Microservices

Amit Kumar

Microservices

Microservice architecture tells us to break a product or project into independent services so that it can be deployed and managed solely at that level and doesn't depend on other services.

Pros of microservices

Easy to develop, test, and deploy

The biggest advantage of microservices over other architectures is that small single services can be built, tested, and deployed independently. Since a deployment unit is small, it facilitates and speeds up development and release. Besides, the release of one unit isn’t limited by the release of another unit that isn’t finished. And the last plus here is that the risks of deployment are reduced as developers deploy parts of the software, not the whole app.

Pros of microservices

Easy to develop, test, and deploy : The biggest advantage of microservices over other architectures is that small single services can be built, tested, and deployed independently. Since a deployment unit is small, it facilitates and speeds up development and release.

Increased agility : With microservices, several teams can work on their services independently and quickly. Each individual part of an application can be built independently due to the decoupling of microservice components.

Ability to scale horizontally: horizontal scaling (creating more services in the same pool) isn’t limited and can run dynamically with microservices. Furthermore, horizontal scaling can be completely automated.

Cons of microservices

Complexity: The biggest disadvantage of microservices lies in their complexity. Splitting an application into independent microservices entails more artifacts to manage

Security concerns : In a microservices application, each functionality that communicates externally via an API increases the chance of attacks. These attacks can happen only if proper security measurements aren’t implemented when building an app.

Different programming languages: The ability to choose different programming languages is two sides of the same coin. Using different languages make deployment more difficult. In addition, it’s harder to switch programmers between development phases when each service is written in a different language.

Advantages of Microservices

- Microservices are self-contained, independent deployment module.
- The cost of scaling is comparatively less than the monolithic architecture.
- Microservices are independently manageable services. It can enable more and more services as the need arises. It minimizes the impact on existing service.
- It is possible to change or upgrade each service individually rather than upgrading in the entire application.
- Microservices allows us to develop an application which is organic (an application which latterly upgrades by adding more functions or modules) in nature.

Advantages of Microservices

- It enables event streaming technology to enable easy integration in comparison to heavyweight interposes communication.
- Microservices follows the single responsibility principle.
- The demanding service can be deployed on multiple servers to enhance performance.
- Less dependency and easy to test.
- Dynamic scaling.
- Faster release cycle.

Disadvantages of Microservices

- Microservices has all the associated complexities of the distributed system.
- There is a higher chance of failure during communication between different services.
- Difficult to manage a large number of services.
- The developer needs to solve the problem, such as network latency and load balancing.
- Complex testing over a distributed environment.

Challenges of Microservices Architecture

- Microservice architecture is more complex than the legacy system. The microservice environment becomes more complicated because the team has to manage and support many moving parts. Here are some of the top challenges that an organization face in their microservices journey:

- Bounded Context
- Dynamic Scale up and Scale Down
- Monitoring
- Fault Tolerance
- Cyclic dependencies
- DevOps Culture

Challenges of Microservices Architecture

- **Bounded context**: The bounded context concept originated in Domain-Driven Design (DDD) circles. It promotes the Object model first approach to service, defining a data model that service is responsible for and is bound to. A bounded context clarifies, encapsulates, and defines the specific responsibility to the model. It ensures that the domain will not be distracted from the outside. Each model must have a context implicitly defined within a sub-domain, and every context defines boundaries.
- In other words, the service owns its data and is responsible for its integrity and mutability. It supports the most important feature of microservices, which is independence and decoupling.

Challenges of Microservices Architecture

- **Dynamic scale up and scale down**: The loads on the different microservices may be at a different instance of the type. As well as auto-scaling up your microservice should auto-scale down. It reduces the cost of the microservices. We can distribute the load dynamically.
- **Monitoring**: The traditional way of monitoring will not align well with microservices because we have multiple services making up the same functionality previously supported by a single application. When an error arises in the application, finding the root cause can be challenging.

Challenges of Microservices Architecture

- **Fault Tolerance**: Fault tolerance is the individual service that does not bring down the overall system. The application can operate at a certain degree of satisfaction when the failure occurs. Without fault tolerance, a single failure in the system may cause a total breakdown. The circuit breaker can achieve fault tolerance. The circuit breaker is a pattern that wraps the request to external service and detects when they are faulty. Microservices need to tolerate both internal and external failure.
- **Cyclic Dependency**: Dependency management across different services, and its functionality is very important. The cyclic dependency can create a problem, if not identified and resolved promptly.
- **DevOps Culture**: Microservices fits perfectly into the DevOps. It provides faster delivery service, visibility across data, and cost-effective data. It can extend their use of containerization switch from Service-Oriented-Architecture (SOA) to Microservice Architecture (MSA).

Other challenges of microservices

- As we add more microservices, we have to be sure they can scale together. More granularity means more moving parts, which increase complexity.
- The traditional logging is ineffective because microservices are stateless, distributed, and independent. The logging must be able to correlate events across several platforms.
- When more services interact with each other, the possibility of failure also increases.

Difference between Microservices Architecture (MSA) and Services-Oriented Architecture (SOA)

Microservice Based Architecture (MSA)	Service-Oriented Architecture (SOA)
Microservices uses **lightweight protocols** such as **REST**, and **HTTP**, etc.	SOA supports **multi-message protocols**.
It focuses on **decoupling**.	It focuses on application service **reusability**.
It uses a **simple messaging system** for communication.	It uses **Enterprise Service Bus** (ESB) for communication.
Microservices follows "**share as little as possible**" architecture approach.	SOA follows "**share as much as possible architecture**" approach.
Microservices are much better in **fault tolerance** in comparison to SOA.	SOA is not better in fault tolerance in comparison to MSA.
Each microservice have an **independent** database.	SOA services share the **whole** data storage.
MSA used **modern** relational databases.	SOA used **traditional** relational databases.
MSA tries to **minimize** sharing through bounded context (the coupling of components and its data as a single unit with minimal dependencies).	SOA **enhances** component sharing.
It is better suited for the **smaller** and **well portioned**, web-based system.	It is better for a **large** and **complex** business application environment.

