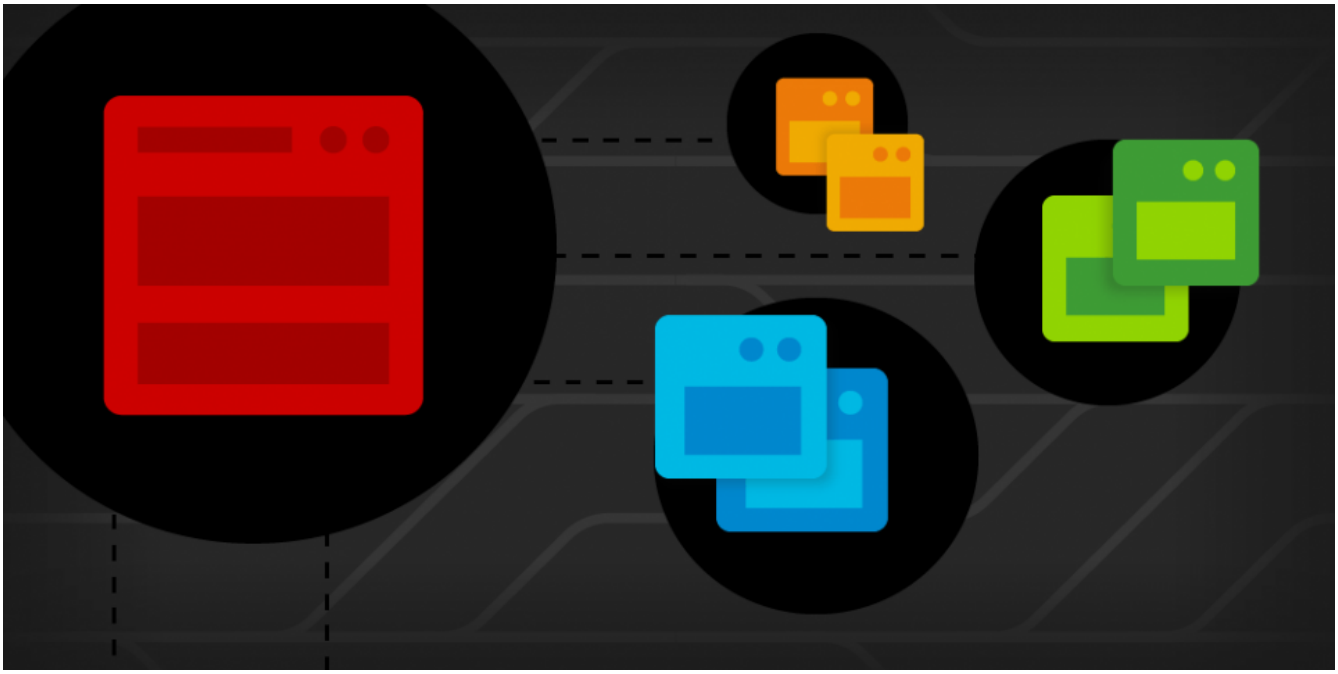
Article

Application modernization patterns with Apache Kafka, Debezium, and Kubernetes

June 14, 2021 Event-Driven, Kubernetes, Microservices

Bilgin Ibryam

Red Hat Product Manager



"*We build our computers the way we build our cities—over time, without a plan, on top of ruins.*"

Ellen Ullman wrote this in 1998, but it applies just as much today to the way we build modern applications; that is, over time, with short-term plans, on top of legacy software. In this article, I will introduce a few patterns and tools that I believe work well for thoughtfully modernizing legacy applications and building modern event-driven systems.

> **Note**: Consider joining me for my session at Red Hat Summit 2021 on June 16, or at the Event-driven architecture virtual event on June 22. I will present about tools and patterns for application modernization at both sessions, and you will be able to ask me questions live.

Application modernization in context

Application modernization refers to the process of taking an existing legacy application and modernizing its infrastructure—the internal architecture—to improve the velocity of new feature delivery, improve performance and scalability, expose the functionality for new use cases, and so on. Luckily, there is already a good classification of modernization and migration types, as shown in Figure 1.

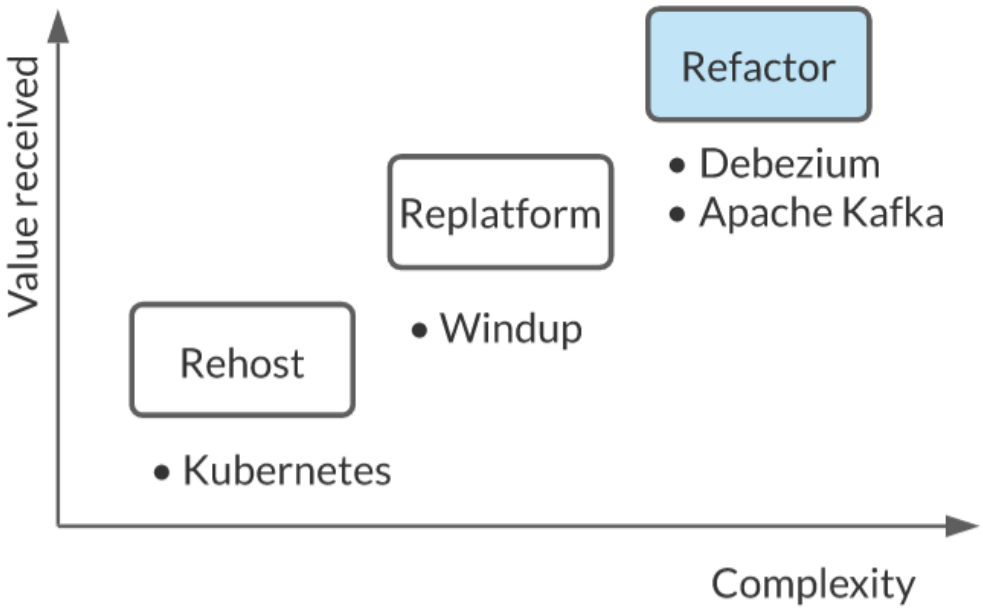


Figure 1: Three modernization types and the technologies we might use for them.

