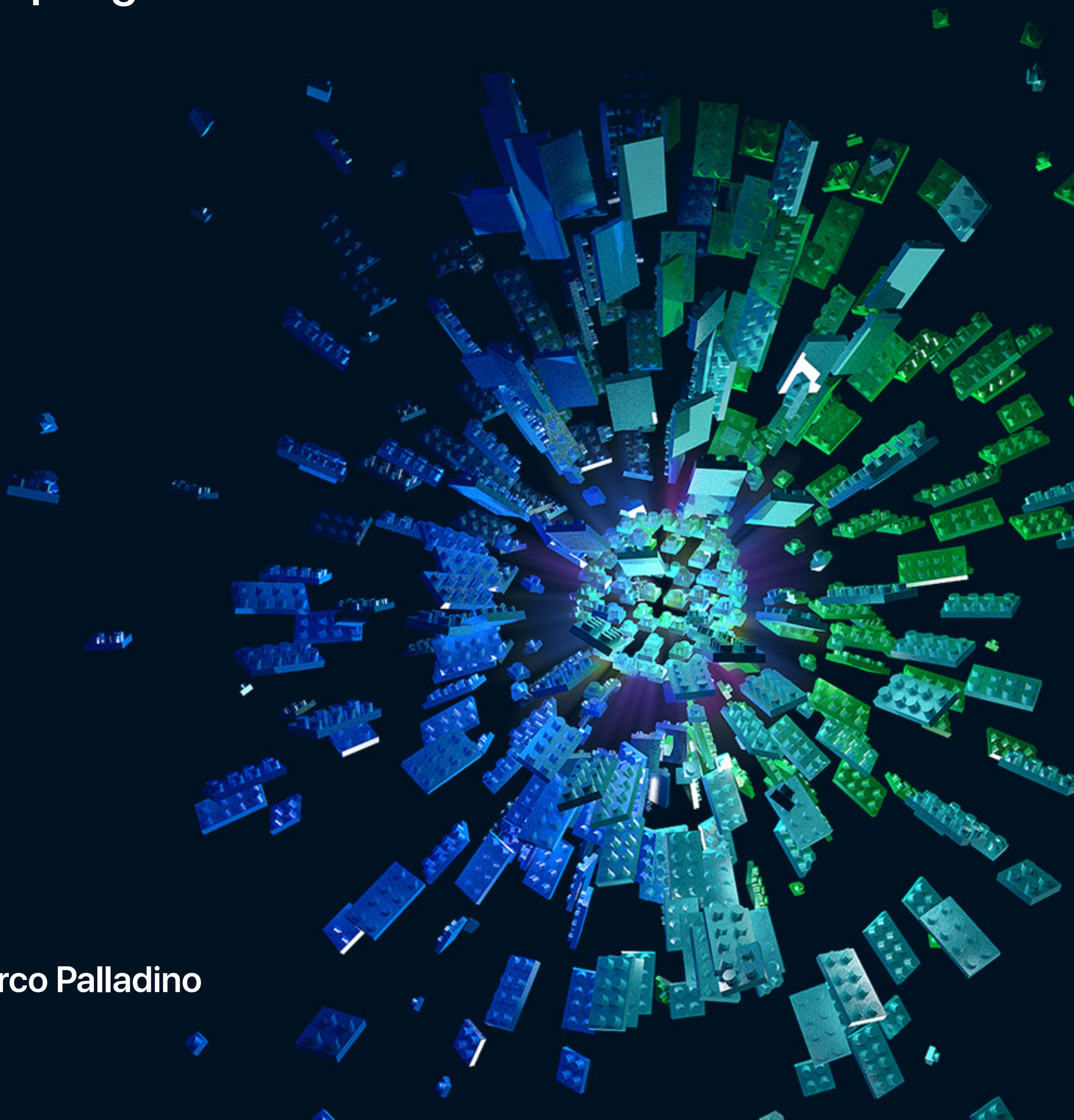
Kong

Blowing Up the Monolith

Adopting a Microservices-Based Architecture



Marco Palladino

In this eBook we'll discuss how to move from a monolithic application architecture to a microservices-based architecture with its pros and cons, and consider the technical aspects and learn common mistakes to avoid. This will include incentives, considerations, strategies for transitioning to microservices, and best practices for the process.