Microservices Monitoring

- Monitoring is the control system of the microservices. As the microservices are more complex and harder to understand its performance and troubleshoot the problems. Given the vivid changes to software delivery, it is required to monitor the service. There are **five** principles of monitoring microservices, as follows:
- Monitor container and what's inside them.
- Alert on service performance.
- Monitor services that are elastic and multi-location.
- Monitor APIs.
- Monitor the organizational structure.

These principles allow us to address technological changes associated with the microservices and organizational changes related to them.

Microservices Monitoring

- There are three monitoring tools are as follows:
- Hystrix dashboard
- Eureka admin dashboard
- Spring boot admin dashboard

Microservice Virtualization

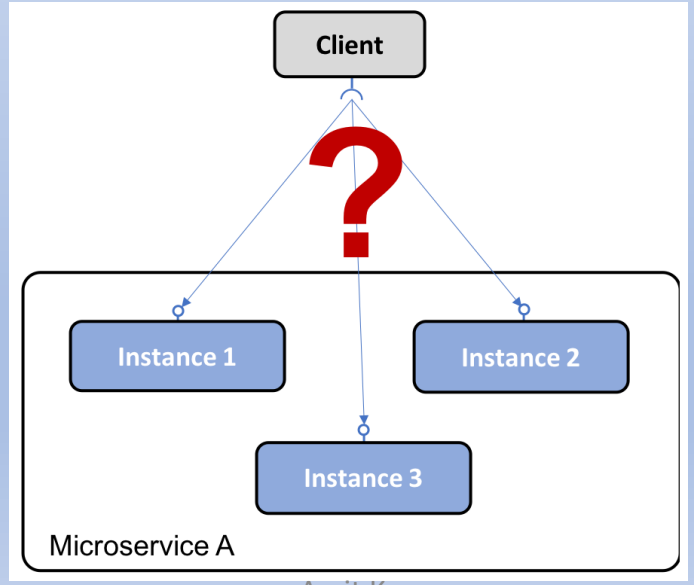
- Microservices virtualization is the method to simulate the behavior of specific components in various component-based application like cloud-based application, SOA, and API driven architecture. Service virtualization also reduces cost and save time. By combining service virtualization, an organization can develop the application which can be delivered from various locations and dissimilar environments.

Service discovery

- The **service discovery** pattern has the following problem:

How can clients find microservices and their instances?

Microservices instances are typically assigned dynamically allocated IP addresses when they start up, for example, when running in containers. This makes it difficult for a client to make a request to a microservice that, for example, exposes a REST API over HTTP. Consider the following diagram:



Service discovery

- The **service discovery** pattern has the following solution:

Automatically register/unregister microservices and their instances as they come and go.

The client must be able to make a request to a logical endpoint for the microservice. The request will be routed to one of the microservices available instances.

Requests to a microservice must be load-balanced over the available instances.

We must be able to detect instances that are not currently healthy; that is, requests will not be routed to them.

Implementation notes: As we will see, this design pattern can be implemented using two different strategies:

Client-side routing: The client uses a library that communicates with the service discovery service to find out the proper instances to send the requests to.

Server-side routing: The infrastructure of the service discovery service also exposes a reverse proxy that all requests are sent to. The reverse proxy forwards the requests to a proper microservice instance on behalf of the client.

Edge server

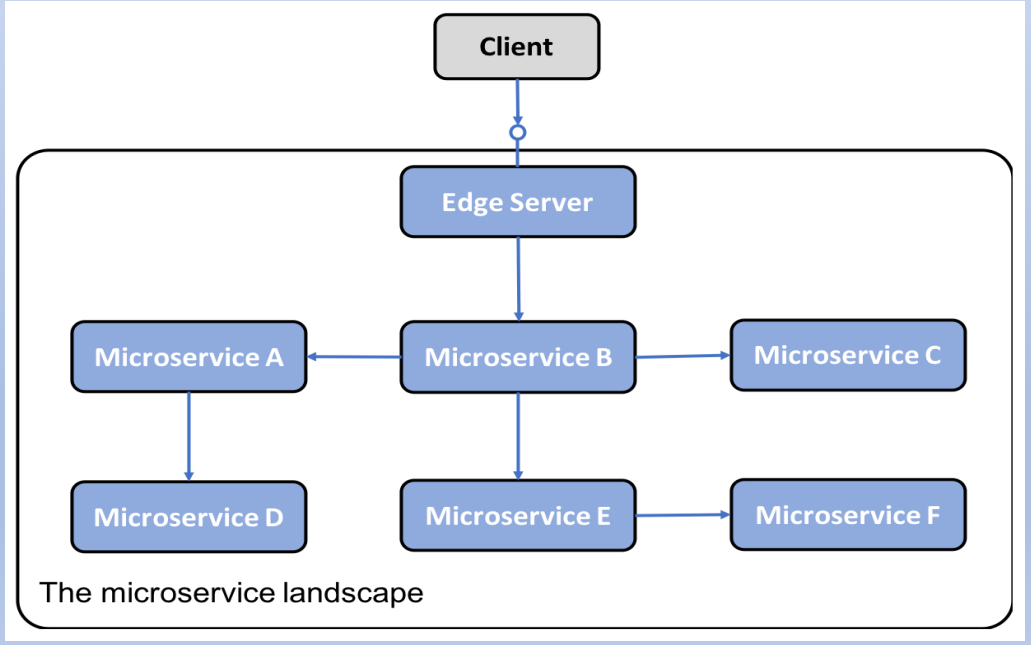
- **Problem**

In a system landscape of microservices, it is in many cases desirable to expose some of the microservices to the outside of the system landscape and hide the remaining microservices from external access. The exposed microservices must be protected against requests from malicious clients.

