Monolith to microservices migration: 10 critical challenges to consider

with in-depth examples

Written by
Emre Baran, Cerbos CEO

cerbos

Table of contents

Chapter 1

Determining service boundaries and decomposing your monolith

Transitioning from a monolithic architecture to microservices is an intricate, time-consuming task. It demands both strategic foresight and meticulous execution.

In this 10-part ebook, we'll guide you through the most common challenges faced during monolith to microservices migration. In this first chapter, we'll start with the pivotal task of decomposing your app and defining service boundaries.

> *"Despite the level of effort and cost to migrate to microservices, the benefits far outweigh the cons, especially because you will never need to replatform again." - Thoughts from Ryan Bartley, ex-eBay, Co-founder at Canopy*

Service boundaries, cohesion, and decomposition

Your first major hurdle when decomposing a monolith application is defining appropriate boundaries for each microservice. Creating the right boundaries will lay a robust foundation for your microservice transition without creating excess complexity.

The process involves breaking down a monolithic application into smaller, independently deployable services that align with business capabilities or domains.

Your goals at this stage are to:

- **Define boundaries between microservices** by understanding how services align with business capabilities.
- **Define relationships between microservices** by balancing cohesion within a service with loose coupling.
    - **Cohesion** refers to the degree to which the components within a service are related and focused on a single responsibility. A highly cohesive service encapsulates related functionalities and data, making it easier to understand, develop, and maintain.

- On the other hand, **loose coupling** ensures that services can be developed, deployed, and scaled independently, without tight dependencies on other services.

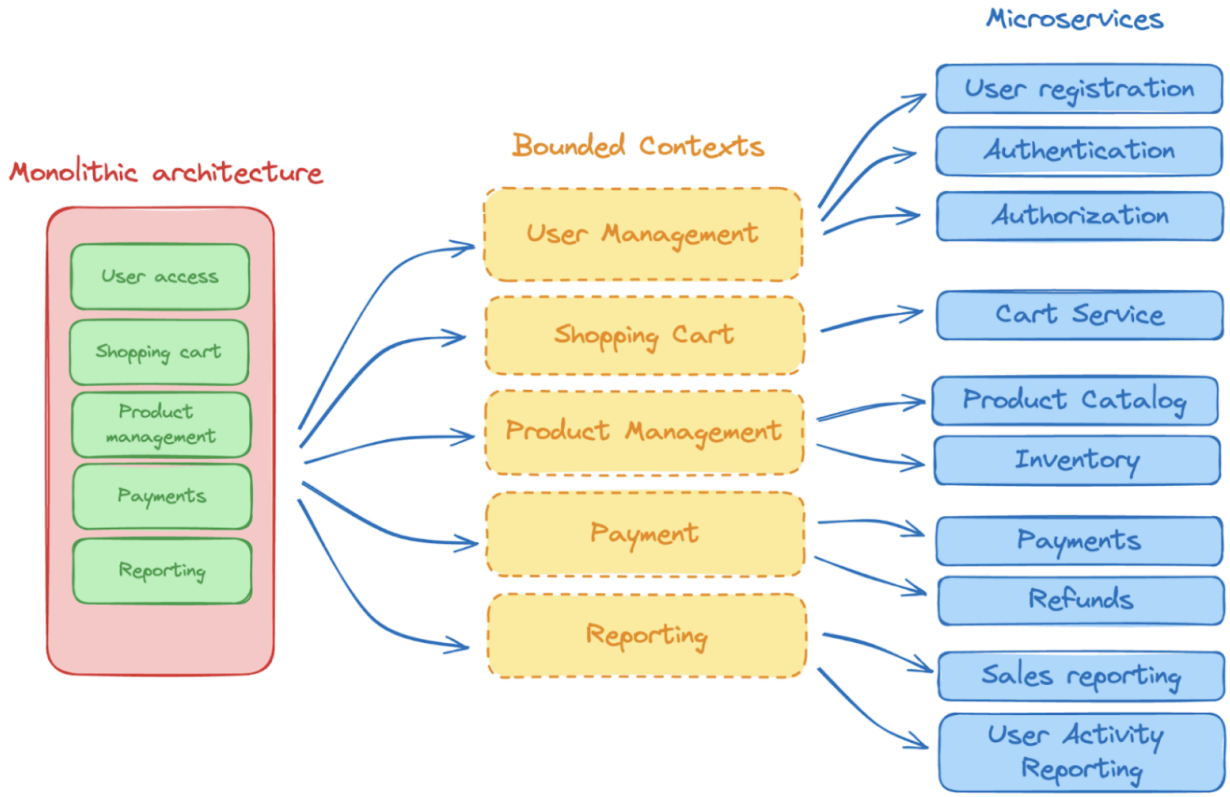
If services are too coarse-grained—with multiple responsibilities and tight coupling—they can become what is commonly known as a "distributed monolith". This effectively negates the benefits of microservices.

Conversely, if services are too fine-grained, with overly narrow responsibilities, it can lead to increased complexity, performance overhead, and difficulties in data consistency and transaction management.

To strike the right balance, you should start with a domain analysis. Once you've identified the core business capabilities, you can gradually extract services based on them. Techniques like event storming, domain storytelling, and use case analysis will also help you on the way.

How to define boundaries between microservices

To identify the right service boundaries, use the following principles: Domain-Driven Design and Single Responsibility Principle.



Domain-Driven Design (DDD)

In DDD there's the concept of bounded context that is helpful in defining boundaries between microservices. Each bounded context represents a specific domain or subdomain within the business that contains its own ubiquitous language, domain model, and set of business rules.

> *"DDD is solving a complex problem by usually breaking the problem into smaller parts and focusing on those smaller problems that are relatively easy. A complex domain may contain sub domains. And some of sub domains can combine and group with each other for common rules and responsibilities", explains Mehmet Ozkaya, ex-Ericsson Software Solutions Architect and Udemy course creator.*

During decomposition, it's helpful to think of each service as a bounded context. Any microservice that responds to that bounded context becomes part of that service's

domain. So, by mapping your services to bounded contexts, you ensure each service has a clear and focused responsibility aligned with the business domain.

Single Responsibility Principle (SRP)

SRP is the first of five principles in the SOLID approach to object-oriented design. SRP states that a class or module should have only one reason to change.

When you apply SRP to monolith decomposition, it guides the process to constrain each service to a single, well-defined responsibility. All related data and functionalities are then encapsulated within that domain.

Finding the right balance

Decomposing a monolith application into microservices is not a simple or straightforward process. Doing it well requires an iterative approach, with continuous refactoring and evolution.

Here is some good advice from Eldad Palachi, Principal Architect at vFunction on how to get started:

> *"Start with functionality that is already somewhat decoupled from the monolith, does not require changes to client-facing applications, and does not use a data store. Convert this to a microservice. This helps the team upskill and set up the minimum DevOps architecture to build and deploy the microservice."*

Now, let's take a look at how companies decompose a monolith in real life.

How Netflix decomposed their monolith into microservices

Through an iterative process, Netflix successfully decomposed their monolithic application into microservices using DDD and SRP principles.

Utilizing bounded contexts

First, Netflix identified key bounded contexts within their business domain, like user management, content catalog, recommendations, and playback. Then, each bounded context was mapped to a set of microservices responsible for that specific domain.

For example, the user management bounded context was decomposed into services like:

- user authentication
- user profile
- user preferences

These services encapsulated the related functionalities and data, ensuring high cohesion within each service.

Implementing SRP

Netflix also applied SRP to their microservices design. So, each service had a clear and focused responsibility, such as handling user authentication, managing the content catalog, or providing personalized recommendations. This approach allowed Netflix to develop, deploy, and scale services independently, which promoted loose coupling and greater flexibility.

Optimizing with patterns

To handle complex business transactions and ensure data consistency, Netflix used patterns like event sourcing and CQRS (Command Query Responsibility Segregation).

Event sourcing allowed them to capture all changes to a service's state as a sequence of events. This provided a complete audit trail and enabled event-driven architectures. CQRS separated the read and write responsibilities of a service, which optimized performance and scalability.

Gradually improving architecture

Netflix gained insights into service performance after decomposition through monitoring, logging, and tracing. Using this data, they identified improvement opportunities and continually refined their service boundaries. Over time, they optimized their microservices to maintain alignment with business domains.

How Netflix benefited from decomposition

Applying domain-driven design, SRP, and continuous refactoring, Netflix decomposed its monolith into cohesive and loosely coupled microservices. This allowed them to scale their platform, accelerate development, and deliver a seamless streaming experience.

Chapter 2

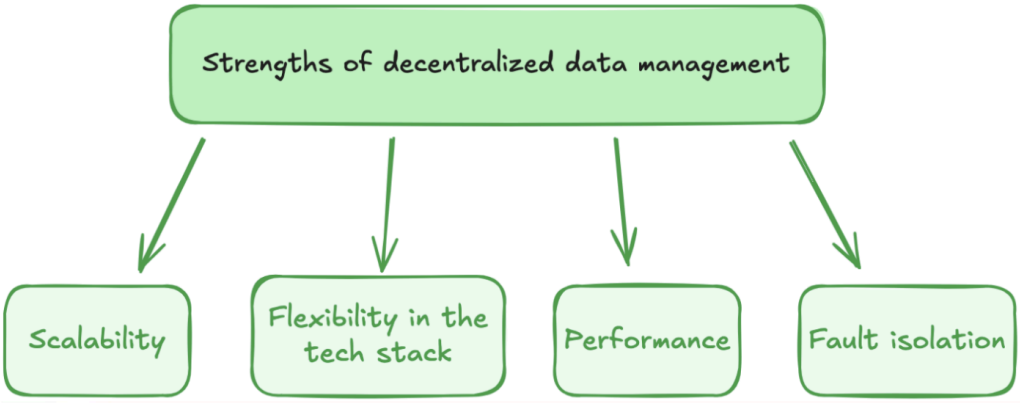
Decentralizing data management

Data management and consistency

Unlike monolithic applications—where data is stored in a single, centralized database—microservices typically take a decentralized data management approach. Often, each service will have its own, dedicated database or data store, optimized for its specific requirements. Decentralized data management brings both strengths and challenges with it.

In this chapter, we will cover the strengths and challenges you should know before you migrate to a decentralized data storage system. Then, we'll suggest patterns & techniques you can use to overcome those challenges. Let's start with the good part.

Strengths of decentralized data management



1. Scalability

Each microservice can scale independently based on its specific load and performance requirements. Decentralized data architecture allows more efficient resource utilization and better handling of varying traffic patterns across different parts of an app.

2. Flexibility in the tech stack

Teams are free to plug-and-play their preferred data storage solutions (e.g., SQL, NoSQL, in-memory databases) so they can find the solution that best fits the service's needs. This means each team can tailor its data storage strategy for optimal performance of each service.

3. Performance

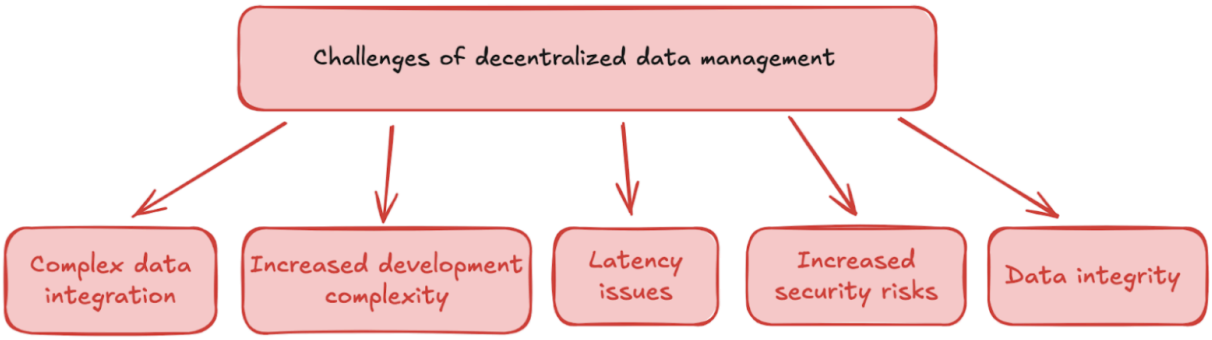
When your team isn't stuck with one option, they can increase the speed of query execution and data retrieval by tailoring data storage technology to each service's unique access patterns and data types.

4. Fault isolation

If one service encounters an issue, it does not necessarily impact the entire system. This isolation enhances system reliability and makes it easier to manage expectations and maintain uptime. So you get a better resilience against the system failures overall.