Content

Should you blow up your monolith? 5
Considerations when adopting microservices 6
Two ways for microservice communication 7
  Communications scenarios 8

A technical look into the transition 9
  Boundaries 10
  Testing 10
  Transition strategies 10
    Ice cream scoop strategy 11
    Lego strategy 11
    Nuclear option strategy 12
  The database 12
  Routing and versioning 14
  Libraries and security 15

Containers and service mesh 17
Conclusions 18
References 18

This is an exciting time for software...

We are living in a revolutionary age for software, with massive changes in the way we build, deploy and consume our services. Changes that, as we will learn, are not just technical but organizational as well. New paradigms, patterns and foundations are substantially shaping the industry and we are entering a new era, both technologically and culturally speaking. Open-source software has always been an important contributor to the industry, but in this day and age it's becoming a major player - like never before - of enterprise adoption to new paradigms and architectures.

It's been suggested that decoupling the monolithic application is much like taking a single, cumbersome structure and **re-building it with Legos**. 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 and performs 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. And since you are very 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?

This is a pretty 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 help achieve these goals? The pros and cons of moving to a microservices architecture are 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. Also, microservices enable an application to be far 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 contribute to the consideration of staying with a monolithic architecture. While microservices provide greater flexibility in overall application development and management, it also increases the complexity of the same actions 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 for development, whereas microservices, which can be widely distributed, require greater demand from the network in order for each microservice to communicate and work toward a common objective.
These are really only a handful of the pros and cons to 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 argument that maintaining a monolithic codebase is incredibly inefficient and makes the organizational pursuit of business agility incredibly difficult. This does not mean that transitioning to microservices is going to be smooth.

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 are allocated.When it comes to microservices it can be 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 is 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 is that as you begin to transition to microservices,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 means that a keen eye must be kept on resource allocation, as a side effect this split workload has the potential to cause friction across the two teams, since maintaining a large monolithic application is not as exciting as working on new technologies. This problem can be more easily dealt with in a larger team, where team members can rotate between the two projects. This exposes all team members to new development tasks and gaining new skills - which programmers love, and keeps 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, it's just not going to happen quickly.

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 very specific use-cases.
The main difference between the two patterns is that an asynchronous system won't directly communicate with another microservice, but it 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 - for example - AWS SQS.)

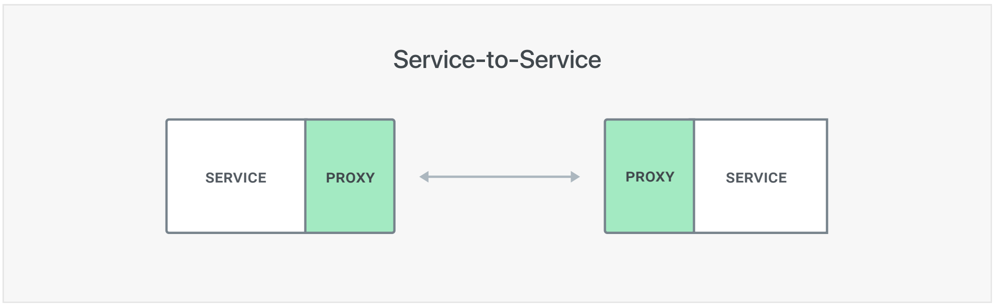


Figure 1: Service to service communication example. Notice the "sidecar" characteristic of this option