Edge server

- **Solution**

Add a new component, an Edge (Api Gateway) Server, to the system landscape that all incoming requests will go through:



Edge server

- **Solution**

Hide internal services that should not be exposed outside their context; that is, only route requests to microservices that are configured to allow external requests.

Expose external services and protect them from malicious requests; that is, use standard protocols and best practices such as OAuth, OIDC, JWT tokens, and API keys to ensure that the clients are trustworthy.

Reactive Microservices

- **Problem**

Traditionally, as Java developers, we are used to implementing synchronous communication using blocking I/O, for example, a RESTful JSON API over HTTP. Using a blocking I/O means that a thread is allocated from the operating system for the length of the request. If the number of concurrent requests goes up (and/or the number of involved components in a request, for example, a chain of cooperating microservices, goes up), a server might run out of available threads in the operating system, causing problems ranging from longer response times to crashing servers.

Also, as we already mentioned in this chapter, overusing blocking I/O can make a system of microservices prone to errors. For example, an increased delay in one service can cause clients to run out of available threads, causing them to fail. This, in turn, can cause their clients to have the same types of problem, which is also known as a chain of failures. See the *Circuit Breaker* section for how to handle a chain-of-failure-related problem.

Reactive Microservices

- **Solution**

Use non-blocking I/O to ensure that no threads are allocated while waiting for processing to occur in another service, that is, a database or another microservice.

- Whenever feasible, use an asynchronous programming model; that is, send messages without waiting for the receiver to process them.
- If a synchronous programming model is preferred, ensure that reactive frameworks are used that can execute synchronous requests using non-blocking I/O, that is, without allocating a thread while waiting for a response. This will make the microservices easier to scale in order to handle an increased workload.
- Microservices must also be designed to be resilient, that is, capable of producing a response, even if a service that it depends on fails. Once the failing service is operational again, its clients must be able to resume using it, which is known as self-healing.

Central Configuration

- **Problem**

An application is, traditionally, deployed together with its configuration, for example, a set of environment variables and/or files containing configuration information. Given a system landscape based on a microservice architecture, that is, with a large number of deployed microservice instances, some queries arise:

- How do I get a complete picture of the configuration that is in place for all the running microservice instances?
- How do I update the configuration and make sure that all the affected microservice instances are updated correctly?

Central Configuration

- **Solution**

Add a new component, a **configuration server**, to the system landscape to store the configuration of all the microservices.

Make it possible to store configuration information for a group of microservices in one place, with different settings for different environments (for example, dev, test, qa, and prod).

Spring Cloud Config server

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments.

The concepts on both client and server map identically to the Spring *Environment* and *PropertySource* abstractions, so they fit very well with Spring applications but can be used with any application running in any language.

As an application moves through the deployment pipeline from dev to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git, so it easily supports labelled versions of configuration environments as well as being accessible to a wide range of tooling for managing the content.

It is easy to add alternative implementations and plug them in with Spring configuration.

Spring Cloud Config server

Spring Cloud Configuration Server is a centralized application that manages all the application related configuration properties. Spring Cloud Config Server provides an HTTP resource-based API for external configuration (name-value pairs or equivalent YAML content). The server is embeddable in a Spring Boot application, by using the *@EnableConfigServer* annotation.

```
@EnableConfigServer
@SpringBootApplication
public class SpringCloudConfigServerApplication {
        public static void main(String[] args) {
            SpringApplication.run(SpringCloudConfigServerApplication.class, args);
        }
}
```

Spring Cloud Config server

Spring Cloud Config server dependency in your build configuration file.

POM.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Build.gradle:

```
compile('org.springframework.cloud:spring-cloud-config-server')
```

Spring Cloud Config server

The default strategy for locating property sources is to clone a git repository (at spring.cloud.config.server.git.uri) and use it to initialize a mini SpringApplication. The mini-application's Environment is used to enumerate property sources and publish them at a JSON endpoint.

The HTTP service has resources in the following form:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

where application is injected as the spring.config.name in the SpringApplication (what is normally application in a regular Spring Boot app), profile is an active profile (or comma-separated list of properties), and label is an optional git label (defaults to master.)

Spring Cloud Config server

Spring Cloud Config Server pulls configuration for remote clients from various sources. The following example gets configuration from a git repository (which must be provided), as shown in the following example:

application.properties:

```
spring.cloud.config.server.git.uri=https://github.com/amitkumar1211/amitkumar1211
```

application.yml:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/amitkumar1211/amitkumar1211
```

Spring Cloud Config server

Spring Cloud Config Client:

A Spring Boot application can take immediate advantage of the Spring Config Server (or other external property sources provided by the application developer). It also picks up some additional useful features related to Environment change events.

application.properties:

