Powering the API world

EBOOK

Making the Move from Monolithic Architectures to Microservices

KongHQ.com

Table of contents

In this eBook, we'll discuss how to move from a monolithic application architecture to a microservices-based architecture.

We'll highlight the pros and cons, as well as common mistakes to avoid. This will include incentives, considerations, strategies for transitioning to microservices, and best practices for the process.

03 **Introduction**

04 **Should you blow up your monolith?**

05 **Considerations when adopting microservices**

06 **Two ways for microservice communication**
- Communications scenarios

08 **A technical look into the transition**
- Boundaries
- Testing
- Transition strategies
  - Ice cream scoop strategy
  - LEGO strategy
  - Nuclear option strategy
- The database
- Routing and versioning
- Libraries and security

14 **Containers and service mesh**

15 **Conclusion**

Introduction

We're living in a revolutionary age for software, with massive changes in the way we build, deploy, and consume our applications or "services." These changes are not just technical but organizational.

New paradigms, patterns, and foundations are substantially shaping the industry, and we're entering a new era of technological and cultural changes. Open source software has always been an important contributor to the industry, but in this day and age, it's becoming a catalyst for enterprise adoption of new paradigms and architectures.

Decoupling the monolithic application is much like taking a single cumbersome structure, breaking it down, and rebuilding it with LEGO pieces. It can be a difficult process, but once completed, each independent piece serves a unique purpose. And when all of the pieces are put together, the whole structure functions like a single entity. In the case of an application, this means moving away from the single, large codebase to smaller isolated services, all communicating with each other to deliver the functionality of the whole application together. And since you're likely to have existing traffic and clients using the application, there needs to be a way to keep the LEGO structure functional even during refactoring.

Should you blow up your monolith?

Should you blow up your monolith? This is a good question to ask before dedicating the time and resources needed to undertake a move to microservices.

What are the overall goals of the organization and what characteristics of the application architecture will best achieve these goals? The pros and cons of moving to a microservices architecture are both equally compelling.

Microservices offer much more flexibility when deploying or updating the application thanks to shorter and more focused development cycles. This carries over into a far more efficient overall application development process as dev teams can be broken down into smaller groups, such as "pizza teams," which can work independently while evolving the application as a whole. Also, microservices enable an application to be much easier to scale, which aligns with any existing or potential cloud strategy.

Ironically, some of the characteristics of a microservices architecture which are considered benefits, can also be seen as drawbacks and may make one want to consider staying with a monolithic architecture.

While microservices provide greater flexibility in overall application development and management, it also increases the complexity due to the **need for more coordination and communication for deploying microservices.**

There are simply more plates to spin. This will also impact the performance of the application. A monolith is a single, self-contained codebase so application performance considerations are essentially confined to development. Whereas microservices, which can be widely distributed, require greater demand from the network in order for each microservice to communicate with each other toward a common objective.

These are only a handful of the pros and cons organizations should consider when deciding between microservices or monolith, but performance, deployability, and scalability are definitely going to be at the top of the list.

Considerations when adopting microservices

As mentioned, a common driver for a move to microservices is the difficulty of maintaining a monolithic codebase and the desire to achieve greater business agility. But this doesn't mean that transitioning to microservices is always simple.

Before starting a transition initiative, identify the biggest pain points and boundaries in the monolithic codebase and decouple them into separate services. Don't put too much energy into the "sizing" of these services as it pertains to the amount of code behind them. It's much more important to make sure these services can continue to handle their specific business logic within the boundary to which they're allocated.

With microservices, it's common for developers and architects to focus too much on the size of code blocks of services decoupled from the monolithic codebase. But the reality is that these services will be as big as they need to handle their specific business logic. Too much decoupling, and you may end up with too many moving parts. There's always going to be time in the future to decouple services even further as you learn the pain points of building and operating under this new architecture.

Another important consideration to keep in mind as you transition to microservices is that your existing business will still be running and growing on the monolith. Therefore, "shared responsibilities" can win the day, with the development efforts split into two smaller teams: one that maintains the old codebase, while the other one works on the new codebase.

Doing this requires a keen eye on resource allocation, as this split workload can cause friction between the two teams as maintaining a large monolithic application isn't as exciting as working on new technologies.