Challenges of decentralized data management

Before you embrace decentralized data management, you should be aware that there are difficult challenges to overcome, including increased complexity in development and data integration and issues with integrity and latency.



1. Complex data integration
Integrating data from multiple decentralized sources can be complex and time-consuming. With different nodes running different storage solutions, data interoperability and compatibility become critical considerations to ensure seamless data exchange.

2. Increased development complexity
Managing multiple databases requires sophisticated strategies for data replication, synchronization, and consistency. This can make the system harder to develop, test, and maintain.

3. Latency issues
Network communication between microservices can increase latency, especially when microservices need to access data from multiple sources.

4. Increased security risks
Decentralized data requires a robust security system that maintains security over multiple nodes. Implementing encryption, access controls, and authentication mechanisms across the system is essential to ensure the safety of your data.

5. Data integrity
Maintaining data integrity requires thoughtful planning to ensure that business rules and validations are consistently applied across all decentralized services.

If you want to dive deeper into the potential data headaches, we highly recommend checking Chad Sanderson, CEO at Gable.ai, ex Microsoft. Substack:

> *"In the traditional on-premise Data Warehouse, an experienced data architect was responsible for defining the source of truth in a monolithic environment. While slow and somewhat clunky to use, it fulfilled the primary role of a data ecosystem. Smart, hard-working data professionals maintained*

an integration layer to ensure downstream consumers could reliably use a set of vetted, trustworthy data sets.

In the world of microservices, however, there is no truth with a capital 'T.' Each team is independently responsible for managing their data product which can and often will contain duplicative information. There is nothing that prevents the same data from being defined by multiple microservices in different ways, from being called different names, or from being changed at any time for any reason without the downstream consumers being told about it."

Patterns and techniques to address data management challenges

While there are no plug-and-play solutions to the above issues, there are patterns and techniques companies have successfully used to mitigate their impact.

1. Eventual consistency accepts that temporary inconsistencies between services may occur but guarantees that the system will eventually reach a consistent state. By allowing this trade-off the system can achieve higher availability and performance in scenarios where strong consistency is not essential.

2. The Saga pattern manages distributed transactions through a sequence of local transactions, ensuring eventual consistency. Each service involved in a transaction performs its local transaction and publishes an event to trigger the next step.

3. Event sourcing captures all changes to an application's state as a series of events. Instead of storing the current state, events are persisted in an event store. Services can subscribe to these events and reconstruct their state by replaying the event stream. Event sourcing ensures data consistency, provides a comprehensive audit trail, and supports eventual consistency.

4. Domain-driven design (DDD) helps define clear boundaries and responsibilities for each service. DDD (which we covered in a previous chapter) ensures that data ownership and consistency are maintained within the service boundaries by better aligning the business domain and the microservices architecture.

5. Command query responsibility segregation (CQRS) separates the read and write operations of a service into different models, optimizing for different data access patterns and scalability requirements. When read and write operations are separated, the system can more efficiently handle queries and commands, improving the overall system performance and scalability.

Now that we've covered the concepts, let's take a look at how all of that plays out in the real world. As in the previous chapter, we'll dive deeper into a case study. This time, we'll see how Uber dealt with these challenges.

How Uber ensured consistency and speed across millions of requests

Uber fields millions of simultaneous rides per minute. Each of those rides accesses driver requests, payment services, user drop-off data, and more. So when they scaled their microservices architecture, they faced significant data management and consistency challenges.

So, how did Uber scale without losing control of their data? To ensure data consistency, platform integrity, and provide a seamless user experience while scaling, they had to completely rethink how they stored and accessed data.

Coordinating interactions with the saga pattern

Remember the Saga pattern? Uber chose it to coordinate interactions between services, including user, service, and payment services. So when a user requests a

ride, each service performs its local transaction and publishes events to trigger the next step in the saga. If any step fails, compensating actions roll back the previous steps to maintain data consistency.

Capturing all actions with event sourcing

Event sourcing helped Uber capture all changes to their system as a sequence of events. Each service publishes domain events whenever there is a state change, like a ride request, or drop-off. Other services consume these events and update their own state to reflect the event. With event sourcing, the team had a complete history of all actions, which they could use for data auditing, debugging, and analysis.

Maintaining data consistency with DDD

Uber applied DDD principles to ensure data integrity and enforce business rules. They defined clear bounded contexts for each domain, such as user management, ride management, and payment processing. Then, they gave each bounded context its own set of services, data models, and business rules. These strong borders ensured data consistency within the context boundaries.

Using CQRS to scale systems independently

With this new architecture, Uber needed a way to take advantage of the decoupled scaling microservices allow. They chose Command Query Responsibility Segregation (CQRS) which allowed them to optimize data access patterns and separate read and write responsibilities.

This allowed Uber to scale each service independently based on the specific requirements of each operation. As a result, the Uber team was able to drive performance improvements where it mattered and more easily maintain their microservices architecture.

Visibility in a distributed system

With their app distributed across various systems, Uber needed a way to see if their systems were working together as designed. So, they invested in monitoring, logging, and tracing capabilities to gain visibility into their distributed system.

They used:

- Jaeger for distributed tracing
- Apache Kafka for event streaming
- Apache Cassandra for high-performance data storage

These tools helped their engineering team identify and troubleshoot data consistency issues, ensure reliable event delivery, and maintain the overall health of their microservices ecosystem.

Two-pronged plan

Decentralized data management challenges do pose a risk to those who aren't aware of them. But with proper planning and tools, they can be handled, which is exactly what Uber did. They applied successful patterns and techniques and backed them up with robust monitoring and data management technologies.

With those techniques in place, Uber maintained data consistency and integrity while scaling their large-scale microservices architecture, enabling them to deliver a reliable ride-hailing experience.

Chapter 3

Establishing inter-service communication patterns

Picking the right communication pattern for your microservices

Without seamless communication between microservices, the functionality of the app and the experience of the end-user suffers. To make sure you maintain a good user experience you need to tailor inter-service communication to match the demands of your system and make sure it can handle failure scenarios.

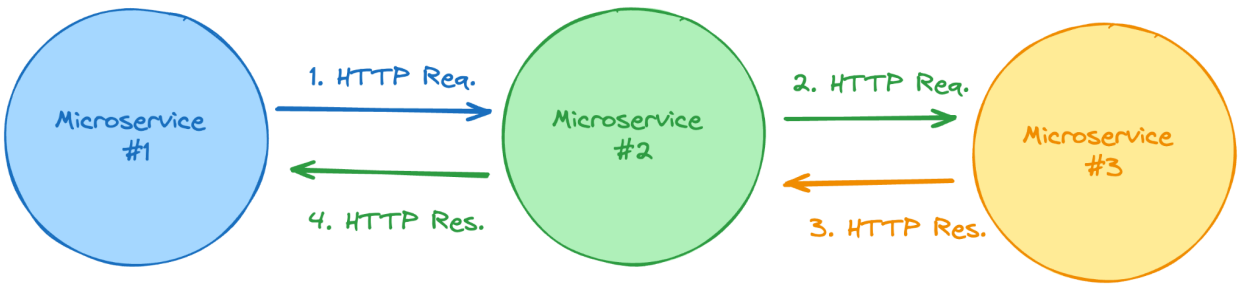
The first step to tailoring communication in your system is finding the right communication patterns.

Synchronous communication

Synchronous communication patterns, such as REST or gRPC, are simple request-response interactions. A service sends a request and then waits for a response from another service.

- **REST (Representational State Transfer)** is a stateless protocol often used over HTTP. It's highly scalable and widely supported.
- **gRPC (Google Remote Procedure Call)** uses HTTP/2 for transport, providing features like bi-directional streaming and efficient binary serialization.

When multiple microservices synchronously communicate (like in the diagram below), they end up executing the interactions in series. This means the final response must come after all other steps have finished.



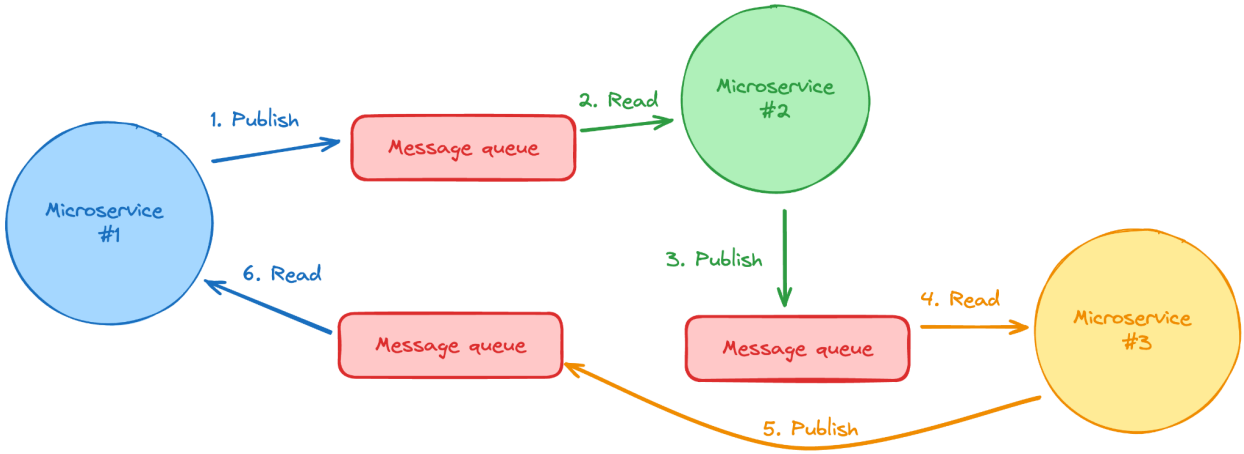
This approach ensures consistency, but can also create a performance bottleneck if not managed properly. It's also important to note that synchronous communication creates tight coupling between all involved services. This pattern is ideal for scenarios that require immediate feedback, including simple and direct interactions.

But, many microservices interactions require complex interactions between multiple microservices. That requires a more complex communication pattern.

Asynchronous communication

Asynchronous communication patterns involve services interacting with each other without waiting for an immediate response. Common asynchronous communication patterns include message queues and streaming platforms.

- **Message queues**, like RabbitMQ and Apache ActiveMQ, operate much like a message board. Instead of waiting in line to give a message, one microservice simply leaves a message in a queue. These messages are then processed by the receiving services when it is able.
- **Streaming platforms**, like Apache Kafka, allow microservices to publish and subscribe to continuous flows of data, while at the same time providing a scalable and highly available service.



When multiple microservices use a queue to asynchronously communicate (like in the above diagram), each is free to leave a message without waiting for an answer. The result is non-linear communication that does not require each service to wait before executing.

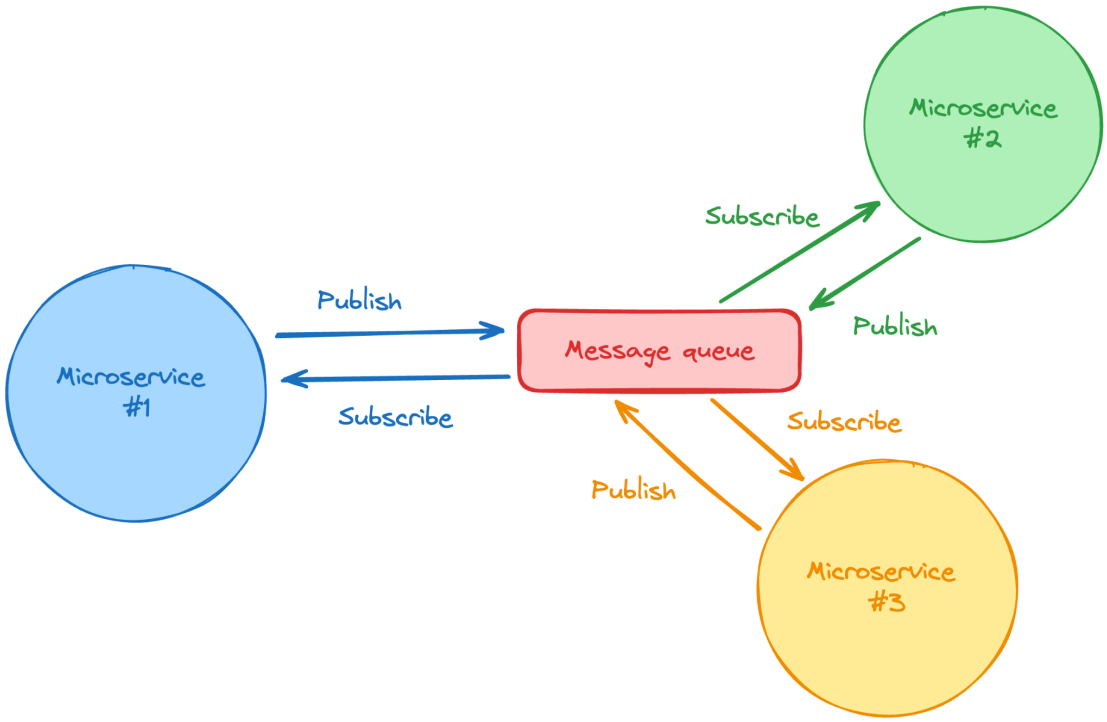
This pattern decouples services, enhancing scalability and fault tolerance. So, services can work independently from each other, mitigating potential bottlenecks. It does, however, introduce more complexity than synchronous communications.