```
spring.config.import=optional:configserver:http://localhost:8888
```

Centralized log analysis

- **Problem**

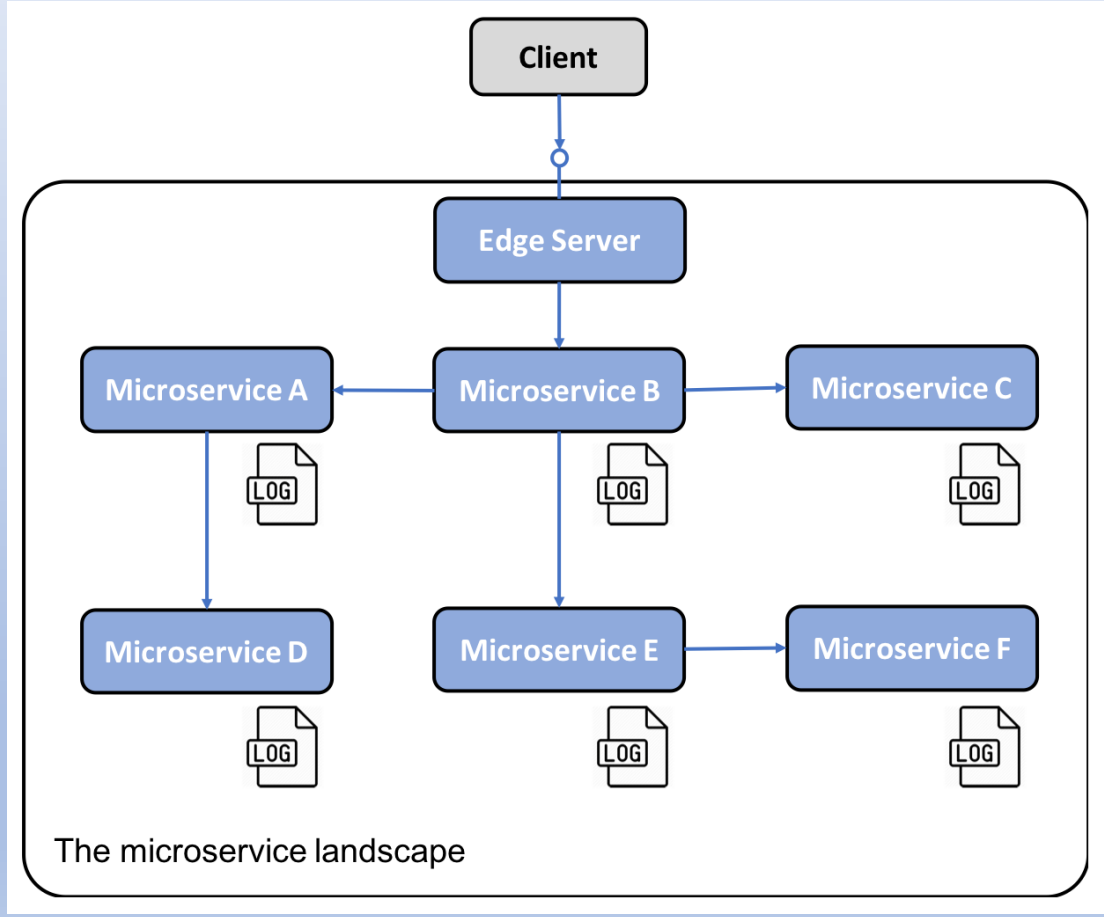
Traditionally, an application writes log events to log files that are stored on the local machine that the application runs on. Given a system landscape based on a microservice architecture, that is, with a large number of deployed microservice instances on a large number of smaller servers, we can ask the following questions:

- How do I get an overview of what is going on in the system landscape when each microservice instance writes to its own local log file?
- How do I find out if any of the microservice instances get into trouble and start writing error messages to their log files?

Centralized log analysis

- **Problem**

If end users start to report problems, how can I find related log messages; that is, how can I identify which microservice instance is the root cause of the problem? The following diagram illustrates the problem:



Centralized log analysis

- **Solution**

Add a new component that can manage centralized logging and is capable of the following:

Detecting new microservice instances and collecting log events from them

Interpreting and storing log events in a structured and searchable way in a central database

Providing APIs and graphical tools for querying and analyzing log events

Distributed tracing

- **Problem**

It must be possible to track requests and messages that flow between microservices while processing an external call to the system landscape.

Some examples of fault scenarios are as follows:

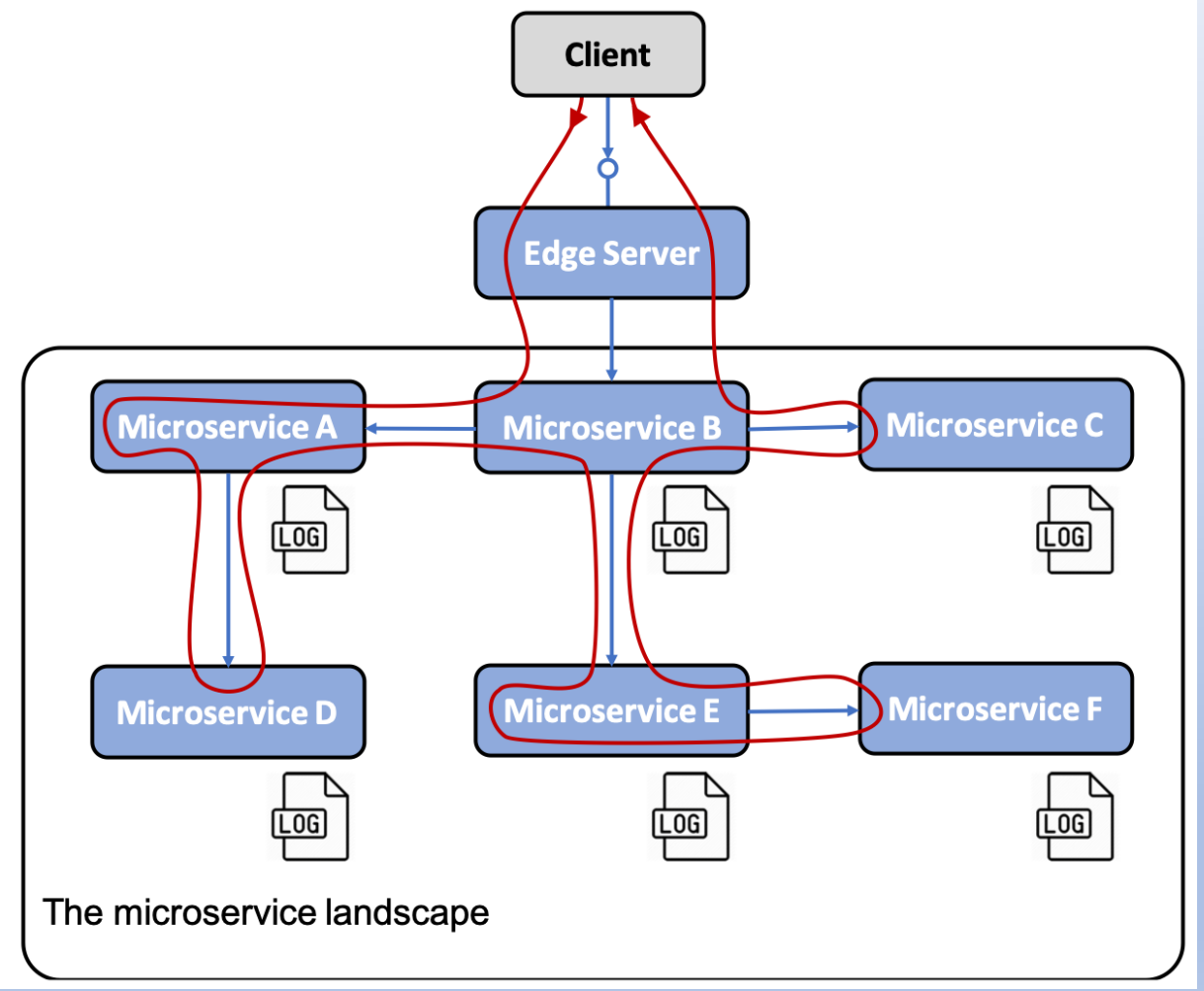
If end users start to file support cases regarding a specific failure, how can we identify the microservice that caused the problem, that is, the root cause?

If one support case mentions problems related to a specific entity, for example, a specific order number, how can we find log messages related to processing this specific order – for example, log messages from all microservices that were involved in processing this specific order?

Distributed tracing

- **Problem**

The following diagram depicts this:



Distributed tracing

- **Solution**

To track the processing between cooperating microservices, we need to ensure that all related requests and messages are marked with a common correlation ID and that the correlation ID is part of all log events. Based on a correlation ID, we can use the centralized logging service to find all related log events. If one of the log events also includes information about a business-related identifier, for example, the ID of a customer, product, order, and so on, we can find all related log events for that business identifier using the correlation ID.

Distributed tracing

- **Solution**

Assign unique correlation IDs to all incoming or new requests and events in a well-known place, such as a header with a recognized name.