Depending on your needs and appetite for change, there are a few levels of modernization:

- **Retention**: The easiest thing you can do is to retain what you have and ignore the application's modernization needs. This makes sense if the needs are not yet pressing.
- **Retirement**: Another thing you could do is retire and get rid of the legacy application. That is possible if you discover the application is no longer being used.
- **Rehosting**: The next thing you could do is to rehost the application, which typically means taking an application as-is and hosting it on new infrastructure such as cloud infrastructure, or even on Kubernetes through something like KubeVirt. This is not a bad option if your application cannot be containerized, but you still want to reuse your Kubernetes skills, best practices, and infrastructure to manage a virtual machine as a container.
- **Replatforming**: When changing the infrastructure is not enough and you are doing a bit of alteration at the edges of the application without changing its architecture, replatforming is an option. Maybe you are changing the way the application is configured so that it can be containerized, or moving from a legacy Java EE runtime to an open source runtime. Here, you could use a tool like windup to analyze your application and return a report with what needs to be done.
- **Refactoring**: Much application modernization today focuses on migrating monolithic, on-premises applications to a cloud-native microservices architecture that supports faster release cycles. That involves refactoring and rearchitecting your application, which is the focus of this article.

For this article, we will assume we are working with a monolithic, on-premise application, which is a common starting point for modernization. The approach discussed here could also apply to other scenarios, such as a cloud migration initiative.

Challenges of migrating monolithic legacy applications

Deployment frequency is a common challenge for migrating monolithic legacy applications. Another challenge is scaling development so that more developers and teams can work on a common code base without stepping on each other's toes. Scaling the application to handle an increasing load in a reliable way is another concern. On the other hand, the expected benefits from a modernization include reduced time to market, increased team autonomy on the codebase, and dynamic scaling to handle the service load more efficiently. Each of these benefits offsets the work involved in modernization. Figure 2 shows an example infrastructure for scaling a legacy application for increased load.

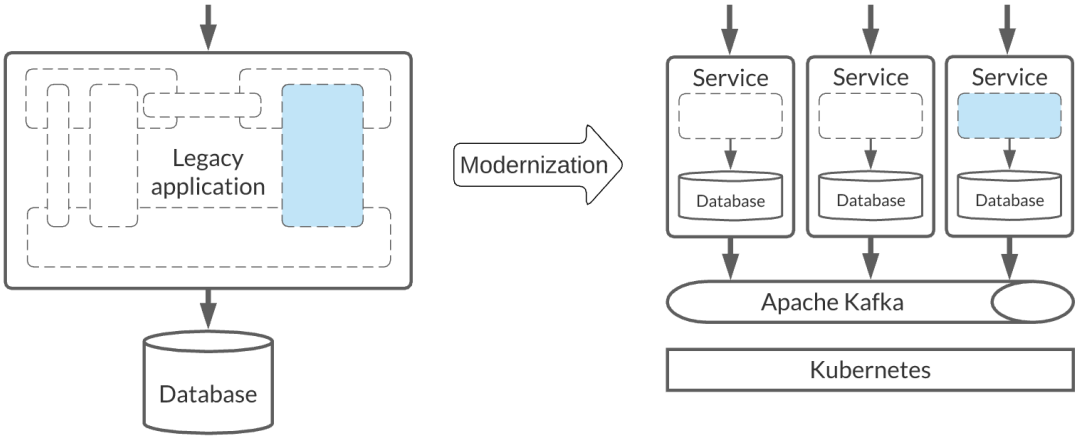


Figure 2: Refactoring a legacy application into event-driven microservices.

Envisioning the target state and measuring success

For our use case, the target state is an architectural style that follows microservices principles using open source technologies such as Kubernetes, Apache Kafka, and Debezium. We want to end up with independently deployable services modeled around a business domain. Each service should own its own data, emit its own events, and so on.

When we plan for modernization, it is also important to consider how we will measure the outcomes or results of our efforts. For that purpose, we can use metrics such as lead time for changes, deployment frequency, time to recovery, concurrent users, and so on.

The next sections will introduce three design patterns and three open source technologies—Kubernetes, Apache Kafka, and Debezium—that you can use to migrate from brown-field systems toward green-field, modern, event-driven services. We will start with the Strangler pattern.

The Strangler pattern

The Strangler pattern is the most popular technique used for application migrations. Martin Fowler introduced and popularized this pattern under the name of Strangler Fig Application, which was inspired by a type of fig that seeds itself in the upper branches of a tree and gradually evolves around the original tree, eventually replacing it. The parallel with application migration is that our new service is initially set up to wrap the existing system. In this way, the old and the new systems can coexist, giving the new system time to grow and potentially replace the old system. Figure 3 shows the main components of the Strangler pattern for a legacy application migration.