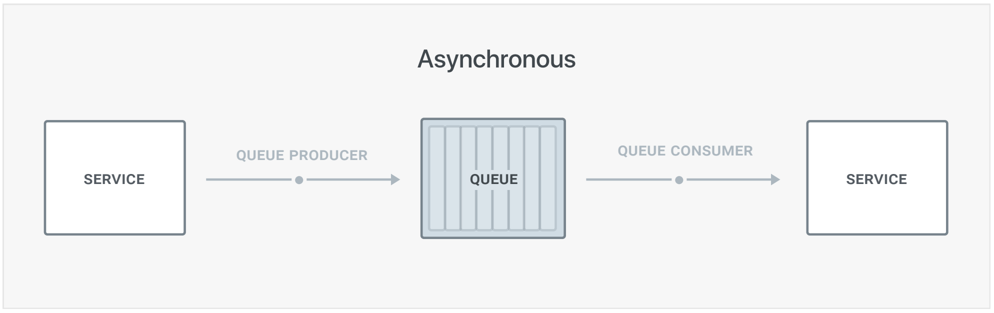


Figure 2: Asynchronous communication from service to service through a proxy

This pattern is very useful every time an immediate response is not required, since it basically implements error handling within the architectural pattern since day one. Therefore it makes a great candidate to propagate eventually consistent data state changes across every microservice avoiding to create inconsistencies.

Communications scenarios

Let's assume for example that you have two microservices, "Orders" and "Invoices", and that every time an order is created 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 being 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 as it will still try to make the request over and over again but eventually - if Invoices is still down - 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 our 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 clientes that require an **immediate** response, or need to aggregate multiple service 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 (ie, retrieve list of users).	Example: Making a request that doesn't require an immediate response (ie "orderCreated" event that triggers an invoice creation by other microservice).

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

Ideally the end state of our microservice-oriented architecture will leverage all of these patterns depending on the use-case, but service-to-service is not the only answer.

A technical look into the transition

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

These preparation tasks are fundamentally important to our success as you dive into the transition and cannot 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 microservice architecture, how big or how small you want them to be, and how they will be communicating with each other. A starting point is to examine those 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, post the transitional phase, all the functionalities that once existed in the monolith are still working in the re-designed architecture. A best practice here is that before attempting any change, a solid and reliable suite of integration and regression tests are put into place for the monolith.

Some of these tests will likely fail along the way, but having well tested functionality will help to track down what is not working as expected.

Transition strategies

There are several strategies to choose from for a transition 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 microservice oriented architecture by "scooping out" different components within the monolith into separate services. This transition is gradual and there will be times were both the monolith and the microservices will be existing at the same time.

Pros: gradually migrating over to microservices with reduced risks without affecting much the uptime and the end user experience.

Cons: it's a gradual 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 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. This option effectively calls for stacking the monolithic and microservices on top of each other in a hybrid architecture.

Pros: not having to do much work on the monolith, fast.

Cons: the monolith will continue having its original problems and new APIs will most likely have to be created to support the microservice-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 arely adopted. The entire monolithic application is rewritten into microservices all at once, perhaps still supporting the old monolith with hotfixes and patches but building every new feature in the new codebase. Surprisingly I have met a few enterprises who decided to go with this strategy, because they assumed that working on the old monolith was not doable and decided to give up working on it.

Pros: allows the organization to re-think how things are done, effectively rewriting the app from scratch.

Cons: it requires rewriting the app from scratch

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 with 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. Instead, since this new architecture requires decoupling the business logic in different services, you will also want to decouple 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 could benefit from high-performance writes to an eventually consistent datastore like Cassandra.

Sometimes some services will indeed use the same underlying database technology, and 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 is that if a microservices - for whatever reason - impacts the database uptime then the other microservices will also be impacted (since they are talking to the same db nodes), This should be avoided since it breaks compartmentalization.

Regardless of setup, consistency of data will be an issue. There is 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 will use a separate store for data. Therefore writes, or reads, made by the monolith won't be visible to the microservice and vice-versa. This is not 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 (ie, Kafka) that will take care of propagating writes to both datastores.