This problem can be solved in a larger team by introducing a rotation between the two projects for team members. This exposes all team members to new development tasks and builds new skills while keeping fresh eyes on the monolithic codebase.

This brings us to the next consideration: time. The truth is that transitioning to **microservices won't happen overnight** — no matter how hard you think about it and how much you plan for the transition.

Two ways for microservice communication

Besides the circumstances described above, one of the biggest lessons to learn is understanding when to use service-to-service communication patterns as opposed to asynchronous patterns.

Even today, the narrative seems to be exclusively pushing for service-to-service communication, although that's not necessarily the best way to implement some very specific use cases.

The main difference between the two patterns is that an asynchronous system won't directly communicate with another microservice but will instead propagate an event "asynchronously" to which another service can intercept and react. (Usually, log collectors like Apache Kafka can be used, but also RabbitMQ or cloud alternatives like AWS SQS.)

Figure 1: Service-to-service communication example — notice the "sidecar" characteristic of this option

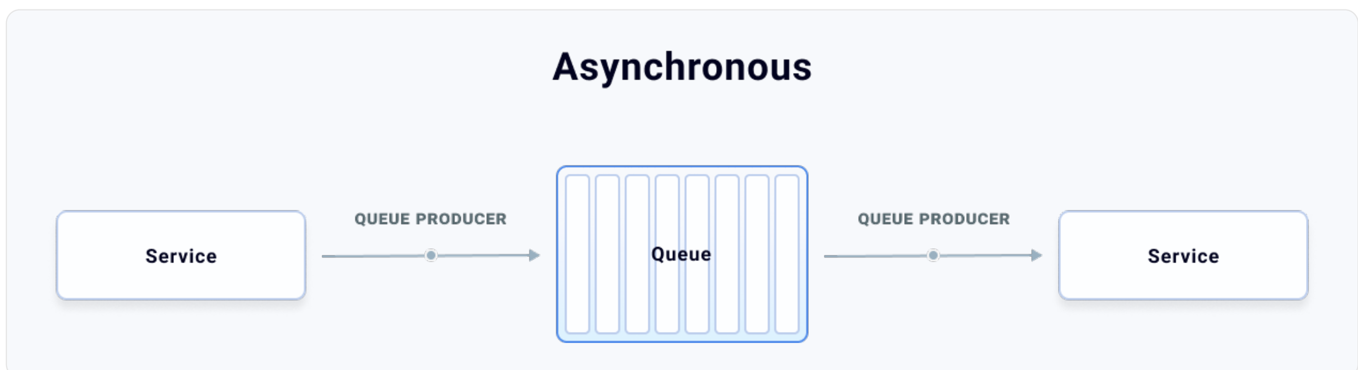


Figure 2: Asynchronous communication from service to service through a proxy

This pattern is very useful when an immediate response isn't required since it basically implements error handling within the architectural pattern from the beginning. Therefore, it makes a great candidate to propagate eventually consistent data state changes across every microservice avoiding creating inconsistencies.

Communications scenarios

Let's assume that you have two microservices, "Orders" and "Invoices," and that every time an order is made, an invoice also needs to be created.

In a service-to-service pattern, the "Orders" service will have to issue a request to "Invoices" every time an order is created. But if the "Invoices" microservice is completely down and not available, eventually that request will timeout and fail. This would be true even if the request was issued by an intermediate proxy — it will still try to make the request over and over again but eventually, it will timeout and fail leading to data inconsistency.

With an asynchronous pattern, the "Orders" microservice will create an event into a log collector/queue that will be asynchronously processed by the "Invoice" microservice whenever it decides to poll or listen for new events. Therefore even if "Invoices" is currently down for a long period of time, those invoices won't be lost and the data will be eventually consistent across the system as soon as the microservice goes online again (and assuming that the log collector will persist the events).

Using a system like Apache Kafka also makes it easy for the developer to replay a series of events starting from a specific timestamp in order to reproduce the state of data at any given time either locally or on staging.