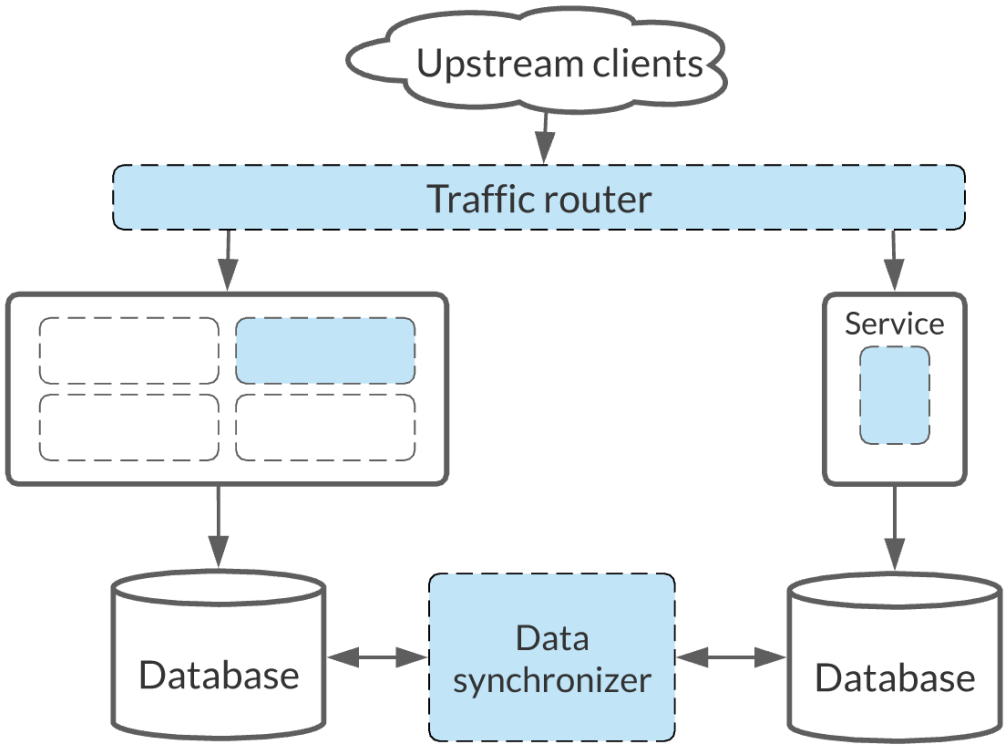


Figure 3: The Strangler pattern in a legacy application migration.

The key benefit of the Strangler pattern is that it allows low-risk, incremental migration from a legacy system to a new one. Let's look at each of the main steps involved in this pattern.

Step 1: Identify functional boundaries

The very first question is where to start the migration. Here, we can use domain-driven design to help us identify aggregates and the bounded contexts where each represents a potential unit of decomposition and a potential boundary for microservices. Or, we can use the event storming technique created by Antonio Brandolini to gain a shared understanding of the domain model. Other important considerations here would be how these models interact with the database and what work is required for database decomposition. Once we have a list of these factors, the next step is to identify the relationships and dependencies between the bounded contexts to get an idea of the relative difficulty of the extraction.

Armed with this information, we can proceed with the next question: Do we want to start with the service that has the least amount of dependencies, for an easy win, or should we start with the most difficult part of the system? A good compromise is to pick a service that is representative of many others and can help us build a good technology foundation. That foundation can then serve as a base for estimating and migrating other modules.

Step 2: Migrate the functionality

For the strangler pattern to work, we must be able to clearly map inbound calls to the functionality we want to move. We must also be able to redirect these calls to the new service and back if needed. Depending on the state of the legacy application, client applications, and other constraints, weighing our options for this interception might be straightforward or difficult:

- The easiest option would be to change the client application and redirect inbound calls to the new service. Job done.
- If the legacy application uses HTTP, then we're off to a good start. HTTP is very amenable to redirection and we have a wealth of transparent proxy options to choose from.
- In practice, it likely that our application will not only be using REST APIs, but will have SOAP, FTP, RPC, or some kind of traditional messaging endpoints, too. In this case, we may need to build a custom protocol translation layer with something like Apache Camel.

Interception is a potentially dangerous slippery slope: If we start building a custom protocol translation layer that is shared by multiple services, we risk adding too much intelligence to the shared proxy that services depend on. This would move us away from the "smart microservices, dumb pipes" mantra. A better option is to use the Sidecar pattern, illustrated in Figure 4.

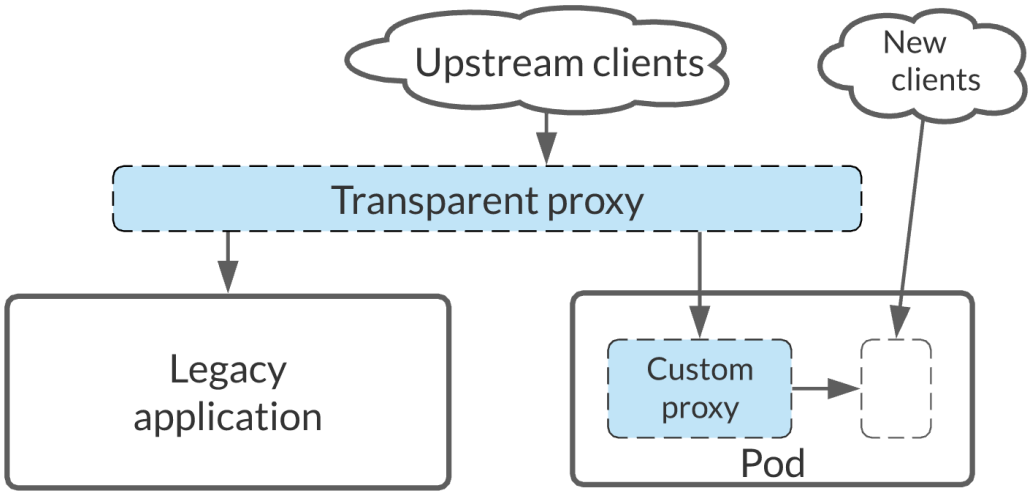


Figure 4: The Sidecar pattern.