When a microservice makes an outgoing request or sends a message, it must add the correlation ID to the request and message.

All log events must include the correlation ID in a predefined format so that the centralized logging service can extract the correlation ID from the log event and make it searchable.

Circuit Breaker

- **Problem**

A system landscape of microservices that uses synchronous intercommunication can be exposed to a chain of failure. If one microservice stops responding, its clients might get into problems as well and stop responding to requests from their clients. The problem can propagate recursively throughout a system landscape and take out major parts of it.

This is especially common in cases where synchronous requests are executed using blocking I/O, that is, blocking a thread from the underlying operating system while a request is being processed. Combined with a large number of concurrent requests and a service that starts to respond unexpectedly slowly, thread pools can quickly become drained, causing the caller to hang and/or crash. This failure can spread unpleasantly fast to the caller's caller, and so on.

Circuit Breaker

- **Solution**

Add a Circuit Breaker that prevents new outgoing requests from a caller if it detects a problem with the service it calls.

Open the circuit and fail fast (without waiting for a timeout) if problems with the service are detected.

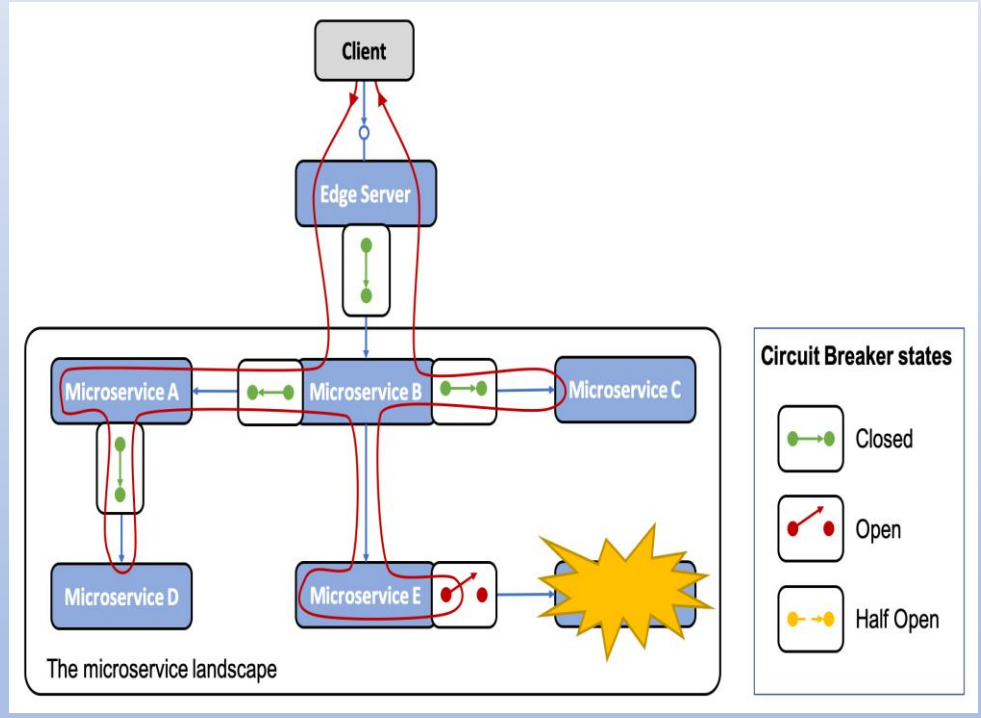
Probe for failure correction (also known as a half-open circuit); that is, allow a single request to go through on a regular basis to see if the service operates normally again.

Close the circuit if the probe detects that the service operates normally again. This capability is very important since it makes the system landscape resilient to these kinds of problems; that is, it self-heals.

Circuit Breaker

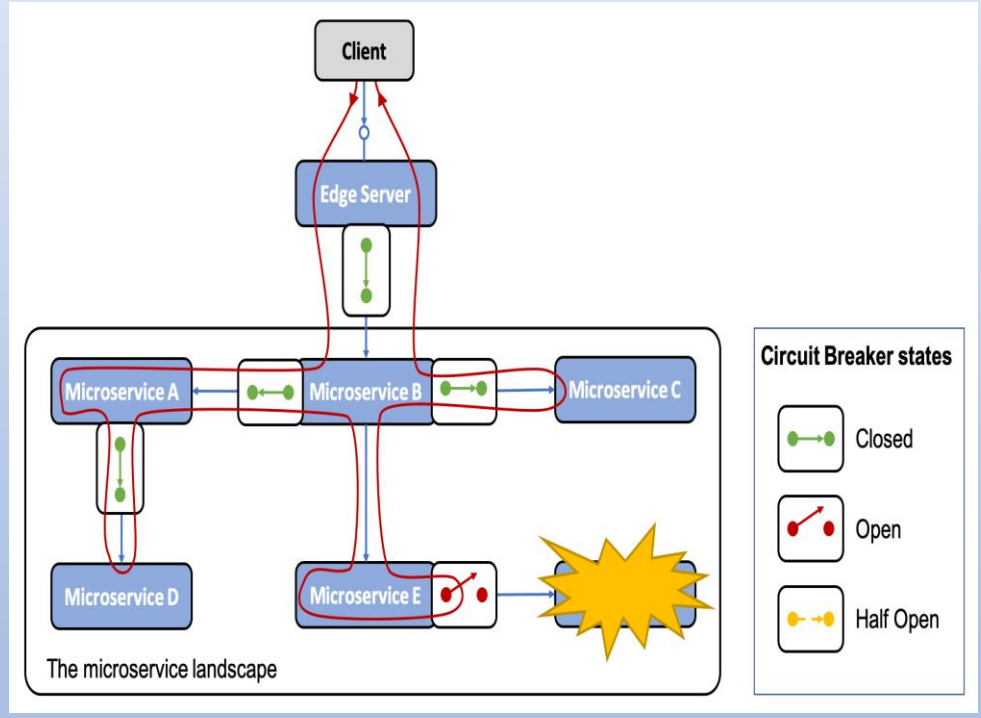
- **Solution**

The following diagram illustrates a scenario where all synchronous communication within the system landscape of microservices goes through Circuit Breakers. All the Circuit Breakers are closed; that is, they allow traffic, except for one Circuit Breaker detected problems in the service the requests go to. Therefore, this Circuit Breaker is open and utilizes fast-fail logic; that is, it does not call the failing service and waits for a timeout to occur. In the following, it immediately returns a response, optionally applying some fallback logic before responding.



Circuit Breaker

- The concept of a circuit breaker is to prevent calls to microservice when it's known the call may fail or time out. This is done so that clients don't waste their valuable resources handling requests that are likely to fail. Using this concept, you can give the server some spare time to recover.

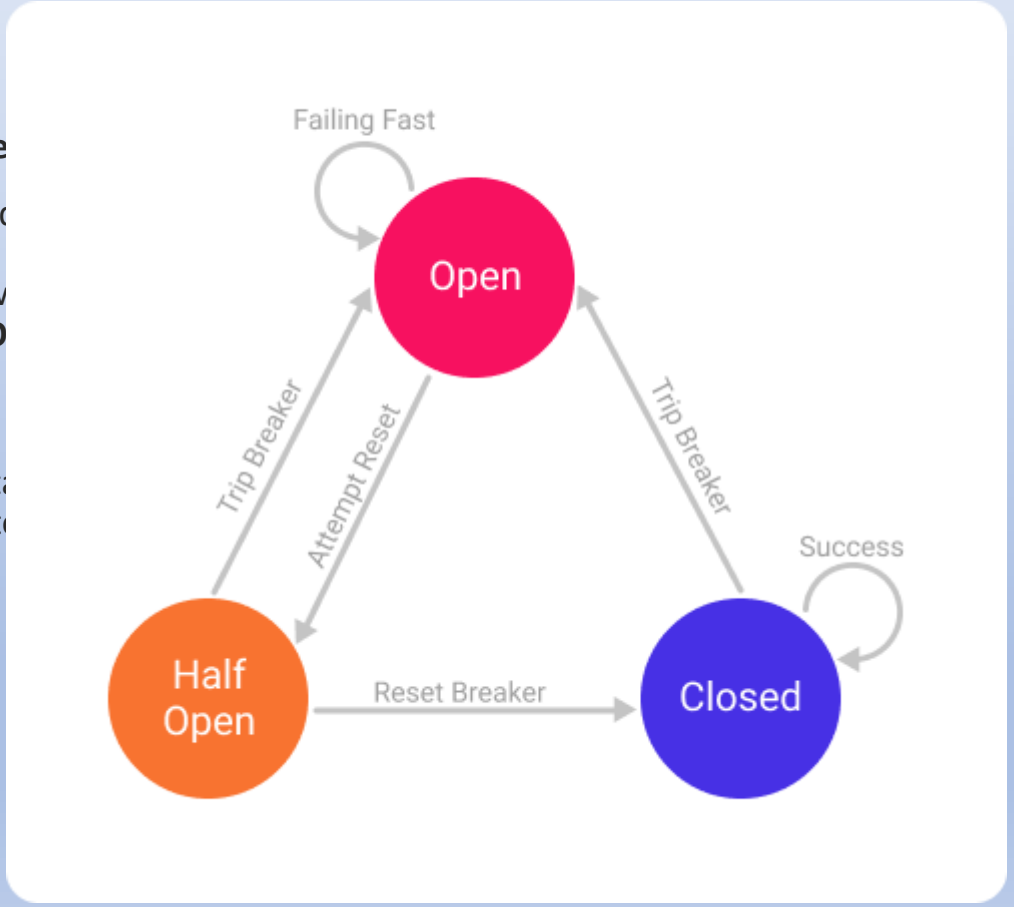


Circuit Breaker

Circuit Breaker State

In the circuit breaker, there are 3 states **Closed**, **Open**, and **Half-Open**.