Service-to-Service	Asynchronous
Microservices and clients **directly** consume and invoke other microservices.	Microservices and clients push events into an event collector that's being consumed by other Microservices.
Ideal for clients that require an **immediate** response, or need to aggregate multiple services together.	Ideal for microservice-to-microservice communication for changing state **without** requiring an immediate response.
Done via HTTP, TCP/UDP, gRPC, etc.	Done via Kafka, RabbitMQ, AWS SQS, etc.
Example: Making a request to retrieve an immediate response of some sort (e.g., retrieve list of users).	Example: Making a request that doesn't require an immediate response (e.g., "orderCreated" event that triggers an invoice creation by other microservice).

Figure 3: Differences between service-to-service and asynchronous communications

Ideally, microservices-oriented architecture will leverage all of these patterns depending on the use case, but service-to-service isn't the only answer.

A technical look into the transition

Now that there's an understanding of what monolithic and microservices bring to the table, it's time to think about approaching the technical transition. Different strategies can be adopted, but all of them share the same preparation tasks: identifying boundaries and improving testing.

These preparation tasks are fundamental to the success of the transition and can't be overlooked.

Boundaries

The first thing to figure out before starting the transition is what services need to be created or broken out from the monolithic codebase and what your architecture will look like in a completed microservices-based architecture, how big or how small you want them to be, and how they will communicate with each other. A starting point is to examine the boundaries that are more negatively impacted by the monolith, for example, those that you deploy, change, or scale more often than the others.

Testing

Transitioning to microservices is effectively a refactoring.Therefore, all the regular precautions followed before a "regular" refactoring also apply here — in particular, testing.

As the transition proceeds, so do changes to how the system fundamentally works. It's important to be mindful that, following the transitional phase, all the functionalities that once existed in the monolith are still working in the redesigned architecture.

A best practice here is before attempting any change to include a solid and reliable suite of integration and regression tests for the monolith.

Some of these tests will likely fail along the way, but having well-tested functionality will help track down what isn't working as expected.

Transition strategies

There are several strategies to choose from when transitioning to microservices, each with their respective pros and cons which should be considered prior to this process.

Ice Cream Scoop Strategy

This strategy calls for a gradual transition from a monolithic application to a microservices-oriented architecture by “scooping out” different components within the monolith into separate services. This transition is gradual and there will be times when both the monolith and the microservices will exist at the same time.

Pros: Gradually migrating with reduced risks and without affecting much of the uptime and the end user experience.

Cons: It’s a process that will take time to fully execute.

LEGO Strategy

For organizations that believe their monolith is too big to refactor and prefer to keep it as it is, this strategy advocates for only building the new features as microservices. Effectively, this won’t fix the problems with the existing monolithic codebase, but it will fix problems for future expansions of the product. Basically, this option calls for stacking the monolithic and microservices on top of each other in a hybrid architecture.

Pros: A faster migration with less work on the monolith.

Cons: The monolith will continue to have its original problems, and new APIs will most likely have to be created to support the microservices-oriented features.

The LEGO strategy helps buy some time on the big refactor but ultimately runs the risk of adding more tech debt on top of the monolith.

Nuclear Option Strategy

This third strategy is rarely adopted. The entire monolithic application is rewritten into microservices all at once, perhaps still supporting the old monolith with hotfixes and patches but while building every new feature in the new codebase.

Pros: Allows the organization to rethink how things are done, effectively rewriting the app from scratch.

Cons: It requires rewriting the app from scratch, which can introduce additional operational overhead.

An outcome of the nuclear option that should also be considered is that you may end up with a “second system syndrome” where end users will be affected by a stalled monolith until the new architecture is ready for deployment.

The Database

The end goal for the new microservice-oriented architecture is to **not rely on one database** that every service utilizes. Since this new architecture requires decoupling the business logic in different services, you will also want to decouple the database access and have one database for each microservice. Sometimes this is inevitable. For example, a microservice that handles users and their data will benefit from a relational datastore while a microservice that deals with orders or logs will benefit from high-performance writes to an eventually consistent datastore like Cassandra.

Sometimes some services will indeed use the same underlying database technology. Although it would still be better to have separate database clusters dedicated to each microservice, for low throughput use cases, sometimes it's just too convenient to leverage the same datastore nodes but with data stored in different logical databases/ keyspaces. The cons of this solution are that if a microservice — for whatever reason — impacts the database uptime, then the other microservices will also be impacted since they're talking to the same database nodes. This should be avoided since it breaks compartmentalization.