Event-driven architecture

Event-driven architectures extend asynchronous communication by focusing on events, which are significant state changes associated with a point in time. Services publish events, then other services consume (or subscribe to) these "event streams" as needed.

- **Event publishing** - When specific actions occur, such as changes in data or user actions services generate events.
- **Event consumption** - To keep up with appropriate events, services subscribe to and process published events, allowing them to react to changes in real time.

When multiple microservices communicate through event-driven architecture (like in the diagram below), each service pulls data from and writes data to a central, shared message queue. This provides a flexible, scalable communication model with loose coupling.



Once you've chosen the right communication pattern for your microservices, you need to settle on a communication protocol that fits the pattern.

Protocols and their roles

Protocols define the rules for data exchange and ensure interoperability between services. The most common ones used for inter-service communication are:

Synchronous communication protocols

- **HTTP/HTTPS** - Commonly used with REST for straightforward web communication.
- **Protobuf** (Protocol Buffers) - Used with gRPC for efficient binary serialization.

Asynchronous communication protocols

- **AMQP** (Advanced Message Queuing Protocol) - Often used with message queues for reliable message delivery.

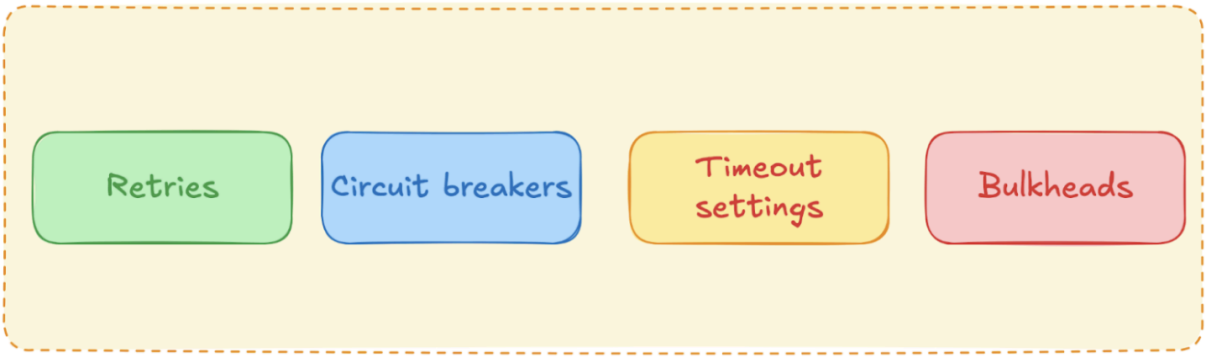
Event driven protocols

- **Pub/sub** - Used as a simple communication protocol for microservices.

Handling failure scenarios in communication

Simply picking the right communication isn't enough to ensure robust inter-service communication. Without proper fault tolerance, communication between services is a weak point in the system, leading to cascading failures.

The following four strategies will help your engineering team build resilience into your inter-service communication.



Retries

Retries is a simple strategy that automatically attempts to resend failed requests after a brief delay. This helps mitigate transient issues, such as temporary network glitches or brief service disruptions.

How it works

- When a request fails, the service waits for a predefined interval before retrying the request.
- The number of retry attempts and the delay between attempts can be configured based on the specific use case.

- Exponential backoff can be used to gradually increase the delay between retries, reducing the load on the failing service.

When dialed in, retries smooth out temporary disruptions, so users aren't aware of these small faults. It also frees your team from constantly having to manually intervene with requests by increasing the chance of success for every request.

Circuit breakers

Circuit breakers monitor the health of services and temporarily degrade or disable communication with services that are experiencing failures. This prevents one service outage from causing a chain reaction of cascading failures throughout the system.

How it works

- The circuit breaker has three states: closed, open, and half-open.
    - When the service is healthy the breaker stays **closed** so that requests flow normally.
    - When a failure is detected in a service, the breaker **opens**. This blocks requests, allowing the service time to recover.
    - After stipulations are met, the breaker will **half-open**, letting a limited number of requests through to test if the service has recovered.
- If the service responds successfully during the half-open state, the circuit breaker closes, and normal traffic resumes. If failures continue, the circuit breaker reopens.

Excessive pressure can quickly overwhelm a service, leading to cascading faults. Circuit breakers prevent services from being overwhelmed, allowing them to recover by relieving load pressure. And, if they fail, circuit breakers increase the stability of a system by isolating these failures and rerouting traffic.

Timeout settings

Timeout settings define the maximum interval a service will wait for a response from another service before considering the request failed. With proper timeout configuration, you can mitigate prolonged delays as services wait too long for a response and the resource exhaustion that comes from that.

How it works

- Each service call is assigned a timeout period.
- If the response is not received within the timeout period, the request is aborted, and an error is returned.

Timeout prevents services from waiting indefinitely for responses, allowing the system to keep running without the drag of open requests piling up. It also helps you identify and handle slow services promptly.

Bulkheads

Bulkheads partition a system into isolated sections to prevent failures in one part from affecting the larger system. This is similar to compartmentalization in ship design, where individual sections can be sealed off to contain damage.

How it works

- Resources (such as threads, memory, or database connections) are divided into separate pools, each of which is dedicated to specific services or functions.
- This allows the majority of the system to continue to operate normally even if one pool becomes exhausted due to a failure.

By partitioning your system into well-designed bulkheads, you limit the impact of failures, ensuring the rest of your critical services remain available even when others are experiencing failures.

With the right communication patterns and robust strategies to handle failure systems, even the largest apps, like Spotify, can build in resilience.

How Spotify built resilience with an event-driven architecture

When Spotify transitioned to a microservices-based system, they adopted an event-driven architecture using Apache Kafka for inter-service communication. This gave Spotify the ability to build loose coupling, scalability, and fault tolerance into their microservices ecosystem.

Asynchronous, event-driven architecture

Spotify chose Kafka so their services could both publish and consume events asynchronously. This decoupled the services from each other, allowing each to evolve independently so Spotify could scale services as needed. Kafka's fault-tolerant and scalable design ensured that events were reliably delivered and processed, even in the face of failures or high loads.

To simplify integration, Spotify developed their own tooling and frameworks. They established guidelines and best practices for event design, schema evolution, and error handling to ensure consistent and reliable communication across services. This allowed engineering teams to build based on business logic rather than low-level communication details.

Building in resilience

Additionally, Spotify implemented advanced patterns like Event Sourcing and CQRS (Command Query Responsibility Segregation) to enhance the resilience and scalability of their system. While event sourcing allowed them to capture all state changes as a sequence of events, providing an audit trail and enabling event replay.

CQRS separated read and write operations, optimizing for different access patterns and scalability requirements.

Scaling and evolving a popular app

As we see, Spotify successfully transitioned to a microservices-based system that could scale and evolve independently, while maintaining loose coupling and fault tolerance. This approach allowed them to handle their app's massive scale and complexity – a topic we're going to touch on a little later in the ebook.

Chapter 4

Service discovery and load balancing

Static service discovery mechanisms, such as hardcoding service locations or using load balancers, fail when tasked with navigating the complexities of a microservices architecture.

Service discovery mechanisms allow microservices to locate and connect to the appropriate instances of other services that they need to communicate with. When tasked with locating and connecting instances across microservices, these systems become overwhelmingly complex and difficult to manage. And because they are built to communicate within a monolithic environment, they encourage tight coupling across your architecture creating a 'distributed monolith', as discussed in the first chapter of this series.

Dynamic service discovery, like service registries and service meshes, are better solutions for microservices architecture. They allow your distributed system to stay flexible through loose coupling because locations don't need to be hardcoded for services to discover each other. They also built in resilience with the ability to reroute traffic over failed instances.

We're going to dive into how each of the above service discoveries work so you can decide on the best option for your architecture.

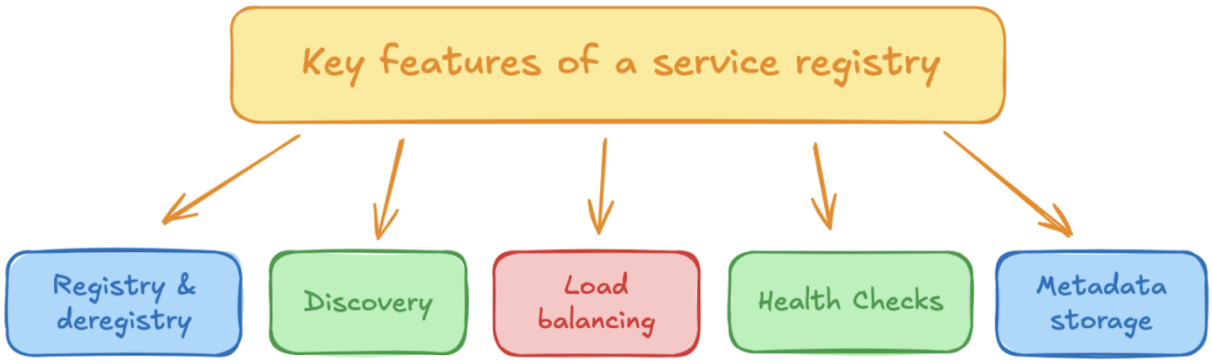
What is a service registry?

A service registry is a centralized database that maintains the instances, and network locations of all available services, and then makes them available for application-level communication. Services first register with the database to signal they are available and then query the registry to find any instances they need.

How does a service registry work?

1. Services register themselves when they come online, making them discoverable.
2. When a service needs to discover and connect with an appropriate instance, it queries the registry to connect with the right instance.
3. As services come and go, the registry is updated to reflect the changing environment to ensure a client receives information that connects it with healthy instances.

Key features of a service registry



Registry and deregistry

When a service instance starts, it registers its network location (IP address and port) with the service registry. When a service instance shuts down or becomes unhealthy (i.e. non-responsive), it deregisters itself from the service registry so that other services don't attempt to connect to it.

Discovery

Service registries allow for dynamic discovery of service endpoints without hardcoding network locations. Instead of having a set network location where

instances can be found, services query the registry to find the instances of another service.

Load balancing

Most service registries work with load balancers. They distribute traffic evenly by splitting traffic across multiple service instances. This helps enhance both performance and fault tolerance.

Health Checks

Many service registries include built-in health check mechanisms, typically a health check URL. The registry then periodically verifies the health of registered service instances by querying the URL. If an unhealthy instance is found, it's deregistered.

Metadata storage

Aside from registering services for discovery, registries can also store metadata about registered services. This data can include version numbers, configuration settings, and custom attributes which it uses to route requests based on specific criteria or provide detailed information for monitoring and debugging.

Commonly, service registries can be implemented using network-accessible key-value stores. Now let's talk about examples of service registries:

- A distributed coordination service, Apache Zookeeper provides robust features for service discovery and configuration management.
- Consul is a popular HashiCorp tool which offers service discovery, health checking, and a distributed key-value store.
- Developed by CoreOS (now part of Red Hat), Etcd is a distributed key-value store. It's often used for storing configuration data and service discovery information.