Rather than placing custom proxy logic in a shared layer, make it part of the new service. But rather than embedding the custom proxy in the service at compile-time, we use the Kubernetes sidecar pattern and make the proxy a runtime binding activity. With this pattern, legacy clients use the protocol-translating proxy and new clients are offered the new service API. Inside the proxy, calls are translated and directed to the new service. That allows us to reuse the proxy if needed. More importantly, we can easily decommission the proxy when it is no longer needed by legacy clients, with minimal impact on the newer services.

Step 3: Migrate the database

Once we have identified the functional boundary and the interception method, we need to decide how we will approach *database strangulation*—that is, separating our legacy database from application services. We have a few paths to choose from.

Database first

In a database-first approach, we separate the schema first, which could potentially impact the legacy application. For example, a `SELECT` might require pulling data from two databases,

and an `UPDATE` can lead to the need for distributed transactions. This option requires changes to the source application and doesn't help us demonstrate progress in the short term. That is not what we are looking for.

Code first

A code-first approach lets us get to independently deployed services quickly and reuse the legacy database, but it could give us a false sense of progress. Separating the database can turn out to be challenging and hide future performance bottlenecks. But it is a move in the right direction and can help us discover the data ownership and what needs to be split into the database layer later.

Code and database together

Working on the code and database together can be difficult to aim for from the get-go, but it is ultimately the end state we want to get to. Regardless of how we do it, we want to end up with a separate service and database; starting with that in mind will help us avoid refactoring later.

Having a separate database requires data synchronization. Once again, we can choose from a few common technology approaches.

Triggers

Most databases allow us to execute custom behavior when data is changed. In some cases, that could even be calling a web service and integrating with another system. But how triggers are implemented and what we can do with them varies between databases. Another significant drawback here is that using triggers requires changing the legacy database, which we might be reluctant to do.

Queries

We can use queries to regularly check the source database for changes. The changes are typically detected with implementation strategies such as timestamps, version numbers, or status column changes in the source database. Regardless of the implementation strategy, polling always leads to the dilemma between polling often and creating overhead over the source database, or missing frequent updates. While queries are simple to install and use, this approach has significant limitations. It is unsuitable for mission-critical applications with frequent database interactions.

Log readers

Log readers identify changes by scanning the database transaction log files. Log files exist for database backup and recovery purposes and provide a reliable way to capture all changes including `DELETE`s. Using log readers is the least disruptive option because they require no modification to the source database and they don't have a query load. The main downside of this approach is that there is no common standard for the transaction log files and we'll need specialized tools to process them. This is where Debezium fits in.

Before moving on to the next step, let's see how using Debezium with the log reader approach works.

Change data capture with Debezium

When an application writes to the database, changes are recorded in log files, then the database tables are updated. For MySQL, the log file is `binlog`; for PostgreSQL, it is the `write-ahead-log`; and for MongoDB it's the `op` log. The good news is Debezium has connectors for different databases, so it does the hard work for us of understanding the format of all of these log files. Debezium can read the log files and produce a generic abstract event into a messaging system such as Apache Kafka, which contains the data changes. Figure 5 shows Debezium connectors as the interface for a variety of databases.

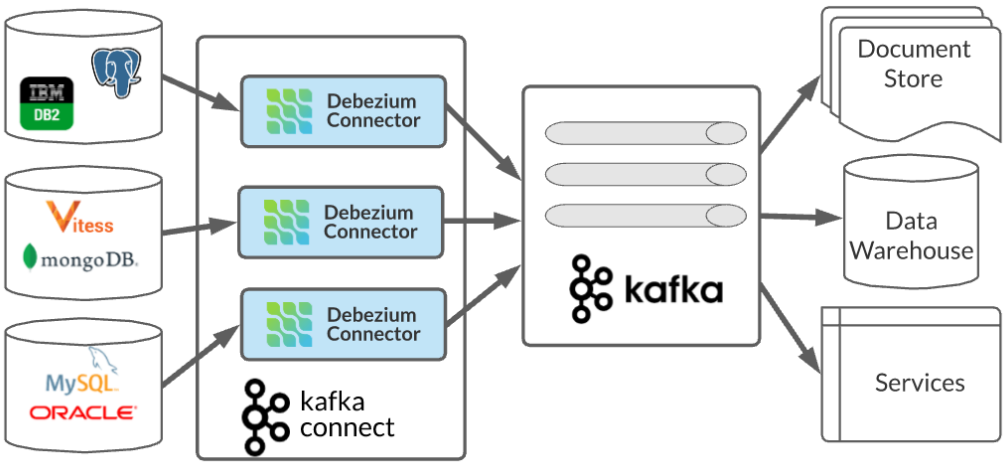


Figure 5: Debezium connectors in a microservices architecture.

Debezium is the most widely used open source change data capture (CDC) project with multiple connectors and features that make it a great fit for the Strangler pattern.

Why is Debezium a good fit for the Strangler pattern?

One of the most important reasons to consider the Strangler pattern for migrating monolithic legacy applications is reduced risk and the ability to fall back to the legacy application. Similarly, Debezium is completely transparent to the legacy application, and it doesn't require any changes to the legacy data model. Figure 6 shows Debezium in an example microservices architecture.