Regardless of setup, consistency of data will be an issue. There's going to be a limbo period when the old codebase is still writing and reading to the underlying database, and the new database for microservices is using a separate store for data. Therefore, writes or reads made by the monolith won't be visible to the microservice and vice-versa. This isn't an easy problem to solve, and there are a few options to consider, including:

- Writes from the monolith are also propagated to the microservice database and vice versa.
- Building an easy-to-use API for the old database that the microservice will use to query data from the old database.
- Introduction of an event collector layer (like Kafka) that will take care of propagating writes to both data stores.

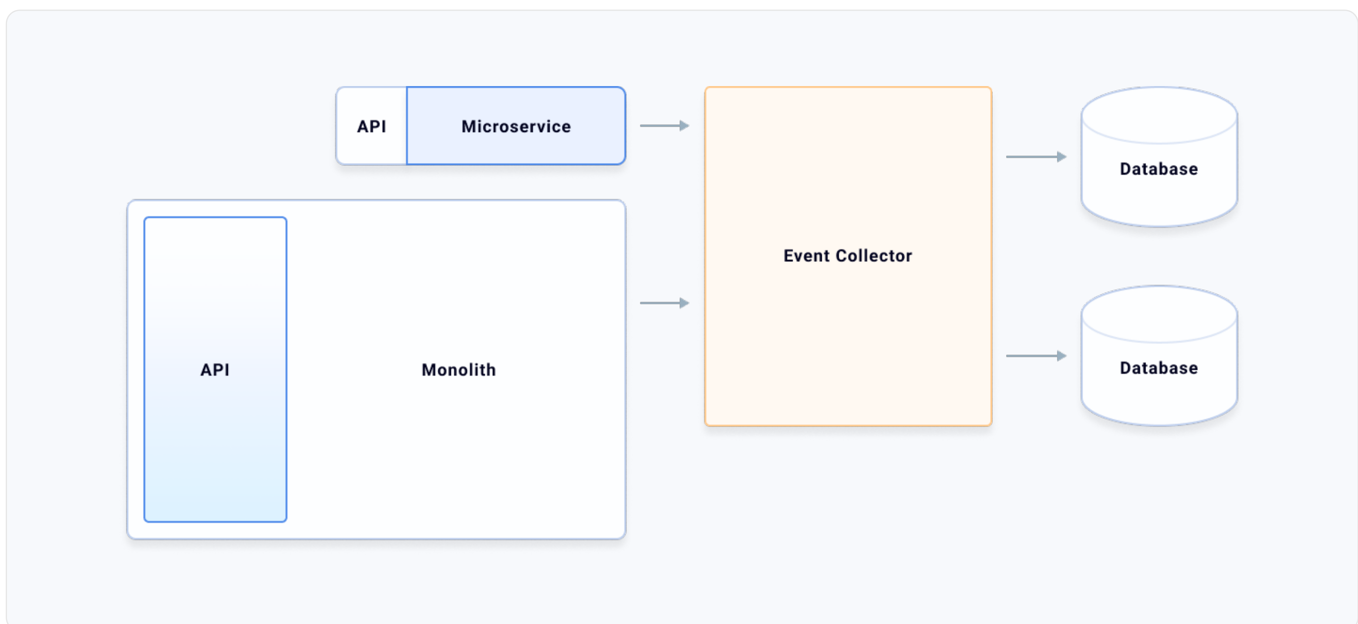


Figure 4: Ensuring data consistency through the use of an event collector which will write to specified data stores

Routing and versioning

Every microservice will be accessible by some sort of API and will be consuming other services via their API, being totally agnostic of their underlying implementation.

Therefore, updates for improved performance and bug fixes can be made as long as there's no change to the API interface (e.g., how requests are being made, their parameters, and the response payload). This ensures that other microservices will still be able to work like nothing ever happened.

Every time a change is made, you don't want to route all the traffic to the new version of the service, as that would be a risky move as any bugs would impact the application. Instead, gradually move traffic over, monitor how the new version behaves, and only when confident of its performance, transition the rest of the traffic. This strategy is also called a canary release and reduces the downtime caused by faulty updates.