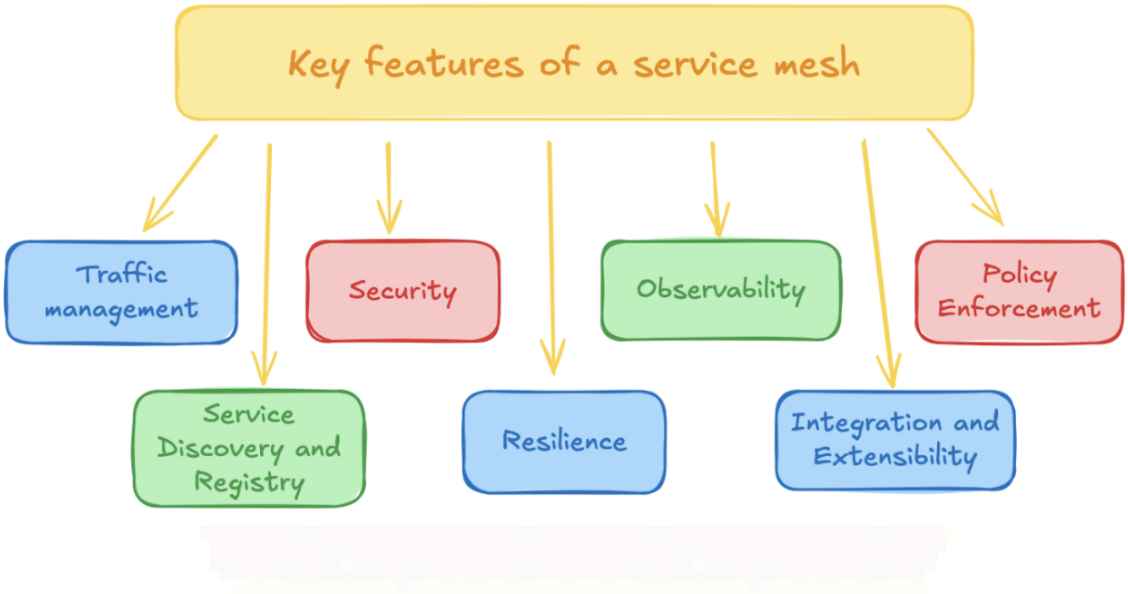
What is a service mesh?

A service mesh takes service discovery a step further by implementing a dedicated infrastructure layer for service-to-service communication. This frees individual services from having to implement the complex functionalities of service discovery.

How does a service mesh work?

Typically, service meshes are implemented as a network of lightweight proxies (e.g. sidecars) deployed alongside each service instance. These proxies handle service discovery, load balancing, encryption, and other cross-cutting concerns.

Key features of a service mesh



Traffic management

A service mesh can help manage traffic through your distributed systems through a combination of load balancing, traffic shaping and splitting, and routing. It prevents bottlenecks in the system by distributing incoming requests across multiple service instances.

Service meshes also control traffic between services through rate limiting and throttling to shape traffic and smooth out spikes. In the same way, service meshes can split traffic between service versions, allowing for canary releases, A/B testing, and gradual rollouts of new features. Finally, service meshes typically offer advanced routing capabilities including request-based routing, header-based routing, and URL path-based routing. This allows for granular control over how traffic is directed between services.

Security

With various measures to provide enhanced security in microservices architectures, service meshes help your team build a secure system. Mutual TLS (mTLS) secures communication between services by encrypting traffic between endpoints. They also support authentication and authorization by offering fine-grained control policies.

Properly set up, these authorizations restrict services so only those authorized can access specific services and endpoints. The ability to manage service identities and certificates simplifies the implementation of secure service-to-service communication.

Observability

Typically, meshes include the automatic collection of metrics related to service communication, including the four golden signals (traffic, latencies, saturation, and error rates). This gives your team insights into the health and performance of services. They also integrate with distributed tracing systems (which we'll cover in our next chapter) and gather and aggregate logs of service interactions. This increased visibility gives developers more visibility in the system so they can effectively monitor and troubleshoot systems to enhance performance.

Resilience

A service mesh offers greater flexibility than static systems, building resilience into your architecture. One of the key tools meshes have is the ability to implement circuit breakers. Teams are also able to configure automatic retries to set timeouts to handle transient failures and ensure timely responses. A mesh also allows your team

to control the rate of incoming requests to prevent services from getting overwhelmed with requests.

Policy enforcement

Service policies are built into a service mesh, allowing your team to define and enforce traffic management policies, security and resource usage. Your team can also control access with role-based control (RBAC). This allows you to regulate who can perform specific actions and on which services they are able to do so.

Service discovery and registry

Dynamic service discovery automatically discovers services and their instances. This ensures the mesh is constantly up-to-date and able to route traffic to the right instance. The service registration of a mesh allows dynamic scaling and seamless service discovery, even in a dynamic microservices architecture.

Integration and extensibility

Integration with existing tools is simple with service meshes, including building in observability, security, and management tools This gives your team the ability to seamlessly add your service mesh to existing infrastructures. They also offer extensible frameworks that allow operators to add custom plugins and extend their capabilities to meet specific requirements.

Popular service meshes to check out:

- A widely used service mesh, Istio offers robust traffic management, security, and observability features.
- Linkerd is a lightweight service mesh focused on simplicity, performance, and security.
- Part of HashiCorp Consul, Consul Connect provides service discovery, configuration, and secure service-to-service communication.

Load balancing in complex microservices architectures

Both service registries and service meshes have built-in load balancing aspects to distribute incoming traffic across multiple instances of a service. This way, they ensure optimal resource utilization, guarantee high availability, and generally improve performance. Here's a helpful explanation of load balancing from Software Engineering StackExchange:

> *"Load balancing a service allows clients to be decoupled from the scalability of those other services. All clients have a single URL to interact with. Cloud environments have automated tools that can add and remove nodes behind a load balancer. This helps enable the scalability promised with micro services.*
>
> *If load balancing decouples clients from the scalability of a service, then service discovery decouples clients from knowing which URLs can be used to communicate with the other services. Think of service discovery as an index of all the microservices in your ecosystem. The meta data about each service should return the URL of the load balancer in front of a service."*

There are supplementary tools you can use to increase the resiliency of your system and improve performance. NGINX, HAProxy, and cloud-based load balancers (e.g. AWS Elastic Load Balancing, Google Cloud Load Balancing), help distribute traffic across service instances based on various algorithms like round-robin, least connections, or weighted distribution.

These tools can implement load balancing at different levels, such as the network level (L4) or the application level (L7).

- **L4 load balancing** operates at the transport layer and distributes traffic based on IP addresses and ports.

- **L7 load balancing** operates at the application layer and can make routing decisions based on the content of the request, such as URLs or headers.

Now that we've covered the technical concepts, let's dive into an interesting example from Airbnb.

Airbnb builds in dynamic service discovery and load balancing with SmartStack

When Airbnb moved to microservices, their static service discovery solutions didn't work anymore. So, they developed proprietary software to manage service discovery, service registration and load balancing in their complex microservices architecture, which they called SmartStack.

Supported by observability and monitoring tools, SmartStack allowed Airbnb to create an effective service discover system in their microservices environment.

Registration, discovery and load balancing with SmartStack

Airbnb's development team decided to break the three services down into two distinct components, which they called nerve and synapse.

- **Nerve** is a service registration daemon that registers services with a distributed key-value store, much like Zookeeper. It periodically checks the health of the services and updates their registration status to maintain a clean registration of all functioning instances.
- **Synapse** is a service discovery and load-balancing component that acts as a transparent proxy for service communication. It subscribes to the key-value store to discover the available service instances and routes traffic to them based on configurable load-balancing algorithms.

These components integrate with Airbnb's infrastructure automation and deployment tools. They automatically register services as they are deployed to ensure all healthy service instances are discoverable. SmartStack also provides automatic failover, so if the system is compromised it re-routes all data to a standby system.

Integrating flexibility & resilience

SmartStack allowed services to discover and communicate without the need for manual configuration or hardcoded service locations, freeing the team to scale their services independently. Load balancing and automatic failover built resilience into the system by optimizing resource allocation (without overwhelming instances) and ensuring failures were handled gracefully.

Real-time insight

Airbnb integrated their observability and monitoring tools, such as Datadog and Grafana, with SmartStack to gain real-time visibility into the health and performance of their services. This gave them a better understanding of service dependencies, traffic patterns, and potential bottlenecks. Armed with this in-depth knowledge, Airbnb proactively identified and resolved issues before they caused chaos in the system.

With SmartStack, Airbnb created a service discovery system that could successfully manage the complexities of their microservices architecture. It improved reliability, enabled a structure that could scale independently and, in the end, delivered a seamless user experience to millions of users worldwide.

Chapter 5

Monitoring and observability

In a microservices environment, multiple services run concurrently, creating a complex network of processes. That makes it difficult to get a clear view of the overall health, performance, and behaviour of the application. And it often renders obsolete many traditional tools used in monolithic architectures.

That's why effective monitoring and observability tools are critical for understanding what is happening at each layer of your application.

- **Observability tools** provide insights into the internal state of the system. This makes it easier to understand and debug the complex interactions between services.
- **Monitoring tools** collect and analyze metrics, logs, and traces from various services. This gives you critical data you can use to identify potential issues, bottlenecks, and anomalies

Challenges of implementing monitoring and observability in microservices architectures

There are three major challenges companies have to overcome before achieving effective monitoring and observability.

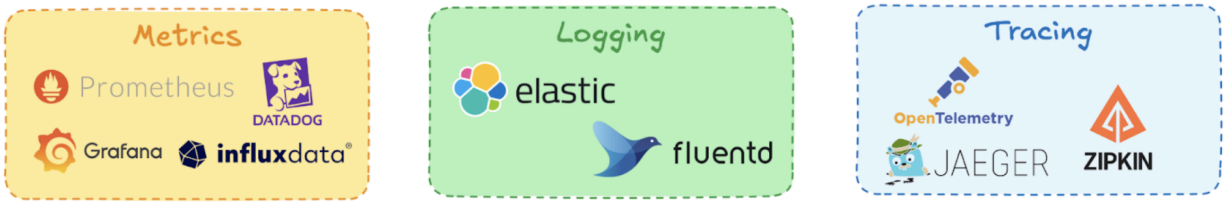
1. Interaction of data silos. Treating each microservice separately when implementing monitoring and observability solutions creates "data silos". These silos are easy to understand in isolation, without fully understanding how they interact as one. This can lead to difficulty when debugging or understanding the root cause of problems.

2. Scalability. As your microservices architecture scales, the complexity of monitoring and observability grows with it. So monitoring everything with the same tools you were using for a single monolith quickly becomes unmanageable.

3. Lack of standard tools. One of the benefits of microservices is that different teams can choose the data storage system that makes the most sense for their microservice (as we covered in chapter 2, Data management and consistency). But, if you don't have a standard for monitoring and observability, tying siloed insights together to gain insights on the system as a whole is challenging.

The foundation of monitoring and observability

To achieve effective monitoring and observability in a microservices environment, start with the three pillars of observability: metrics, logging, and tracing. Once the basics are established, you can move into enrichment, correlation, arbitrary querying, and more.



Metrics

Metrics provide quantitative measurements of various aspects of the system, such as response times, error rates, resource utilization, and throughput. By collecting and analyzing these metrics, you can assess the performance and health of individual services and the system as a whole.

Prometheus, Grafana, InfluxDB, and Datadog are all popular tools for collecting and visualizing metrics. They help define and collect metrics from services, set up alerts and thresholds, and create dashboards for real-time monitoring.

Logging

Capturing and centralizing log messages generated by services during their execution provides valuable information about the behaviour of services, including error messages, debug information, and important events. This data can be saved to keep a record of performance or used as input for analytics tooling.

Centralized logging solutions like the ELK Stack (Elasticsearch, Logstash, and Kibana) or Fluentd help aggregate and analyze logs across microservice architectures. They enable you to search, filter, and visualize log data, making it easier to troubleshoot issues and understand the flow of requests through the system.

Tracing

Tracing involves capturing the end-to-end flow of requests as they traverse across multiple services. Tracing gives your team a better understanding of interactions between services so they can identify performance bottlenecks and pinpoint the root cause of issues.

Distributed tracing tools like Jaeger, Zipkin, and OpenTelemetry allow you to capture the timing and metadata of requests as they flow through the system. This provides a detailed view of the request lifecycle.

Metrics, logging, and tracing tools provide comprehensive visibility into your microservices system. When you implement these tools you are better able to monitor the health and performance of services, detect anomalies, and troubleshoot issues efficiently. Now let's talk about some examples.

Uber maintains observability through growth

Uber experienced significant growth in their microservices architecture. This led to challenges in maintaining clear monitoring and observability. So, Uber focussed on building up the three pillars of monitoring and observability by implementing a robust stack of various open-source tools. This gave them a better understanding of what was happening across their app.

Metrics

For metrics, Uber used Prometheus to collect and store data and Grafana to visualize the results. Prometheus, a time-series database and monitoring system, was chosen for its flexible query language. This flexibility allowed Uber to define custom metrics relevant to their business. Grafana was used to create interactive dashboards and alerts based on the collected metrics.

Logging

Uber used Apache Kafka and Elasticsearch to build a centralized logging infrastructure that could handle their growth. Services published their logs to Kafka topics, which acted as a buffering layer. The logs were then consumed by a log aggregation pipeline that processed, transformed, and stored them in Elasticsearch. To make the data easier to search and more digestible, Uber used Kibana to visualize the log data.