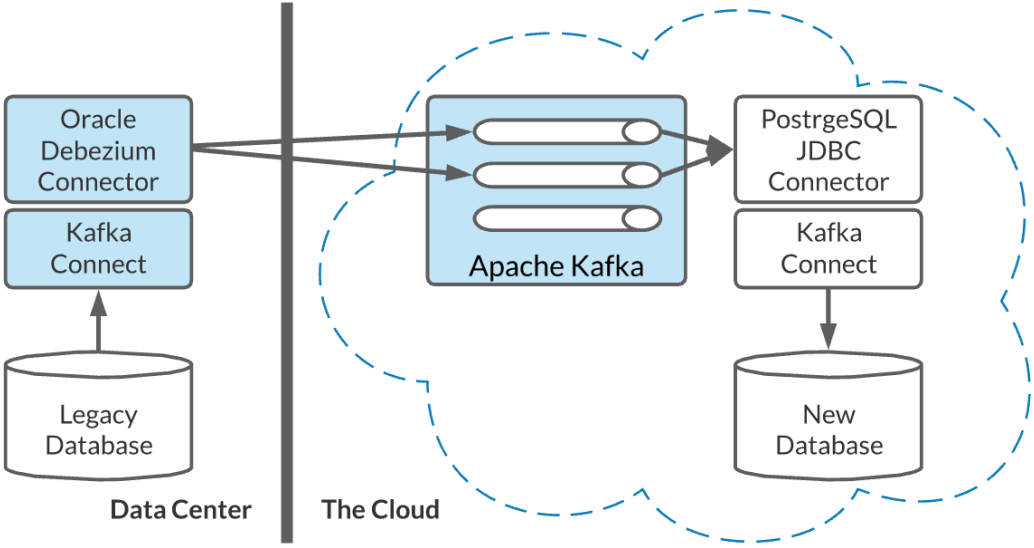


Figure 6: Debezium deployment in a hybrid-cloud environment.

With a minimal configuration to the legacy database, we can capture all the required data. So at any point, we can remove Debezium and fall back to the legacy application if we need to.

Debezium features that support legacy migrations

Here are some of Debezium's specific features that support migrating a monolithic legacy application with the Strangler pattern:

- **Snapshots**: Debezium can take a snapshot of the current state of the source database, which we can use for bulk data imports. Once a snapshot is completed, Debezium will start streaming the changes to keep the target system in sync.
- **Filters**: Debezium lets us pick which databases, tables, and columns to stream changes from. With the Strangler pattern, we are not moving the whole application.
- **Single message transformation (SMT)**: This feature can act like an anti-corruption layer and protect our new data model from legacy naming, data formats, and even let us filter out obsolete data
- **Using Debezium with a schema registry**: We can use a schema registry such as Apicurio with Debezium for schema validation, and also use it to enforce version compatibility checks when the source database model changes. This can prevent changes from the source database from impacting and breaking the new downstream message consumers.
- **Using Debezium with Apache Kafka**: There are many reasons why Debezium and Apache Kafka work well together for application migration and modernization. Guaranteed ordering of database changes, message compaction, the ability to re-read changes as many times as needed, and tracking transaction log offsets are all good examples of why we might choose to use these tools together.

Step 4: Releasing services

With that quick overview of Debezium, let's see where we are with the Strangler pattern. Assume that, so far, we have done the following:

- Identified a functional boundary.

- Migrated the functionality.
- Migrated the database.
- Deployed the service into a Kubernetes environment.
- Migrated the data with Debezium and kept Debezium running to synchronize ongoing changes.

At this point, there is not yet any traffic routed to the new services, but we are ready to release the new services. Depending on our routing layer's capabilities, we can use techniques such as dark launching, parallel runs, and canary releasing to reduce or remove the risk of rolling out the new service, as shown in Figure 7.

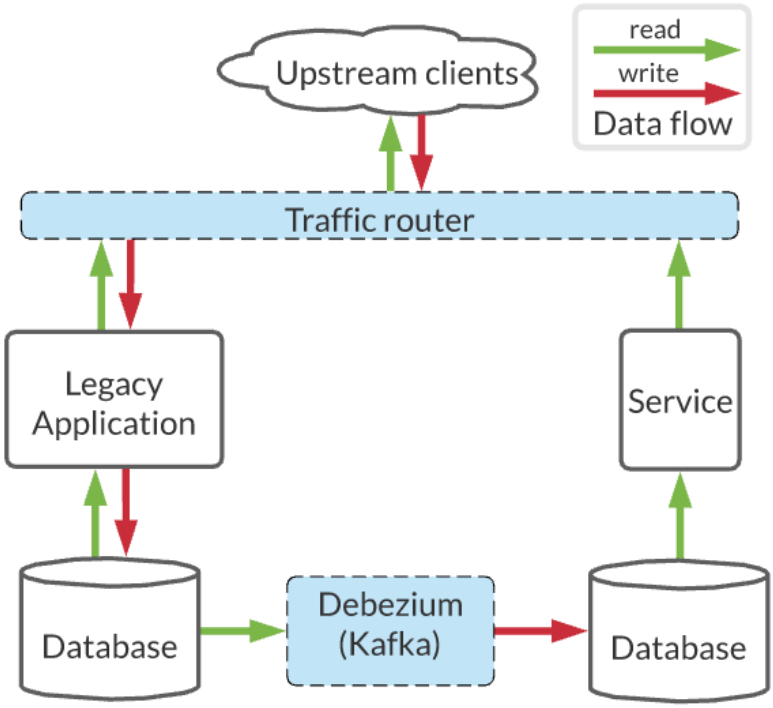


Figure 7: Directing read traffic to the new service.