- **Closed**: when everything is normal. Initially, the circuit breaker is in a **Close**
- **Open:** when a failure occurs above predetermined criteria. In this state, re to other microservices will not be executed and *fail-fast* or *fallback* will be performed if available. When this state has passed a certain time limit, it w automatically or according to certain criteria will be returned to the **Half-O state.**
- **Half-Open:** several requests will be executed to find out whether the microservices that we are calling are working normally. If successful, the st be returned to the **Closed state**. However, if it still fails it will be returned t the **Open state**.



Circuit Breaker

Circuit Breaker Type

There are 2 types of circuit breaker patterns, **Count-based** and **Time-based.**

- **Count-based**: the circuit breaker switches from a closed state to an open state when the last N requests have failed or timeout.
- **Time-based**: the circuit breaker switches from a closed state to an open state when the last N time unit has failed or timeout.

In both types of circuit breakers, we can determine what the threshold for failure or timeout is. Suppose we specify that the circuit breaker will trip and go to the **Open state** when 50% of the last 20 requests took more than 2s, or for a time-based, we can specify that 50% of the last 60 seconds of requests took more than 5s.

Control loop

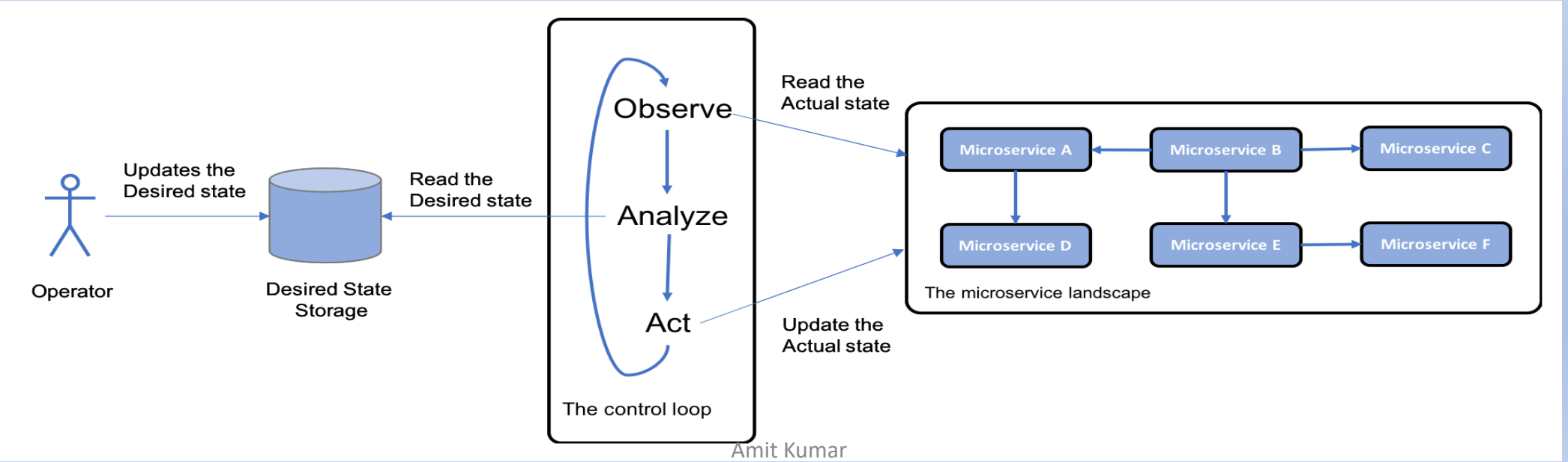
- **Problem**

In a system landscape with a large number of microservice instances spread out over a number of servers, it is very difficult to manually detect and correct problems such as crashed or hung microservice instances.

Control loop

- **Solution**

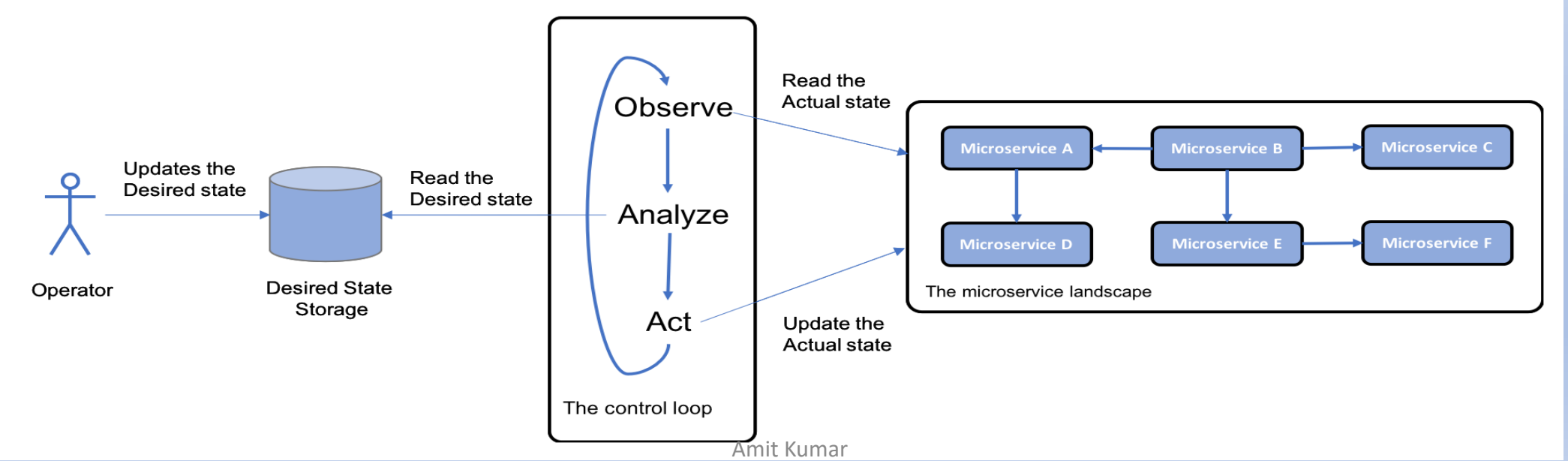
Add a new component, a control loop, to the system landscape; this constantly observes the actual state of the system landscape; compares it with the desired state, as specified by the operators; and, if required, takes action. For example, if the two states differ, it needs to make the actual state equal to the desired state:



Control loop

- **Solution**

Implementation notes: In the world of containers, a container orchestrator such as Kubernetes is typically used to implement this pattern. We will learn more about Kubernetes in Chapter 15, Introduction to Kubernetes.



Centralized monitoring and alarms

- **Problem**

If observed response times and/or the usage of hardware resources become unacceptably high, it can be very hard to discover the root cause of the problem. For example, we need to be able to analyze hardware resource consumption per microservice.

Centralized monitoring and alarms

- **Solution**

To curb this, we add a new component, a **monitor service**, to the system landscape, which is capable of collecting metrics about hardware resource usage for each microservice instance level.