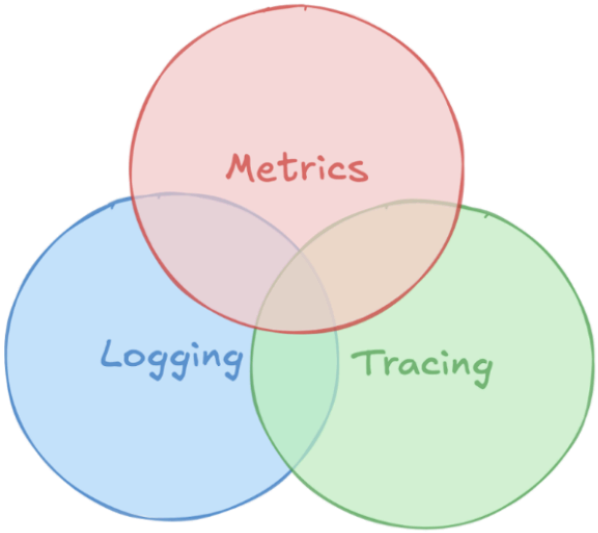
Tracing

For distributed tracing, Uber initially used Zipkin but later transitioned to Jaeger, an open-source tracing system. Jaeger allowed Uber to instrument their services to generate trace data and provided a web UI for visualizing and analyzing traces. It helped them understand the flow of requests through their microservices architecture, identify performance bottlenecks, and debug issues.

Standardizing metrics, logging, and tracing with custom tools

With a variety of tools recording data on how the architecture acted at multiple levels, Uber had to bring this information together to get observability in architecture as a whole.

They decided to develop custom tools and dashboards to standardize metrics, logs and traces across the system. This gave their team a holistic view of their system's health and performance. They can correlate data from different sources and gain insights into the behavior of their microservices.



Building up by focusing on the foundation

Uber re-gained deep visibility into their microservices architecture by revisiting the three pillars of observability and monitoring. With this focus on re-working the foundation of visibility, they are now able to proactively identify and resolve issues, optimize performance, and ensure a reliable and seamless experience for their users.

Chapter 6

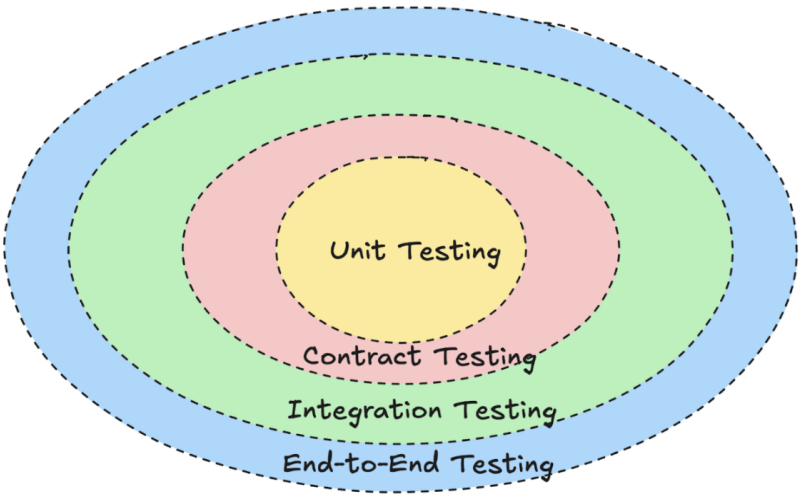
Testing and deployment strategies

The mesh of simultaneously communicating services creates unique challenges when testing and deploying in microservices. Not only do you have to ensure that each service is functioning internally, but that it's interacting effectively with other services. And then, you have to make sure the whole web of microservices is functioning together as a system.

With multiple layers acting and interacting simultaneously, you need strong testing and deployment strategies to maintain the quality and stability of microservices. That's what we will talk about in this chapter.

Testing strategies

You can't test all layers of a microservices architecture with a single type of test. Your team has to test your microservices at four different levels to ensure it is stable at each layer of functionality. You start testing at the granular, or unit level and work your way up to testing the whole system.



Unit testing

Unit testing in microservices is fundamentally the same as in monoliths. Your team tests individual components, or services in isolation to make sure they meet specified requirements. Frameworks such as JUnit, NUnit, and Mocha are popular choices in their specific language spaces. The main difference with microservices is that with microservices you're dealing with multiple frameworks within your app. So, you'll have to use different testing stacks for each microservice based on the technology used to create it.

Contract testing

Once you've ensured units are working well in isolation, you can begin to test how they are interacting. Contract testing is a constrained version of system testing. Focusing on only two units, consumer and provider, contract testing ensures that both have a shared understanding of the API contract. In other words, the consumer can check for aspects like response format, status codes, etc., while the producer can check that its response is in accordance with the agreed contract. Tools like Pact, Spring Cloud Contract, and Swagger can be used for contract testing.

Integration testing

At the next level up, you'll have to test the interaction and communication in the system as a whole. During integration testing, you'll ensure that each of those units, which you have tested and shown to work effectively in isolation, are communicating effectively with other units to create a cohesive system. Tools like Postman, SoapUI, and REST-assured can be used for integration testing of RESTful APIs, for example.

End-to-end testing

Finally, end-to-end testing validates the entire system from the user's perspective. Unlike integration testing, which validates how microservices work together, E2E tests

execute the entire set of calls involved in any user action, testing the end result to validate that the full feature is working as expected to fulfill business requirements. Tools like Selenium, Cypress, and Cucumber can be used for end-to-end testing.

In a dynamic system like microservices, you should be continuously running tests to make sure issues don't creep in through updates or patches. Practices like Continuous Integration (CI) and Continuous Testing automate the testing process and catch issues early in the development cycle.

Deployment strategies

When your team is rolling out your new microservices structure, they're not just rolling out a single project. They're rolling out multiple independent services, each one dependent on the others. That complexity requires a careful, well-thought-through deployment process to minimize problems and the potential knock-on effects throughout the system.

The right deployment strategies help you do just that. Below are four effective approaches to consider for a smoother rollout.

Blue-green deployment

In blue-green deployment, you run two identical production environments, i.e. "blue" and "green", during updates. One of these environments, in this case blue, runs the old version while you deploy the new version of your microservices on green. During the update, all traffic is directed to the stable (blue) environment. Once the green environment (i.e. the updated version) is tested and validated, you switch traffic over to it. This approach allows for quick rollbacks in case of issues and ensures that there is no downtime during deployment.

Canary deployment

With canary deployment, you gradually roll out a new version of a microservice to a small subset of users or servers. The old version runs alongside the new version (the canary) while traffic is diverted to the canary subset by subset. If the canary performs well, the rollout continues until all traffic is shifted to the new version. If there is an issue, you can roll back the update by switching all users back to the old version. This strategy allows you to test the new version in a production environment with real traffic, without risking all users.

Rolling update

Rolling updates are performed in batches, where a portion of the instances are updated while the remaining instances continue to serve traffic. Once the updated instances are stable, the next batch is updated. This process continues until all instances are updated. This system minimizes downtime and allows for a quick rollback in case a problem is found, so you can minimize the number of users affected by it.

Serverless deployment

Serverless deployment uses serverless platforms like AWS Lambda, Azure Functions, or Google Cloud Functions to deploy and run functions or small units of code, which can be used to implement microservices. Essentially, this splits the microservice into smaller, individual parts. Serverless platforms abstract away infrastructure management and automatically scale these functions based on demand. This approach simplifies deployment and allows for granular scaling of individual functions, which can be orchestrated to form a microservices architecture.

Every app needs to be continually updated to stay healthy and up-to-date. So, organizations often adopt Continuous Deployment (CD) practices to streamline the

deployment process. CD pipelines automate the build, testing, and deployment steps to make frequent and reliable releases easier.

Ensuring durable deployment with rigorous testing – Netflix's testing and deployment strategy

When Netflix rolled out their microservices architecture, they had to completely change how they updated the app. In response to the change, they developed a suite of tools and practices they could use to test and deploy their updates without interrupting service to their customers

Netflix’s testing & stress testing strategy

Netflix had to develop new testing strategies to make sure they understood what was happening at all levels of their app, including unit testing, integration testing, and end-to-end testing. They chose Junit, Mockito, and Spock to test at the unit level and REST-assured for integration testing of their RESTful APIs.

Netflix also chose to stress test each microservice through a process called Test Annihilation. Essentially, what they do is purposefully inject failures into their microservices during testing to see how the update deals with the faults. This helps them ensure the resilience and fault tolerance of their microservices.

Deploying updates

Netflix pioneered the use of Canary Deployment and Red/Black Deployment (similar to Blue-Green Deployment). By using a combination of strategies, they were able to ensure continuous service to their subscribers. In the same way, they developed Spinnaker, an open-source, multi-cloud continuous delivery platform, to automate their deployment pipelines. Spinnaker allows Netflix to safely and efficiently deploy microservices across diverse regions and cloud providers.

Planning for the unexpected

Before any update is deployed, Netflix stress tests each microservice again with Chaos Engineering practices. They developed the tools Chaos Monkey and Chaos Kong to simulate failures and disruptions in their infrastructure so they could see how their updates dealt with unexpected scenarios and failures. This gives them the ability to proactively build resilience into their architecture, ensuring that their microservices can handle unexpected scenarios.

Ready to scale reliably

The massive scale of Netflix's reach requires testing to failure before risking the next update. To do that, Netflix adopted comprehensive testing strategies, automated deployment pipelines, and chaos engineering practices. This way, Netflix has built a highly reliable and scalable microservices architecture that can handle the massive scale and complexity of their streaming service.

Chapter 7

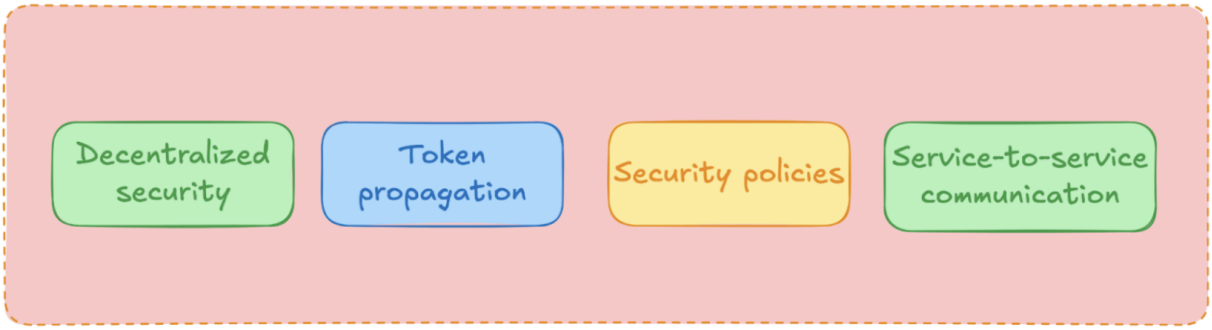
Security and access control

As your team decomposes your monolith over a distributed network of constantly communicating microservices, it creates an increased attack surface. If your security isn't enhanced to deal with these new vulnerabilities, it leaves your system more exposed than it was as a monolith.

That's why it's essential that your team understands the potential vulnerabilities of a microservices architecture and knows how to safeguard against them.

Potential security vulnerabilities in microservice architectures

Flexibility is one of the benefits of using microservices. However, if not done correctly, this flexibility can result in inconsistencies which create vulnerabilities. There are four main aspects where these vulnerabilities show up.



Decentralized security

In a monolith architecture, all of the security-related logic resides inside the same project and codebase. In a microservice context, however, your security can't be so simple. You will have to replicate and tailor your chosen security system to each service. If this isn't done properly, it opens the door to security issues down the line.

Token propagation

A common authentication technique for microservices is token-based authentication, where a token is issued to a user or another service during authentication. Because they interact with a variety of databases and security systems, they are more susceptible to compromise.

Security policies

Each microservice in your architecture has potentially been developed and maintained by different teams. If all teams are equally concerned with security, this isn't a problem. However, if even one team is inconsistent in the implementation of security policies and controls, it creates a vulnerability that can be used to access the whole system.