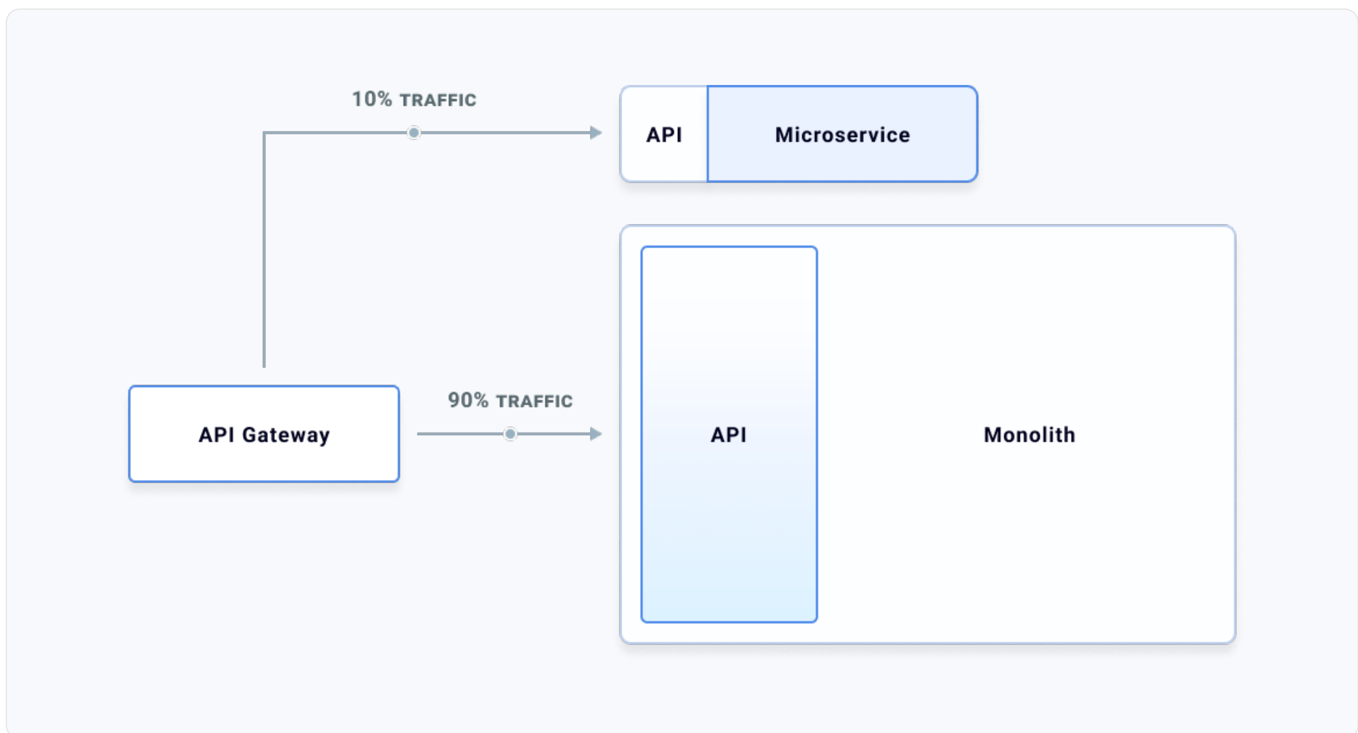


Figure 5: Incremental rerouting of traffic from monolith to microservices through a load balancer

These routing capabilities will both be required when decoupling the monolith and once the decision to upgrade microservices to a different version is made.

These routing capabilities will both be required when decoupling the monolith and once the decision to upgrade microservices to a different version is made.