- It must be able to collect metrics from all the servers that are used by the system landscape, which includes auto-scaling servers.
- It must be able to detect new microservice instances as they are launched on the available servers and start to collect metrics from them.
- It must be able to provide APIs and graphical tools for querying and analyzing the collected metrics

Software enablers

As we've already mentioned, we have a number of very good open-source tools that can help us both meet our expectations of microservices and, most importantly, handle the new challenges that come with them:

- Spring Boot
- Spring Cloud/Netflix OSS
- Docker
- Kubernetes
- Istio (a service mesh)

Design Pattern	Spring Boot	Spring Cloud	Kubernetes	Istio
Service discovery		Netflix Eureka and Netflix Ribbon	Kubernetes kube-proxy and service resources	
Edge server		Spring Cloud and Spring Security OAuth	Kubernetes Ingress controller	Istio ingress gateway
Reactive microservices	Spring Reactor and Spring WebFlux			
Central configuration		Spring Config Server	Kubernetes ConfigMaps and Secrets	
Centralized log analysis			Elasticsearch, Fluentd, and Kibana **Note**: Actually not part of Kubernetes but can easily be deployed and configured together with Kubernetes	
Distributed tracing		Spring Cloud Sleuth and Zipkin		Jaeger
Circuit Breaker		Resilience4j		Outlier detection
Control loop			Kubernetes controller manager	
Centralized monitoring and alarms			Grafana and Prometheus **Note:** Actually not part of Kubernetes but can easily be deployed and configured together with Kubernetes	Kiali, Grafana, and Prometheus

Other important considerations

- **Importance of Dev/Ops**: One of the benefits of a microservice architecture is that it enables shorter delivery times and, in extreme cases allows the continuous delivery of new versions. To be able to deliver that fast, you need to establish an organization where dev and ops work together under the mantra you built it, you run it. This means that developers are no longer allowed to simply pass new versions of the software over to the operations team. Instead, the dev and ops organizations need to work much more closely together, organized into teams that have full responsibility for the end-to-end life cycle of one microservice (or a group of related microservices). Besides the organizational part of dev/ops, the teams also need to automate the delivery chain, that is, the steps for building, testing, packaging, and deploying the microservices to the various deployment environments. This is known as setting up a delivery pipeline.

Other important considerations

- **Organizational aspects and Conway's law**: Another interesting aspect of how a microservice architecture might affect the organization is Conway's law, which states the following:
- "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."
- -- Melvyn Conway, 1967