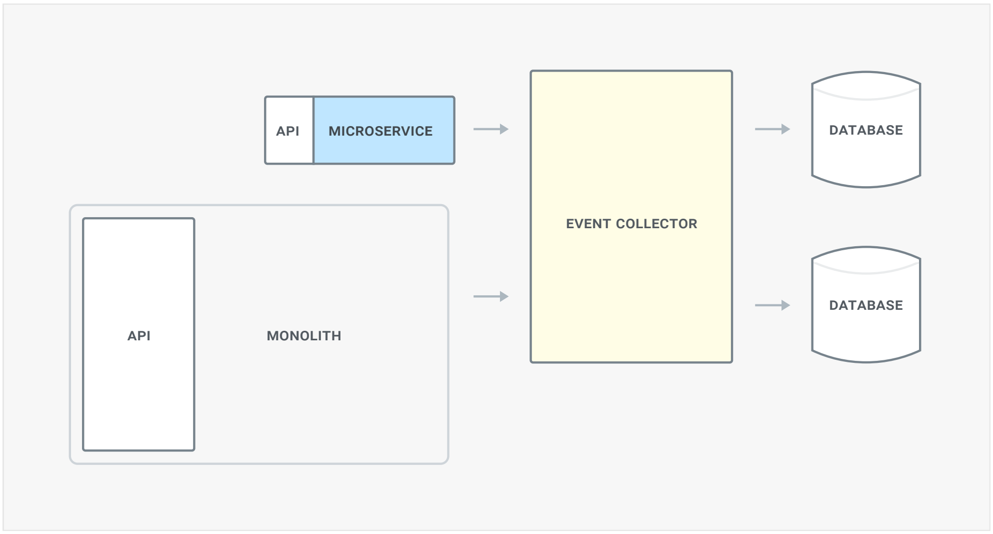


Figure 4: Ensuring data consistency through the use of an event collector which will write to specified data stores.

Routing and versioning

Every microservice will be accessible by some sort of API. As already explained, each microservice will be consuming other services via their API, being totally agnostic of their underlying implementation. Therefore, updates can be made - including fixing bugs or improving performance - and as long as there is no change to the API interface (how requests are being made and their parameters, and the response payload) other microservices will be able to still 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, after all that would be a risky move and if there are any bugs they would impact the application. Instead, gradually move traffic over, monitor how the new version behaves, and only when confident of performance, transition the rest of the traffic. This strategy is also called a canary release, and allows to reduce downtime caused by faulty updates.

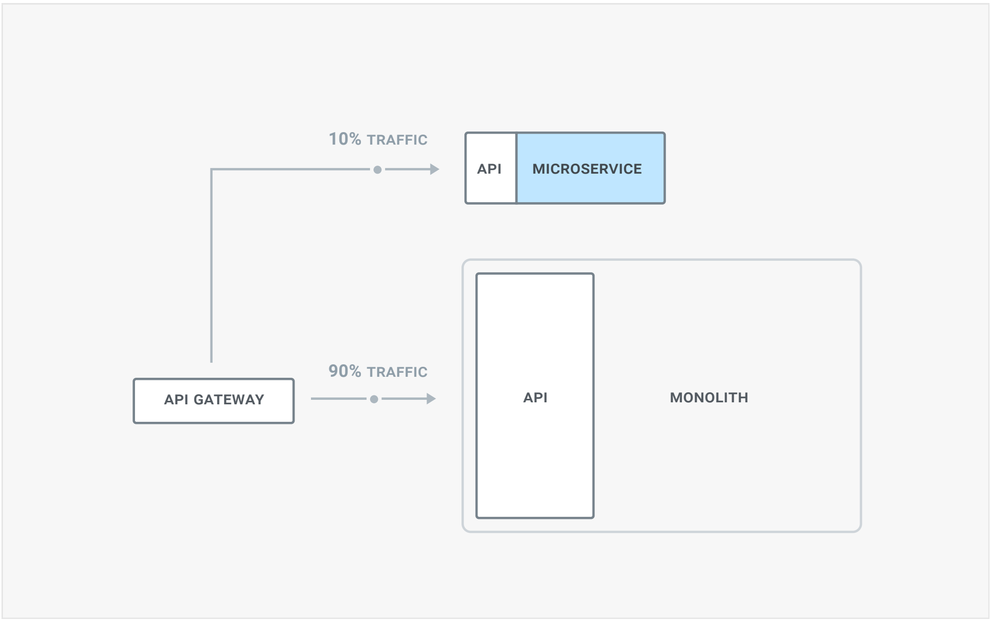


Figure 5: Incremental rerouting of traffic from monolith to microservices through a load balancer.