What we can also do here is to only direct read requests to our new service initially, while continuing to send the writes to the legacy system. This is required as we are replicating changes in a single direction only.

When we see that the read operations are going through without issues, we can then direct the write traffic to the new service. At this point, if we still need the legacy application to operate for whatever reason, we will need to stream changes from the new services toward the legacy application database. Next, we'll want to stop any write or mutating activity in the legacy module and stop the data replication from it. Figure 8 illustrates this part of the pattern implementation.

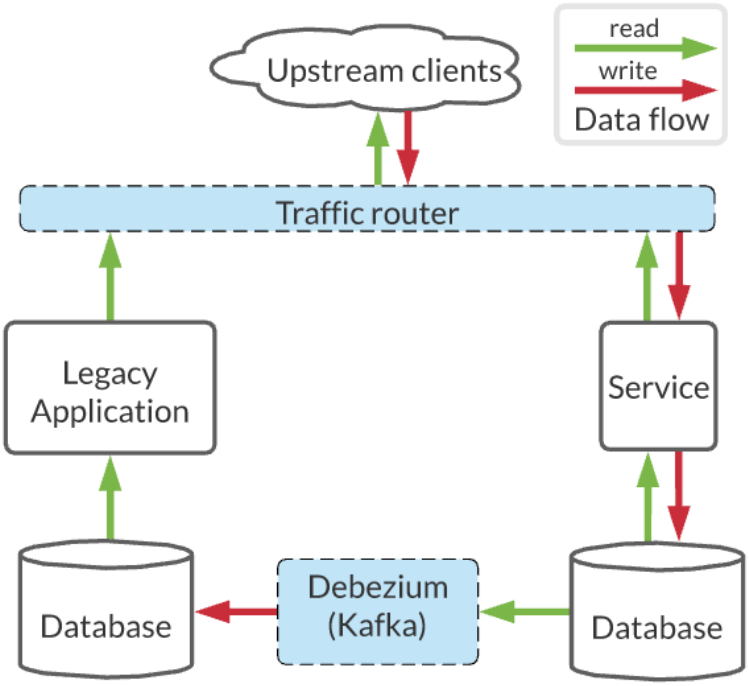


Figure 8: Directing read and write traffic to the new service.

Since we still have legacy read operations in place, we are continuing the replication from the new service to the legacy application. Eventually, we'll stop all operations in the legacy module and stop the data replication. At this point, we will be able to decommission the migrated module.

We've had a broad look at using the Strangler pattern to migrate a monolithic legacy application, but we are not quite done with modernizing our new microservices-based architecture. Next, let's consider some of the challenges that come later in the modernization process and how Debezium, Apache Kafka, and Kubernetes might help.

After the migration: Modernization challenges

The most important reason to consider using the Strangler pattern for migration is the reduced risk. This pattern gives value steadily and allows us to demonstrate progress through frequent releases. But migration alone, without enhancements or new "business value" can be a hard sell to some stakeholders. In the longer-term modernization process, we also want to enhance our existing services and add new ones. With modernization initiatives, very often, we are also tasked with setting the foundation and best practices for building modern applications that will follow. By migrating more and more services, adding new ones, and in general by transitioning to the microservices architecture, new challenges will come up, including the following:

- Automating the deployment and operating a large number of services.
- Performing dual-writes and orchestrating long-running business processes in a reliable and scalable manner.
- Addressing the analytical and reporting needs.

There are all challenges that might not have existed in the legacy world. Let's explore how we can address a few of them using a combination of design patterns and technologies.

Challenge 1: Operating event-driven services at scale

While peeling off more and more services from the legacy monolithic application, and also creating new services to satisfy emerging business requirements, the need for automated deployments, rollbacks, placements, configuration management, upgrades, self-healing becomes apparent. These are the exact features that make Kubernetes a great fit for operating large-scale microservices. Figure 9 illustrates.

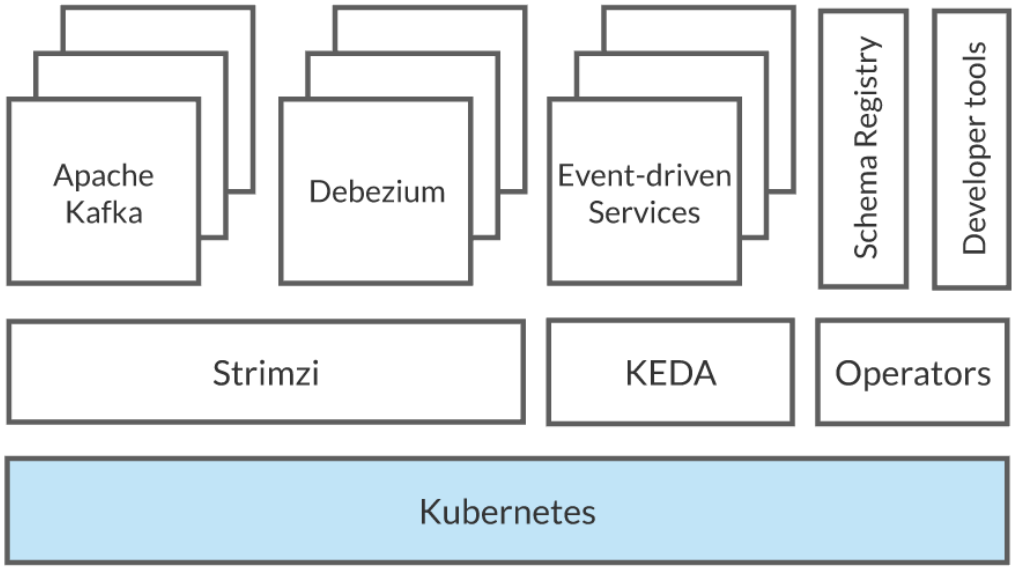


Figure 9: A sample event-driven architecture on top of Kubernetes.