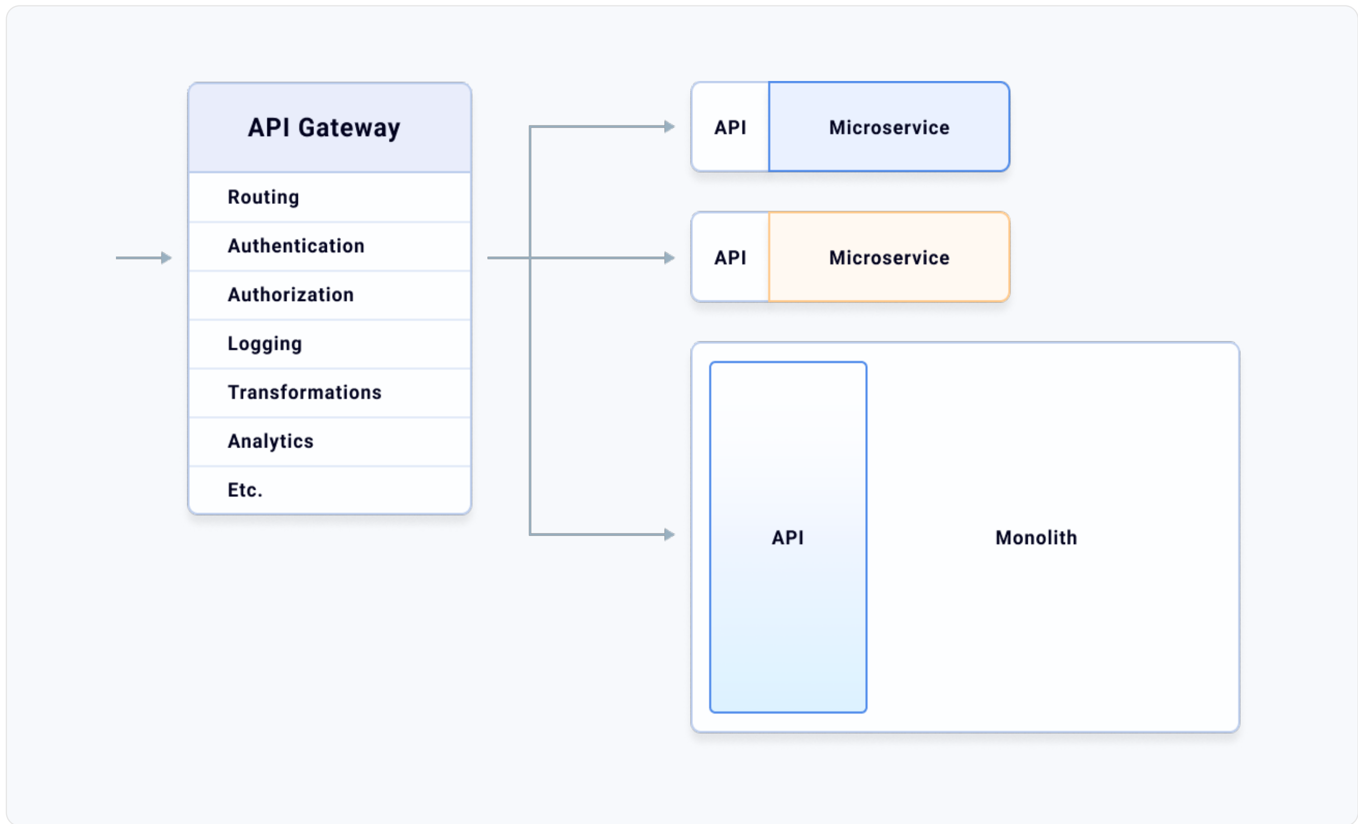
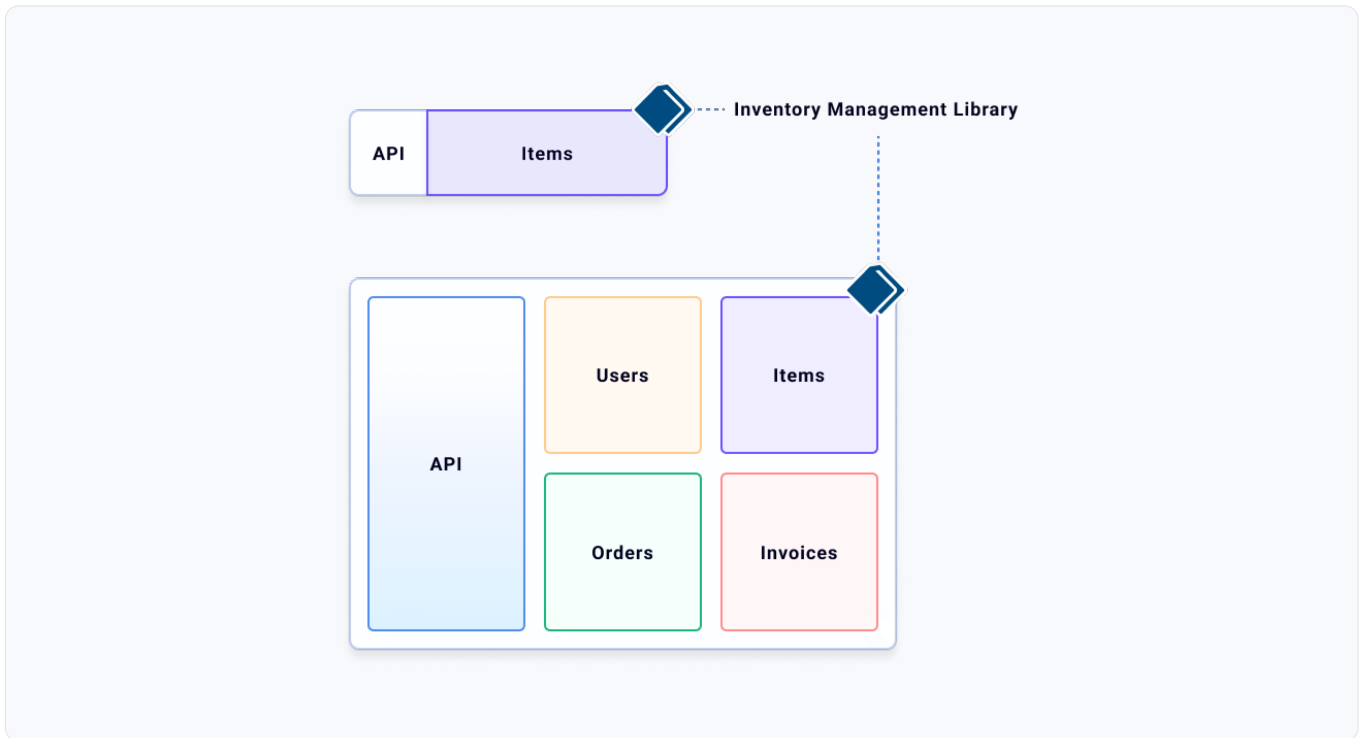