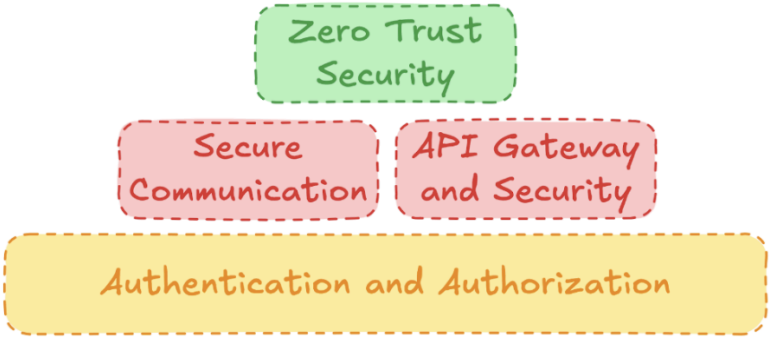
Service-to-service communication

Microservices are constantly communicating with each other. To keep this communication secure, your team needs to make sure that each microservice is authenticated and authorized when communicating with others. Improper authentication and authorization create vulnerabilities in your security that can be utilised to compromise the system.

Protecting your microservices architecture

There is no simple, band-aid solution that will secure every vulnerability listed above. However, by following best practices your team can design your microservices architecture to lower their impact on your microservices architecture.

We'll cover the four most important best practices below



Authentication and authorization

Authentication verifies the identity of a user or service, validating they are who they say they are. Authorization determines what actions or resources they are allowed to access. To ensure users and services are properly authenticated and only allowed to access the resources they're authorized for, microservice architects often use token-based authentication mechanisms like JSON Web Tokens (JWT) or OAuth 2.0.

In token-based authentication, users/services are issued a token after they are successfully authenticated by the system. This token contains a variety of data, including the user's identity and their permissions and is included in every request made by the user/service. This allows the receiving system to authenticate the user at each step and authorize (or deny) access to protected resources.

Spring Security, Microsoft Entra ID, and Cerbos are all able to implement token-based authentication and authorization in your microservices architecture.

Secure communication

Microservices are heavily reliant on communication across servers and storage types, so secure communication channels are essential for the integrity of your security system. Transport Layer Security (TLS) and mutual TLS (mTLS) are both effective ways to secure communication. They can (and should be) used in unison to maximise your security.

TLS encrypts communication channels to protect data from eavesdropping and tampering with authentication from the client. Mutual TLS authentication then adds an additional layer of security as it requires both the client and the server to authenticate each other.

API Gateway and security

An API Gateway acts as a single access point for external clients, limiting the routes into the system to give you more control over your traffic. Besides limiting access to one portal, an API Gateway can also request authentication, validate tokens, control access and even provide rate limiting. It can also act as a reverse proxy, hiding the internal microservices architecture to provide an additional layer of security.

Kong, Apigee, and Amazon API Gateway can all be used to implement API Gateways in your microservices architecture.

Zero Trust security

Zero Trust is a security framework based on the principle "never trust; always verify". Services that use the system assume no implicit trust for any entity, whether inside or outside the network. Instead, it requires authentication and authorization with every request.

When users/services are authenticated, access is only granted based on the principle of least privilege. That means users and services only have the necessary permissions to perform their tasks. This helps to minimize the impact of a potential security breach.

You can implement a Zero Trust framework using Mutual TLS (mTLS) for authentication, then layer in Cerbos for the fine-grained authorization, Role-Based Access Control (RBAC), and Attribute-Based Access Control (ABAC) you'll need to enforce access policies.

Though it can be quite complex, the tools/practices above can help you ensure your microservices architecture remains secure–which is exactly what Netflix did.

The systems and tools Netflix used to ensure security in their microservices

When Netflix decomposed their monolith in favour of a microservices architecture (which we covered in chapter 1), ensuring the continued security of their app was absolutely essential. So, they adopted a Zero Trust security model and then implemented both in-house-developed and off-the-shelf software to ensure all traffic was authenticated and authorized properly.

Minimizing unwanted access with a zero-trust model

One of the biggest steps Netflix took to ensure security in their system was to embrace a zero-trust model. To enforce the principle of least privilege, they use mechanisms like Open Policy Agent (OPA) so they can define and enforce access policies at the microservice level. They also took a Role-Based Access Control approach to managing permissions and accessing rights. So each user and service is assigned specific roles, and access to resources is granted based on these roles.

Taking a closer look at authentication and authorization

Netflix chose to back up their zero-trust approach to authorization with a token-based system. They chose Stethoscope to handle user authentication and generate JSON Web Tokens (JWTs) containing both user identity and their permissions.

Clearing communication

Netflix needed to ensure the integrity of their data in transit, so they chose to use both TLS and mTLS. They use TLS to provide encryption for communication, while mTLS requires mutual authentication, ensuring that both the client and the server authenticate each other.

Developing an API Gateway and security

With a microservices architecture as large as Netflix's traffic could become overly complex, making security difficult. To bring order to the chaos, Netflix developed an in-house API Gateway called Zuul, creating a single entry point for all client requests. Zuul handles authentication, rate limiting, and access control for the services, as well as performing request routing and load balancing, and distributing requests to the appropriate microservices.

Good monkey

In addition to these security measures, Netflix decided to go above and beyond by automating continuous monitoring of their systems. They developed Security Monkey, an automated monitoring tool, to monitor and assess the security posture of their infrastructure without oversight. This constant vigilance identifies misconfigurations, policy violations, and potential security risks before they become problems, allowing Netflix to proactively address security issues.

Netflix addressed the vulnerabilities of a microservices architecture head-on with a comprehensive security architecture and Zero Trust principles. As a result, they have a secure microservices ecosystem that protects their user's data.

Chapter 8

Performance and scalability

One of the benefits of a microservices architecture is that it allows companies to scale services independently and choose different storage and systems for each separate service. This opens up new ways to optimize performance and allows companies to scale services depending on each service's individual workload demands.

Despite this, building a scalable well-performing microservices architecture is a major challenge when transitioning. That's because microservices are more complex, so performance measurements and service scaling drastically differ from what is expected in a monolith.

The challenges of optimizing a microservices architecture

When transitioning from a monolith, you're very aware of your performance goals and the methods you can use to achieve them. But when you transition to a microservices architecture, those goals and tools go out the window.

After the transition, your team will have to set new best practices based on the new architecture and find new tools to achieve them. In the following section, we'll cover some tools and best practices you can use to optimize your microservices architecture.

Service communication and latency

In a microservices architecture, you have to balance granularity and communication speed.

Overly granular architectures require microservices to communicate with many other microservices to answer any specific request. Requested data then needs to "hop" between all these microservices before responding to the client. Each "hop"

that the data makes on its way to generating a proper answer adds microseconds (or more) to the communication.

This causes high latency where the communication protocol takes more time than the actual services, which can increase the chance of failures.

If instead, you build with the sole intent of lowering latency, you'll end up with an overly integrated system that isn't flexible or scaleable. Instead, you need to balance your need for speed with your need for a scalable architecture. When you understand that compromise you can create goals that are attainable for your team.

Data management and consistency

In principle, each microservice manages its own database. However, in execution, this can lead to challenges in maintaining data consistency across microservices. If there is too much difference between service databases, transactions across services become cross-database transactions. This complexifies communication and impacts performance.

So what is theoretically possible (different databases for different services) becomes more trouble than it's worth in the real world. That means you have to find the tradeoffs that work best for you to lower the complexity of transactions by using similar databases while making the most of your microservices flexibility.

Scalability

One of the major advantages of microservices is that each can scale—or be scaled—independently. However, managing and orchestrating the scaling of multiple services can get overwhelmingly complex. So most microservice deployments use auto-scaling policies and mechanisms that act based on predefined criteria. The good news is most cloud providers offer this functionality. And, there are also open-source solutions, including Kubernetes and Docker Swarm, available.

To ensure these scaling mechanisms work seamlessly in practice, testing is essential. Here is a good reminder from Sarada V., Director, Enterprise DevOps Coach at Sun Life:

> *"Testing the scalability of a micro-service is very critical as it ensures that the architecture can handle the increased workload effectively. Different capabilities like: vertical / horizontal scaling should be tested thoroughly to make sure that there is no impact to performance or to overall throughput."*

Deployment and DevOps complexity

Handling one deployment pipeline is difficult enough, but with microservices, each deployment involves managing numerous, independent deployment pipelines.

So while ensuring the role out of multiple pipelines, your team has to ensure compatibility between components and handle the orchestration of services. This increases the complexity of your continuous integration and continuous deployment (CI/CD) processes.

Inter-service dependencies

While it's theoretically possible to make changes in one microservice without changing others, that's not always true in execution. Often the web of inter-service dependencies between microservices means a change in one system requires a change in another.

When you run into these dependencies, it requires careful versioning and intricate backward compatibility considerations to move forward successfully.

Each of these issues above can compromise your microservices architecture if not proactively addressed. However, none of them present an intractable problem. For every issue above there is a pattern or product that can help you solve it.

Designing scalable, high-performing microservice architectures

The best way to mitigate the potential problems above is to design a well-balanced microservices architecture, starting with capacity planning.

Capacity planning and auto-scaling

Before moving to a microservice architecture, your team should have a solid understanding of the resources required to meet their expected workload and performance requirements. That requires your team to estimate the number of instances, as well as the CPU, memory, and storage capacity needed by those instances, giving you a base capacity to start with. Once you've defined your base, your team will have to define how and when you'll scale your architectures to deal with increased workload.

Only after you've set these guidelines in place can you bring in auto-scaling mechanisms to automatically adjust the number of service instances. Platforms like Kubernetes and cloud services like AWS Auto Scaling or Google Cloud Autoscaler are great tools to help you define scaling policies and automatically scale services.

Service granularity

When you're building a new microservices architecture, there's always a temptation to create extremely fine-grained services to increase scalability and flexibility. But this approach quickly hits diminishing returns as fine-grained services start to introduce additional network overhead and latency due to increased inter-service communication (as mentioned previously). Then if you go too far the other way, you end up with a different set of troubles. While maintaining coarse-grained services may reduce network overhead, it also limits scalability and agility.

Finding balance is the key. That takes careful analysis of your business domain, performance requirements, and scalability needs so you can identify the boundaries of each service based on its functionality, data ownership, and scalability characteristics.

Once you understand your needs, you'll know what compromises make more sense so you can find balance in your architecture.

Caching

Caching data allows communication-heavy microservices to improve their performance. By storing frequently accessed data or the results of computationally expensive operations, it reduces the load on downstream dependencies and improves response times.

In microservices, caching can be done at multiple different levels.

- **Local caching** allows each service to maintain its own cache where it can store the data it frequently accesses. Both Redis and Memcached are suitable for creating local caches.
- A **distributed cache**, which is shared across multiple services, gives all services access to unified cache data. This makes it easier for microservices to horizontally scale by externalizing their local cache, allowing requests to land

on any of the copies of the service without loss of context. Tools like Redis Cluster, Hazelcast, or Apache Ignite offer distributed caching.

- Caching can also be implemented at the **HTTP level** by using headers like Cache-Control and ETag to allow clients to cache responses. Here is an interesting opinion on it:

  "Caching can reduce the HTTP request-response time to get data from distant servers. Microservices regularly require information from other sources (data repositories, legacy systems, etc.). Real-time calls to these sources may involve latency. Caching helps minimize the number of backend calls made by your application." - Shares Sumit Bhatnagar, VP of Software Engineering.

Asynchronous communication

When microservices communicate synchronously in a large system, it can lead to bottlenecks, making responses slower than expected.

Asynchronous communication patterns, such as message queues and event-driven architectures, open up those bottlenecks by enabling services to decouple their interactions and operate independently, so producers can send messages to a message broker, and consumers can process those messages at their own pace. This loose coupling allows for better scalability, fault tolerance, and improved performance.

Asynchronous messaging tools like Apache Kafka, RabbitMQ, or AWS SQS allow you to implement asynchronous communication in your architecture.

Database optimization

Microservices architectures usually rely on multiple databases or data storage types, each chosen to complement its service. Because of this, your team can significantly improve the responsiveness and scalability of the entire system by optimizing each of these databases.

Optimizing your caching, as mentioned above, is the first step your team can take to optimize database traffic.

Database sharding can also help you optimize your databases. By partitioning data horizontally across multiple database instances based on a specific key, sharding allows your team to distribute the data load more evenly, improving scalability in your architecture.

Depending on your use case, NoSQL databases, such as MongoDB, Cassandra, or Couchbase can also provide better scalability and performance.

When you take the time to plan the compromises and requirements of microservices architecture, you can mitigate issues around complexity, allowing it to perform and scale to its potential. Let's look at how Amazon did exactly that.

AWS ensures performance for millions of users

AWS supplies a huge variety of services to an ever-expanding clientele base. Despite this complexity and traffic, their microservices architecture continues to perform at a high level. They used a variety of techniques and technologies to maintain optimal performance and scalability across their microservices.

Balancing speed with flexibility

AWS based their microservices architecture on DDD principles, aligning software design with their underlying business concepts and processes. This allowed them to balance flexibility (fine-grained services) and communication speed (coarse-grained services) in a way that worked for their business needs.