When we are working with event-driven services, we will quickly find that we need to automate and integrate with an event-driven infrastructure—which is where Apache Kafka and other projects in its ecosystem might come in. Moreover, we can use Kubernetes Operators to help automate the management of Kafka and the following supporting services:

- Apicurio Registry provides an Operator for managing Apicurio Schema Registry on Kubernetes.
- Strimzi offers Operators for managing Kafka and Kafka Connect clusters declaratively on Kubernetes.
- KEDA (Kubernetes Event-Driven Autoscaling) offers workload auto-scalers for scaling up and down services that consume from Kafka. So, if the consumer lag passes a threshold, the Operator will start more consumers up to the number of partitions to catch up with message production.
- Knative Eventing offers event-driven abstractions backed by Apache Kafka.

Note: Kubernetes not only provides a target platform for application modernization but also allows you to grow your applications on top of the same foundation into a large-scale event-driven architecture. It does that through automation of user workloads, Kafka workloads, and other tools from the Kafka ecosystem. That said, not everything has to run on your Kubernetes. For example, you can use a fully managed Apache Kafka or a schema registry service from Red Hat and automatically bind it to your application using Kubernetes Operators. Creating a multi-availability-zone (multi-AZ) Kafka cluster on Red Hat OpenShift Streams for Apache Kafka takes less than a minute and is completely free during our trial period. Give it a try and help us shape it with your early feedback.

Now, let's see how we can meet the remaining two modernization challenges using design patterns.

Challenge 2: Avoiding dual-writes

Once you build a couple of microservices, you quickly realize that the hardest part about them is data. As part of their business logic, microservices often have to update their local data store. At the same time, they also need to notify other services about the changes that happened. This challenge is not so obvious in the world of monolithic applications and legacy distributed transactions. How can we avoid or resolve this situation the cloud-native way? The answer is to only modify one of the two resources—the database—and then drive the update of the second one, such as Apache Kafka, in an eventually consistent manner. Figure 10 illustrates this approach.

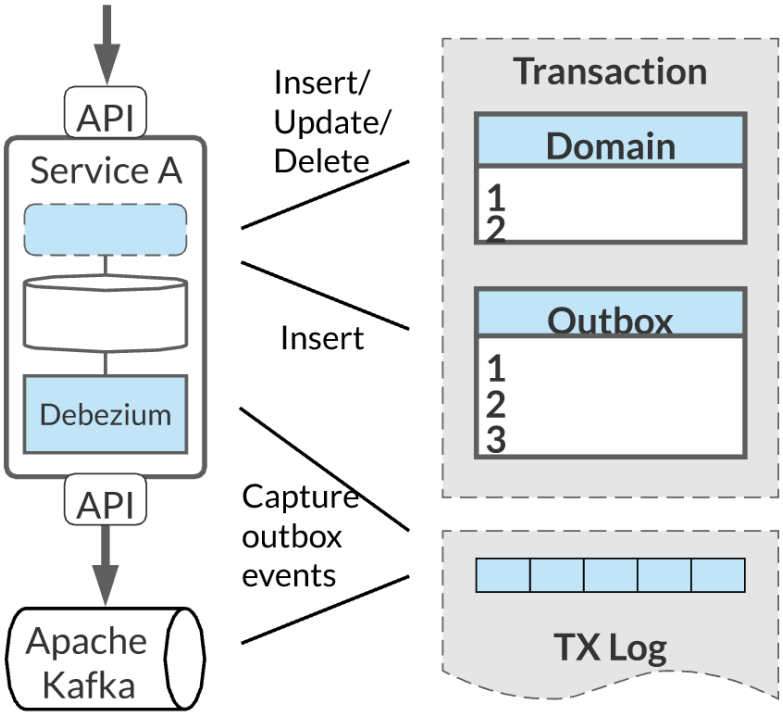


Figure 10: The Outbox pattern.

Using the Outbox pattern with Debezium lets services execute these two tasks in a safe and consistent manner. Instead of directly sending a message to Kafka when updating the database, the service uses a single transaction to both perform the normal update and insert the message into a specific outbox table within its database. Once the transaction has been written to the database's transaction log, Debezium can pick up the outbox message from there and send it to Apache Kafka. This approach gives us very nice properties. By synchronously writing to the database in a single transaction, the service benefits from "read your own writes" semantics, where a subsequent query to the service will return the newly persisted record. At the same time, we get reliable, asynchronous, propagation to other services via Apache Kafka. The Outbox pattern is a proven approach for avoiding dual-writes for scalable event-driven microservices. It solves the inter-service communication challenge very elegantly without requiring all participants to be available at the same time, including Kafka. I believe Outbox will become one of the foundational patterns for designing scalable event-driven microservices.