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

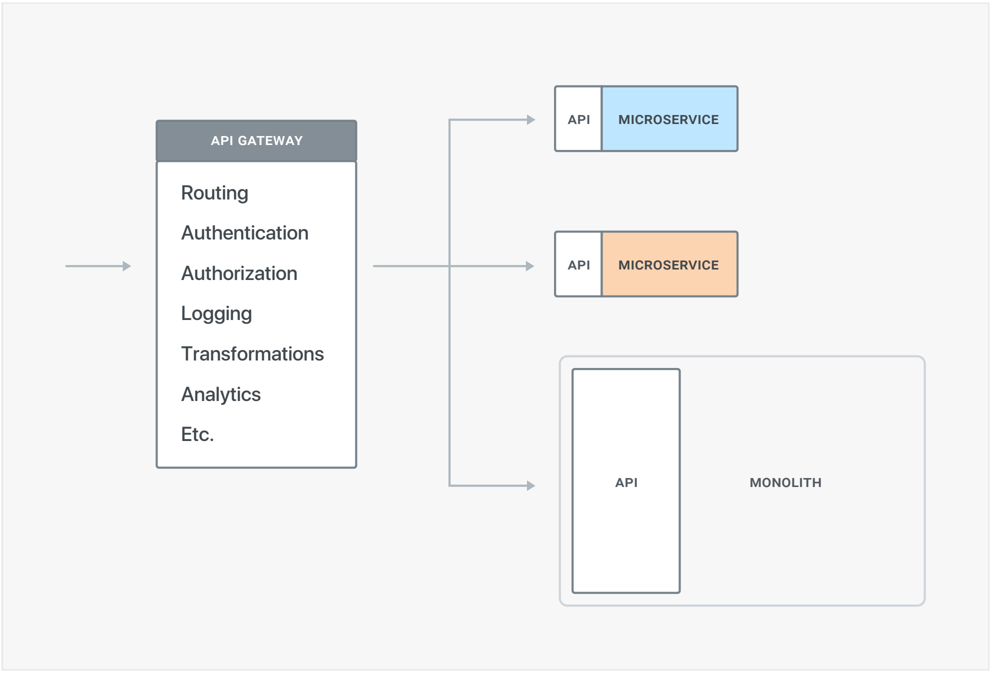


Figure 6:This diagram shows how rerouting through an API gateway will occur as a decoupling from monolith to microservices architecture proceeds.

Libraries and security

This approach needs to be realistic, not too ambitious without 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 monolith/microservice limbo.

In general libraries that deal with specific logic which eventually will be implemented by only one microservice do not cause any problems, since any change in the library will require upgrading only one service, but any update to a library used by more microservices simultaneously will cause problems, since any update will have to trigger multiple re-deployments across multiple services - which should be avoided since 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, but generally speaking 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 there will need to be a re-design of how authentication and authorization are being handled, taking into account scalability as more and more microservices are added.
Security between microservices should be enforced with mutual TLS to make sure that unauthorized clients within the architecture won't be able to consume them. Logging should be enabled all 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 trip and prevent cascade failures across the infrastructure if too many errors occur.

Containers and service mesh

Using containers - like Docker - is not 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 is a new emerging architecture design which aims at helping creating microservice-oriented architectures by providing a pattern to perform service to service communication delegating a few concerns to a third-party proxy - usually run in a Kubernetes sidecar alongside our microservice processes - operations like connection management, security, error handling, and observability.

The Service Mesh pattern also introduces familiar networking concepts like the control-plane, used to administer our system, and the data-plane, which is primarily used for processing our requests. Both planes communicate together so that configuration can be propagated, and metrics can be collected.

Conclusions

Transitioning to microservices is a significant engineering investment with equal significant benefits for applications that reached a certain scale. Therefore Enterprise organizations across every industry are either approaching - or deploying - microservice architectures that can help dealing 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.

Approaching such a radical shift cannot 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 microservice architecture will **enable the organization to be far more nimble** and enable greater velocity in the evolution of the application.

References

[1] https://en.wikipedia.org/wiki/Second-system_effect