Reduced load and improved response times with proper caching

AWS uses caching extensively to improve performance. They use ElastiCache, which uses Redis and Memcached to provide in-memory caching services, to cache each microservice's frequently accessed data. This improves response times by reducing database load.

At the API Gateway level, AWS uses an in-house product, Amazon API Gateway Caching. This tool allows them to cache API responses, which reduces the number of requests hitting the backend services, improves overall performance and reduces costs.

Building in loose coupling with asynchronous communication

AWS built their own in-house message queues and streaming platforms to ensure their services could scale and act independently.

Amazon Simple Queue Service (SQS) is their message queueing service. It's a fully managed service that allows nodes to send messages to SQS queues, so consumers can process those messages asynchronously. This makes scaling services independently easier by decoupling microservices.

Amazon developed Kinesis as a bespoke data streaming service. It uses event-driven architecture to process and analyze data in real time.

Automatic optimization

AWS built a fully managed relational database service, named Amazon Aurora, to maintain high-performance, scalable databases. It optimizes varying workloads on the fly through automatic sharding, read replicas, and serverless options to handle varying workload demands.

For NoSQL workloads, AWS uses Amazon DynamoDB, a highly scalable and performant NoSQL database. DynamoDB offers automatic scaling, in-memory

caching, and support for global tables to ensure high-throughput data access with low latency.

Optimizing resource use with capacity planning and auto-scaling

Before implementing their microservice, the AWS team went through an in-depth capacity planning process. Then, they laid out auto-scaling parameters based on their understanding of each service's needs. With defined scaling parameters in place, they turned to auto-scaling to ensure each service scaled when it should.

Instead of using an off-the-shelf piece of software, they developed Amazon CloudWatch to monitor service metrics and trigger auto-scaling actions based on these predefined rules. They used Amazon Elastic Container Service (ECS) or Amazon Elastic Kubernetes Service (EKS) to deploy newly scaled services. This combination of services allows AWS to define scaling policies so services can automatically scale up or down based on workload.

Chapter 9

Organizational and cultural shift

Transitioning to microservices is primarily a technical exercise, but it's not only technical. It also requires a fundamental cultural shift in your team. This presents a unique difficulty for teams. It can become a major hurdle for engineering teams with deeply ingrained monolithic practices because the first step is challenging the old culture to make way for the new.

Challenging the culture

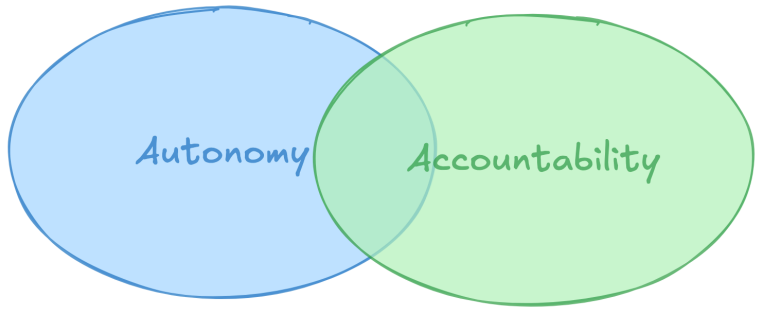
In a monolithic architecture, all teams will typically work in the same codebase but each with its own set of priorities and objectives. This creates a highly coupled code, where one team's priorities and objectives directly affect another's. As a result, the decision-making process relies on a single decision-maker or body of decision-makers. This leadership group has to attempt to understand and account for the interests of every team so they don't work at cross purposes. Naturally, this structure leads to bottlenecks, slow development cycles, and a considerable lack of agility.

Microservices demand a more collaborative, cross-functional team culture. Teams have to collaborate across functions, instead of in silos, to ensure the system functions properly. This requires open communication channels and regular team check-ins.

When teams can let go of their old, monolithic habits, they are freed to share best practices, learn from each other's experiences, and develop a deep collective understanding of the system as a whole.

But making this new culture work isn't simple. The team must have a foundation of autonomy backed by accountability so they can move from doing what they're told to making the decision to do what works best for their service. In a microservices culture, those two words (autonomy and accountability) pack a lot of meaning.

It's worth unpacking some of that here.



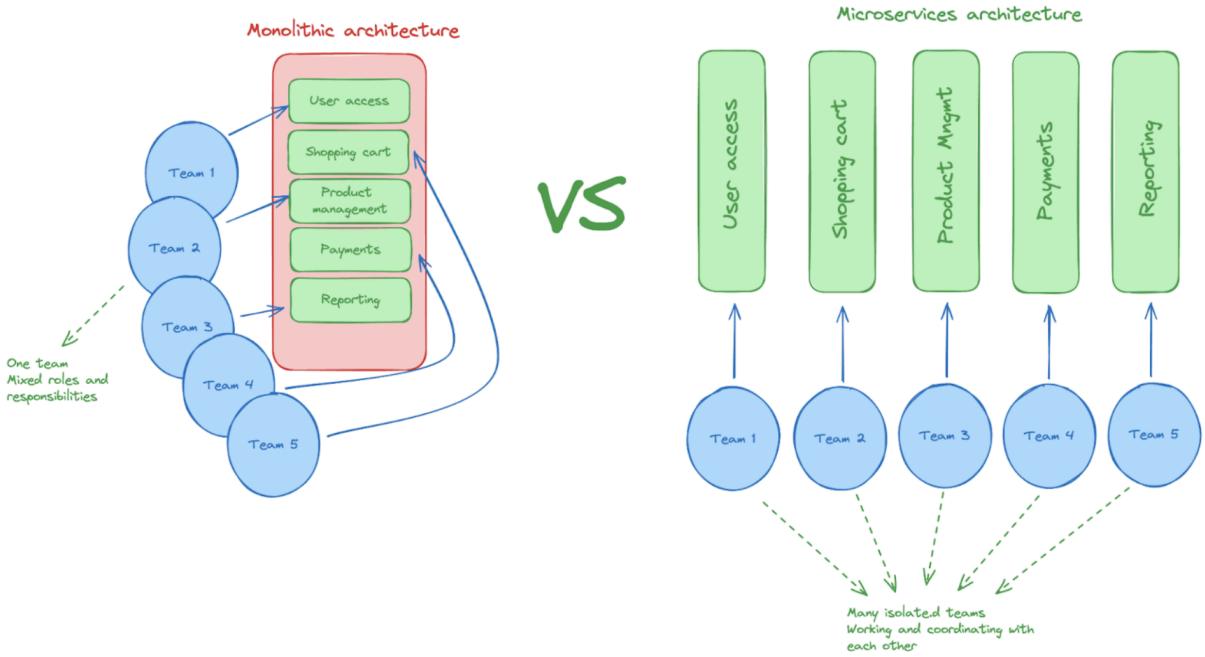
- **Autonomy** is in direct contradiction with the old centralized governance of monolithic architecture. Instead of being constrained by centralized decision-making processes, teams are given the freedom to make their own decisions about the design, development, and deployment of their services.

  That includes choosing the technologies, tools, and processes that best fit their needs. Letting go of that centralized control is difficult for many team leaders, but it's essential for microservices as it enables faster innovation and experimentation.

- Instead of having a centralized governing team who is responsible for the consequences of decision-making, microservices architecture requires that **accountability** is spread to everyone working on the project. Every team must take ownership of the decisions they make for their services throughout the entire lifecycle, from development to production. They are responsible for the quality, performance, and reliability of their services, as well as addressing any issues that may arise.

  When all teams honestly take accountability for their services, it builds a sense of responsibility and promotes a proactive approach to service management.

Successfully decentralizing



Accountability and autonomy are the foundation, but they don't just appear out of nowhere, or as a result of a single meeting. These are built over time as your leadership and your developers become more comfortable with these new concepts.

But that doesn't mean you just have to wait until all the pieces fall into place. There are several steps you can take to encourage this collaborative approach:

- **Assemble cross-functional teams** that cover diverse skill sets and expertise to tackle specific challenges or projects. This allows team members to share their knowledge and skills, while also gaining exposure to new areas.
- **Establish guilds** that bring experts from different domains or functional areas together. This provides a platform for knowledge sharing, experimentation, and innovation between domains.
- **Create internal communities** where team members can come together to share experiences, discuss challenges, and learn from each other.

In addition to this foundation, traditional hierarchical structures may need to be flattened to emphasize the importance of autonomous, self-organizing teams. This truly frees team members from traditional, monolithic decision-makers so they can take ownership of their work, make decisions independently, and respond quickly to changing requirements.

That requires engineering leadership to take a leap of faith and step back from constant decision-making to focus on setting clear goals to provide guidance and direction to teams. They must also be willing to adapt and evolve alongside the organization, embracing the uncertainty and ambiguity that often accompanies a significant cultural shift.

Ultimately, the key to successfully transitioning your company lies in creating an organizational culture that values collaboration, knowledge sharing, and autonomy. To get the most out of a microservices architecture, companies need to let go of the old ways of working (old engineering culture).

Building a microservices culture at Amazon

Shifting the culture in any company is a difficult process. But when you're as big as Amazon, it can seem overwhelming. So instead of simply telling developers and leadership about the change and requiring everyone to get on board, the leadership at Amazon came up with innovative, concrete steps they could take to change the culture.

The first step was snack-sized.

The pizza rule

A simple rule with an interesting name, the "two-pizza team" rule stipulates that all teams should be small enough to be fed with two pizzas. Typically, that means about 6–8 members.

These small teams are responsible for the end-to-end development and operation of their services. That means everything from ideation to deployment and maintenance is the work of a single team. In this type of workflow, teams have the freedom to choose technologies and make architectural decisions based on their specific requirements, which allows them to move quickly and independently.

Owning your metrics

Amazon promotes a culture of ownership and accountability by using metrics and monitoring. Teams can see how their services are doing by tracking the metrics they are responsible for proactively addressing any issues that arise. And if they don't, their metrics will reflect that lack of action to leadership.

Asynchronous and synchronous workflows that facilitate collaboration

Dictating collaboration is impossible because it relies on strong relationships and communication in teams. That's why Amazon has established mechanisms and practices that encourage collaboration and knowledge sharing instead of simply telling their communities to collaborate.

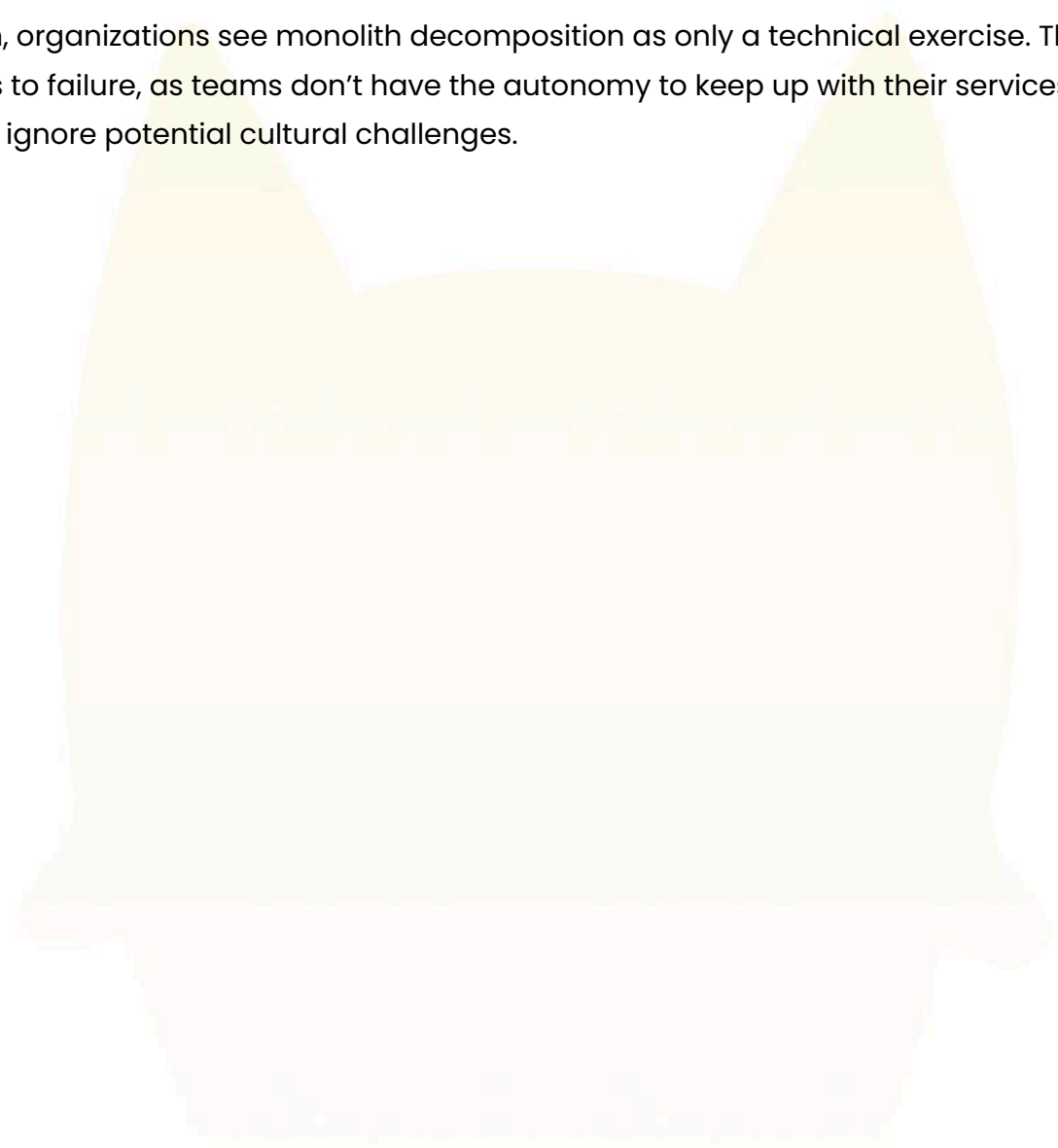
Amazon has long had a strong culture of writing and documentation (including services, APIs, and best practices). After decomposing their monolith, they doubled down on this to include wikis and mailing lists where everyone can learn from each other. They also implemented cross-team meetings to ensure communication across the two-pizza teams.