Challenge 3: Long-running transactions

While the Outbox pattern solves the simpler inter-service communication problem, it is not sufficient alone for solving the more complex long-running, distributed business transactions use case. The latter requires executing multiple operations across multiple microservices and applying consistent all-or-nothing semantics. A common example for demonstrating this requirement is the booking-a-trip use case consisting of multiple parts where the flight and accommodation must be booked together. In the legacy world, or with a monolithic architecture, you might not be aware of this problem as the coordination between the modules is done in a single process and a single transactional context. The distributed world requires a different approach, as illustrated in Figure 11.

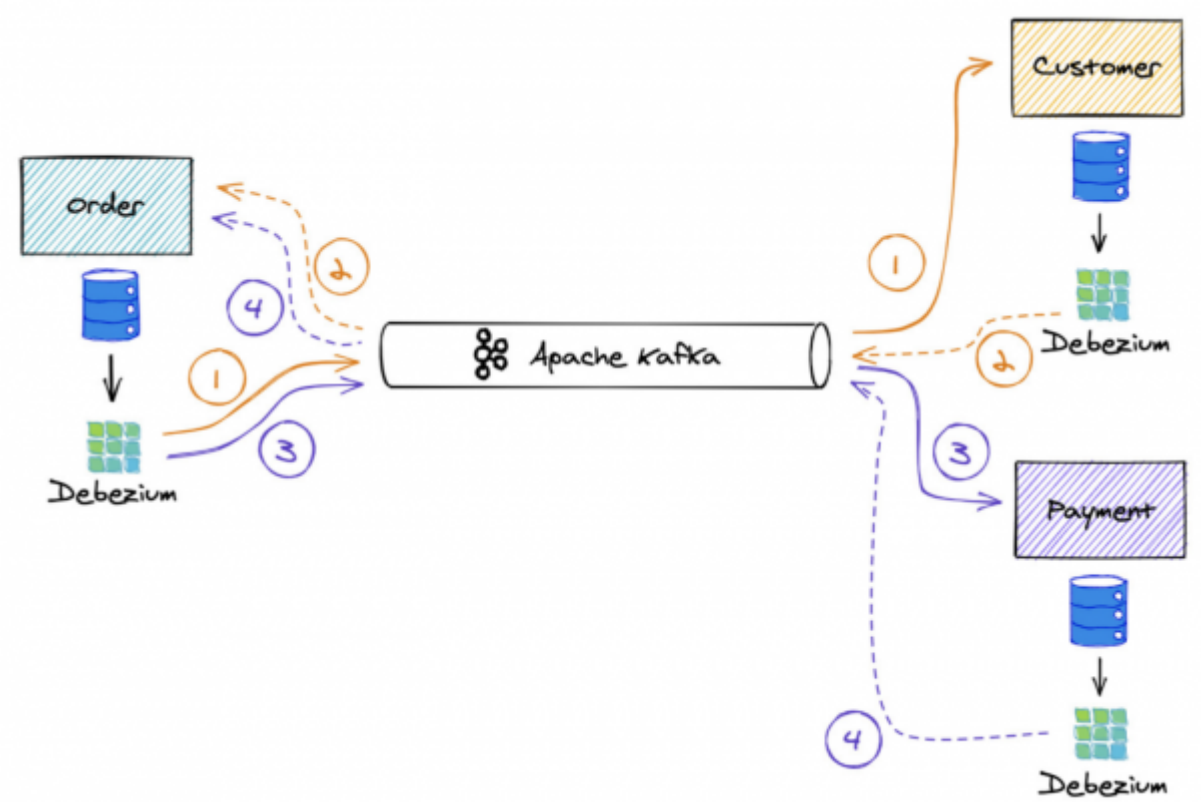


Figure 11: The Saga pattern implemented with Debezium.

The Saga pattern offers a solution to this problem by splitting up an overarching business transaction into a series of multiple local database transactions, which are executed by the participating services. Generally, there are two ways to implement distributed *sagas*:

- **Choreography**: In this approach, one participating service sends a message to the next one after it has executed its local transaction.
- **Orchestration**: In this approach, one central coordinating service coordinates and invokes the participating services.

Communication between the participating services might be either synchronous, via HTTP or gRPC, or asynchronous, via messaging such as Apache Kafka.

> **Note**: See *Patterns for distributed transactions within a microservices architecture*.

The cool thing here is that you can implement sagas using Debezium, Apache Kafka, and the Outbox pattern. With these tools, it is possible to take advantage of the orchestration approach and have one place to manage the flow of a saga and check the status of the

overarching saga transaction. We can also combine orchestration with asynchronous communication to decouple the coordinating service from the availability of participating services and even from the availability of Kafka. That gives us the best of both worlds: orchestration *and* asynchronous, non-blocking, parallel communication with participating services, without temporal coupling.

Combining the Outbox pattern with the Sagas pattern is an awesome, event-driven implementation option for the long-running business transactions use case in the distributed services world. See *Saga Orchestration for Microservices Using the Outbox Pattern* (InfoQ) for a detailed description. Also see an implementation example of this pattern on GitHub.

Conclusion

The Strangler pattern, Outbox pattern, and Saga pattern can help you migrate from brown-field systems, but at the same time, they can help you build green-field, modern, event-driven services that are future-proof.

Kubernetes, Apache Kafka, and Debezium are open source projects that have turned into de facto standards in their respective fields. You can use them to create standardized solutions with a rich ecosystem of supporting tools and best practices.

The one takeaway from this article—and my upcoming talk at Red Hat Summit 2021— is the realization that modern software systems are like cities: They evolve over time, on top of legacy systems. Using proven patterns, standardized tools, and open ecosystems will help you create long-lasting systems that grow and change with your needs.

Want more about microservices? See Bee Travels: A microservices coding adventure.

Last updated: June 21, 2021