Figure 6: How rerouting through an API gateway will occur as a decoupling from monolith to microservices architecture proceeds

Libraries and security

This approach needs to be realistic and not too ambitious. You don't want to be applying too many changes all at the same time. Depending on the codebase, it may be useful to decouple monolith business logic in third-party libraries that can be then used by both the monolith and the microservice. That also gives the benefit of reflecting any bug fix or performance enhancement on both codebases while in the monolith-microservice limbo.

In general, libraries that deal with specific logic, which eventually will be implemented by only one microservice, don't cause any problems. However, updates to a library used by multiple microservices simultaneously can cause problems, since any update will trigger multiple redeployments across multiple services — which should be avoided as it brings back old memories of the monolith.



With that said, there are so many different use cases on the topic that a one-size-fits-all answer would be hard to give. Generally, whatever roadblock prevents a microservice from being deployed independently or being compartmentalized should be removed in the long run.

Authentication and authorization were concerns handled internally by the monolith, which can also be implemented within a library, or implemented in a separate layer of the architecture like the API gateway. Sometimes a re-design of how authentication and authorization are being handled will be needed to account for scalability as more and more microservices are added.

Security between microservices should be enforced with mutual TLS (mTLS) to make sure that unauthorized clients within the architecture won't be able to consume them.

Logging should be enabled across the board to observe and monitor how the microservices are behaving.

At this point, observability becomes a key requirement to detect anomalies, latency, and high error rates since there are so many moving parts. Therefore, a good rule of thumb will be to configure **health checks for each service, and circuit breakers** that can prevent cascading failures across the infrastructure if too many errors occur.