Flattening the hierarchy

As teams proliferated, the towering hierarchy stopped working, so leadership embraced a flatter, decentralized structure. They took a DDD approach to this new

structuring, organizing teams around business capabilities and giving them clear ownership and accountability for their services.

Often, organizations see monolith decomposition as only a technical exercise. This leads to failure, as teams don't have the autonomy to keep up with their services. Don't ignore potential cultural challenges.



Chapter 10

Team collaboration and code ownership

Team collaboration and code ownership are important in every architecture, but when you decompose your monolith, they become essential throughout your organization.

Microservices are typically developed and managed by multiple teams, each of which is responsible for their own specific services or domains. These teams not only have to collaborate to build and maintain an app, but the members of each team have to work together without recourse to an outside judge.

This autonomy requires effective collaboration and clear code ownership for every developer. And for those at the top, it requires a lot of trust in your teams. But that doesn't mean you have to sit on the sidelines hoping for a good result. When you take the time to set up team structures that work in this new context, you position your teams for success. That starts with re-organizing your team.

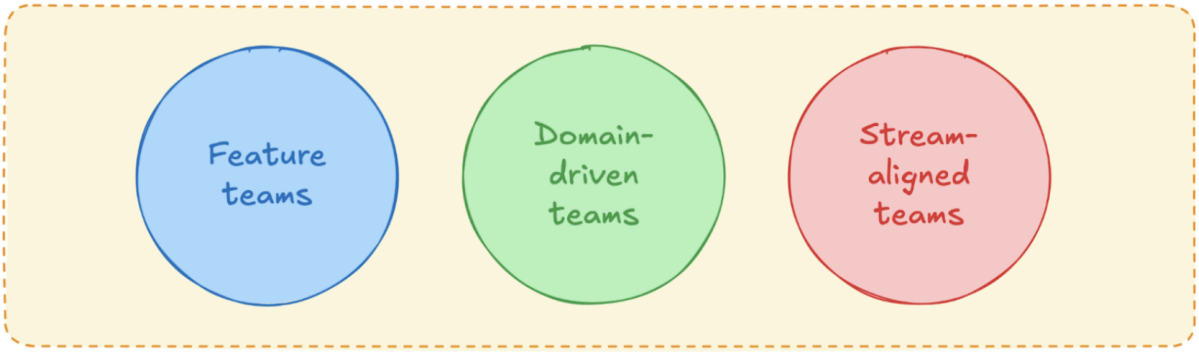
Organizing teams to excel

Much like the services they run, microservices teams are often organized around business capabilities or domains rather than technical layers. This is known as "vertical slicing". It allows teams to have end-to-end ownership of their services, from development to deployment and maintenance. This builds cross-functionality into the team and gives them the freedom to make the best choices for their particular service.

Depending on the goals of your teams, there are a variety of ways you can choose to vertically slice them.

- **Feature teams** are responsible for developing and maintaining a specific set of features or business capabilities.
- Some companies choose to create **domain-driven teams** by aligning each with specific business domains or subdomains following the principles of DDD.

- **Stream-aligned teams** are organized around the flow of work, such as user journeys or data streams, allowing for end-to-end ownership and faster delivery.



These teams need to stay agile, so it's best to keep them small. The "two-pizza team" rule, popularized by Amazon, is an effective rule of thumb to follow when putting together teams. The two-pizza rule states that teams should be small enough to be fed with two pizzas, typically consisting of 6–8 members.

Giving teams ownership of their code and repositories

Each team has to own their code. That means they need control of their own code repository that covers the team's domain and boundaries of service. That often means that each service has its own repository, though not necessarily. Related services may also be grouped together in a single repository which is owned by its respective team.

Regardless of how you define the borders, clear repository boundaries are integral to enforcing access control, managing dependencies, and facilitating independent deployments. This decentralized ownership structure promotes autonomy, which, as we discussed in chapter 9, is an essential aspect of working in a microservices architecture.

Decentralized doesn't mean chaotic, however. Setting a small set of foundational standards—such as basic tech stack, coding "best practices" or project

structures—will allow developers to move between service teams if need arises. In the end, while the teams are independent, they're all working towards releasing the same, integrated platform, and that common goal should be reflected in how they pursue their goals.

Facilitating communication and collaboration

Teams need to establish clear communication channels and practices to share knowledge, coordinate efforts, and resolve dependencies. That doesn't happen by accident.

You can help your team develop good habits by instituting a variety of techniques, like:

- **Scheduling regular cross-team check-ins** where everyone can share updates, discuss challenges, and align on common goals.
- **Establishing knowledge-sharing sessions** around specific technical or domain areas so teams can share best practices, learn from each other, and maintain consistency across services.
- **Encouraging teams to document** knowledge, including their services, APIs, and best practices. Wikis, knowledge bases, or shared repositories can be effective tools for this.

Setting API contracts and service agreements

In the same way that your teams need to communicate, your services do as well. In microservices architectures, that means establishing clear API contracts and service agreements.

Well-defined expectations for inputs, outputs, and behaviour of a service's API create a shared understanding between the service provider and consumer teams which allows your services to collaborate easily and reduces dependencies between teams. When API contracts inevitably need to change, the revisions must be

communicated and versioned properly to avoid breaking dependencies and communication.

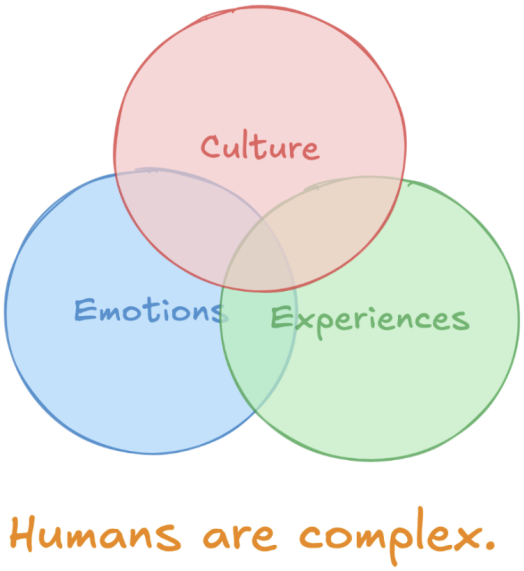
Service level agreements (SLAs) will help your team set expectations and define the responsibilities of each team in delivering and consuming services.

Leveraging continuous integration and continuous delivery

With multiple teams across your application updating services separately, and keeping everyone up-to-date with each node, and ensuring integration through updates becomes an almost overwhelming task.

That's why most teams who use microservice architecture deploy some form of Continuous integration (CI) and Continuous Delivery (CD) pipelines. These automate the build, testing, and deployment processes, so teams can frequently integrate their code changes, and deploy services independently, allowing for efficient collaboration.

As every team needs the ability to work autonomously, each team should have its own CI/CD pipeline. This way they can develop, test, and deploy their services without worrying about what other teams are doing. Pipelines should be configured to run automated tests, perform code quality checks, and deploy services to the appropriate environments based on predefined criteria.



Humans are complex. So, changing team culture is often more difficult than changing technical practices. But that doesn't mean it's impossible. Spotify is a great example of a company that managed to create a lasting change in their culture when they shifted to a microservices architecture.

Organizing cross-company communication with Spotify

When Spotify decomposed its monolith, it also decomposed its hierarchy to create a flatter, more collaborative team culture. This major shift offers great lessons on how to encourage your people to change by increasing communication.

Organized in squads and tribes

While Amazon has their 'two pizza' rule, Spotify has its squads. Like 'two-pizza' teams, squads are small. But they're also defined by more than just their size. They're cross-functional teams that own and develop specific features or services. Each

squad has the autonomy to make their own decisions, choose their own technologies, and deliver value independently.

These squads are then grouped into tribes based on how closely related their areas or domains are. Tribes provide a forum for squads to come together to share knowledge, align on best practices, and coordinate efforts so that they don't become silos.

By creating different levels of organization, and implementing practices that bring them together, Spotify gives each of its teams a high degree of autonomy while still bringing them together.

Maintaining a plethora of clear ownership boundaries

Spotify uses a microservices architecture with hundreds of services, each owned by a specific squad. Each of those squads has the freedom to choose the programming languages, frameworks, and tools that best suit their needs as they develop and maintain their service.

This creates a dizzying array of technologies and frameworks. All the services, irrespective of ownership, are stored in a single repository using a monorepo approach. But they don't all get tossed together without concern for teams or services.

Each squad has its own dedicated folder within the monorepo to maintain clear ownership boundaries.

Creating opportunities to meet new people and learn skills

Formally, all developers are grouped into squads and tribes at Spotify, but the groups don't end there. Spotify encourages all developers to build bridges beyond their squad/tribe with a variety of collaboration practices.

- **Guilds**, which are communities of interest, are open to people from all different squads and tribes. These groups host meetings where anyone interested can come to share knowledge and discuss specific topics. These topics could include web development, data engineering, agile practices, etc.
- **Developers can also join "chapters"**, which are groups of people with similar skills or expertise, such as front-end developers or data scientists. Chapters provide a forum to share best practices, learn from others, and maintain technical excellence.
- **Tribes also hold regular meetings** to discuss high-level goals, share updates, and coordinate efforts among the squads within the tribe.

API contracts and service agreements

Each squad at Spotify is responsible for designing and documenting their APIs, allowing them autonomy. These API specifications are stored alongside the code in the squad's folder in the mono repo. Any changes to those API specifications need to be reviewed by the relevant squads before being put into action. They are accountable for ensuring backward compatibility and avoiding breaking changes.

However, the squads aren't given a full blank slate. Each is required to follow consistent guidelines and standards to ensure compatibility and a shared understanding between them.

In the same way, Spotify defines standards for service level objectives (SLOs), including expected performance, availability, and reliability targets. Squads are then responsible for monitoring and meeting these SLOs.

Continuous integration and delivery

The squad's autonomy continues through to their CI/CD pipeline. They use tools like Jenkins, Travis CI, and Spinnaker to automate their build, testing, and deployment processes. This enables them to continuously integrate code changes, run automated tests, and deploy services independently.

Code reviews and pair programming are common practices within squads to maintain code quality, share knowledge, and foster collaboration.
As they rebuilt their architecture, Spotify also rebuilt their team organization, which is a large part of why their migration was so successful. Through the squad model, Spotify has created a highly collaborative and efficient development environment that many other companies are constantly trying to replicate across the world.

* * *

Switching to microservices can be tricky, but if you're ready for the challenges and plan ahead, you can increase your chances of success. The key is being open to the cultural changes, getting different teams to work together, leveraging best practices and tools, so your team can navigate this transition effectively. It's not going to be simple, but stick with it and be willing to adapt as you go.

The journey to microservices takes time and you'll be learning a lot along the way, but if you can navigate that, the benefits of this new way can really pay off.

cerbos

Authorization management solution for authoring, testing, and deploying access control policies. Implement scalable and secure fine-grained authorization.

cerbos.dev