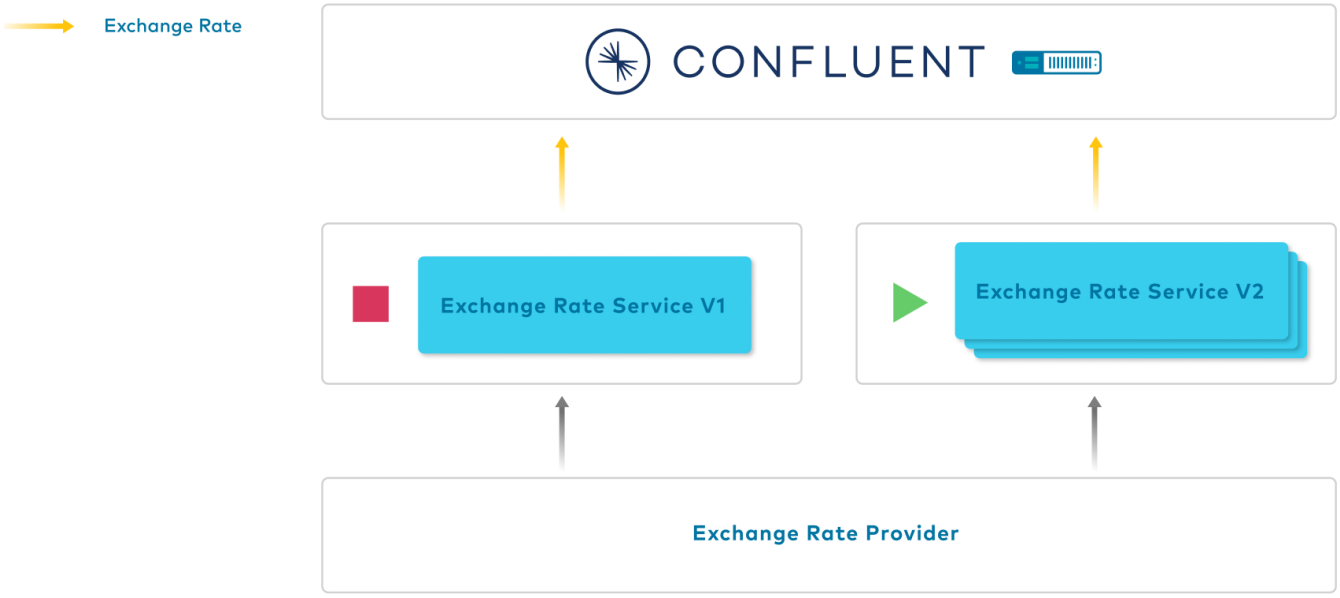
Deploying the First Instance of a New Version



Downscaling Old Version Instances While Scaling Up New Version Instances



Shutting Down the Old Version After Observing No Regression



Summary

In this paper we presented a sample currency exchange platform to illustrate the design and architecture of event-driven microservices using Apache Kafka and Confluent.

Apache Kafka and Confluent enable and extend all the microservice core principles. The primary functions of these technologies are well suited for microservices, including decoupling, separation of concerns, agility, and real-time streaming of event data. Developers and operators can use their preferred tools to deploy microservices since Apache Kafka imposes no precomposed opinion in the

code, build and deployment toolchain. Data can be moved within your organization as highly scalable distributed materialized views. Additionally, since topics can be easily exposed without impacting how producing microservices behave, organizations can offer data associated with microservices as a Service. Because the platform is resilient and fault tolerant, no batches need to be relaunched as events are simply processed (or reprocessed) in the event of a failure. High and abnormal traffic will be managed with back pressure powered by Kafka. Consumers will continue processing events as fast as they can without being overflowed by requests.

These event driven capabilities, when put to use in the service of a microservices architecture, allow businesses to be more productive and application development to be more agile by removing dependencies and impedences between disparate groups in an organization who work with the same data.