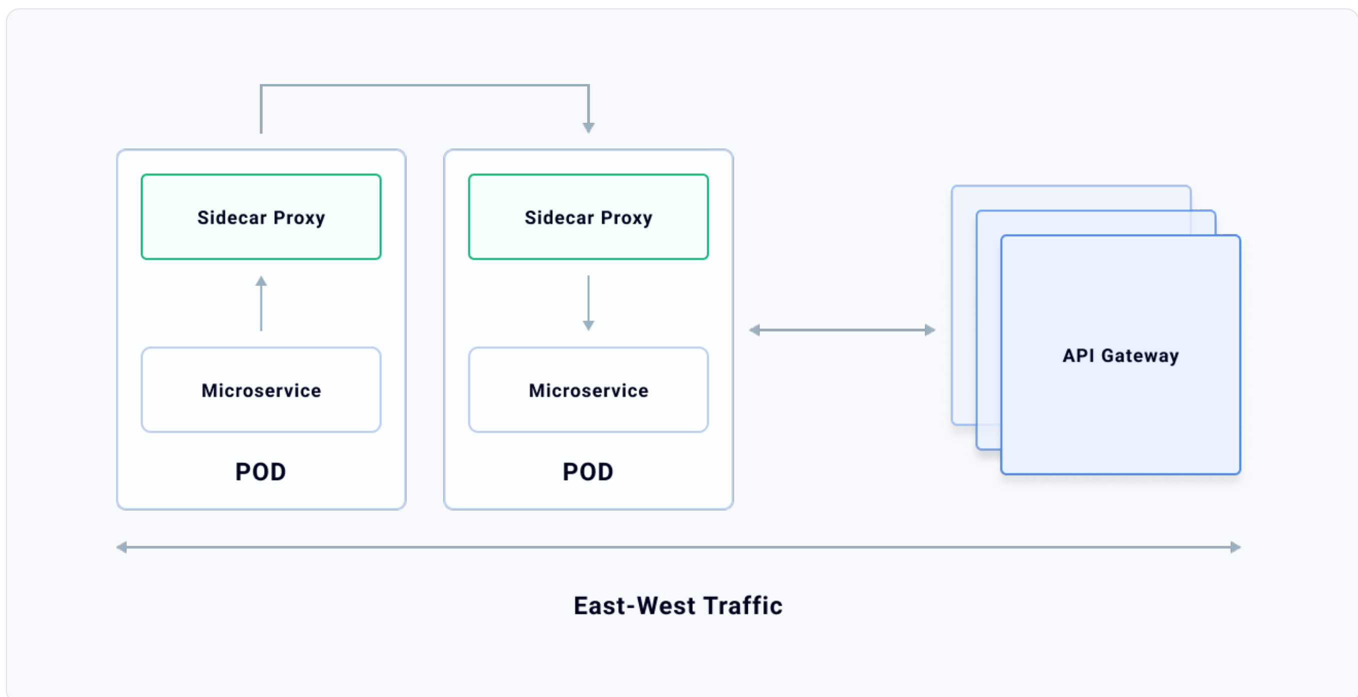
Containers and service mesh

Using containers — like Docker — isn't technically required, although leveraging orchestration tools like Kubernetes, the de facto container orchestration platform, can make life a lot easier if this is the chosen path.

Kubernetes provides many features out of the box, including facilities to scale up and down workloads, service discovery, and networking capabilities to connect microservices. In addition to this, service mesh aims to create microservices-oriented architectures by providing a pattern to perform service-to-service communication and delegates operations like connection management, security, error handling, and observability to a third-party proxy that is usually run in a Kubernetes sidecar alongside our microservice processes.

The service mesh pattern also introduces familiar networking concepts like the control plane for administering our system, and the data plane for processing our requests.

Both planes communicate together so that configuration can be propagated and metrics can be collected.



Conclusion

Transitioning to microservices is a significant engineering investment with equal returns for applications that reach a certain scale.

Therefore, enterprise organizations across industries are either approaching or deploying microservices-based architectures that can help with the pains of a growing codebase and larger teams. It's both a technical and an organizational transition as it requires not only decoupling code but development team structures as well.

Such a radical shift can't be achieved without a long-term plan and preparation tasks that will help with a successful transition, including a good testing strategy. It's also a feat that's not going to happen overnight, and will most likely include transition implementations built along the way, that will have to be removed later on, for example when dealing with legacy database, authentication, or authorization functionality.

But once the hard work is done, and the transition is complete, a microservices-based architecture will **enable the organization to be far more nimble** and enable greater velocity in the evolution of the application.

Powering the API world

Konghq.com

Kong Inc.
contact@konghq.com

77 Geary Street, Suite 630
San Francisco, CA 94108
USA