- This means that the traditional approach of organizing IT teams for large applications based on their technology expertise (for example, UX, business logic, and databases-teams) will lead to a big three-tier application – typically a big monolithic application with a separately deployable unit for the UI, one for processing the business logic, and one for the big database. To successfully deliver an application based on a microservice architecture, the organization needs to be changed into teams that work with one or a group of related microservices. The team must have the skills that are required for those microservices, for example, languages and frameworks for the business logic and database technologies for persisting its data.

Other important considerations

Decomposing a monolithic application into microservices: One of the most difficult and expensive decisions is how to decompose a monolithic application into a set of cooperating microservices. If this is done in the wrong way, you will end up with problems such as the following:

- **Slow delivery:** Changes in the business requirements will affect too many of the microservices, resulting in extra work.
- **Slow performance:** To be able to perform a specific business function, a lot of requests have to be passed between various microservices, resulting in long response times.
- **Inconsistent data:** Since related data is separated into different microservices, inconsistencies can appear over time in data that's managed by different microservices.

Other important considerations

Importance of API design: If a group of microservices expose a common, externally available API, it is important that the API is easy to understand and consumes the following:

- If the same concept is used in multiple APIs, it should have the same description in terms of the naming and data types used.
- It is of great importance that APIs are allowed to evolve in a controlled manner. This typically requires applying a proper versioning schema for the APIs, for example, https://semver.org/, and having the capability of handling multiple major versions of an API over a specific period of time, allowing clients of the API to migrate to new major versions at their own pace.

Other important considerations

Migration paths from on-premise to the cloud: Many companies today run their workload on-premise, but are searching for ways to move parts of their workload to the cloud. Since most cloud providers today offer Kubernetes as a Service, an appealing migration approach can be to first move the workload into Kubernetes on-premise (as microservices or not) and then redeploy it on a Kubernetes as a Service offering provided by a preferred cloud provider.

Other important considerations

Good design principles for microservices, the 12-factor app: The 12-factor app (https://12factor.net) is a set of design principles for building software that can be deployed in the cloud. Most of these design principles are applicable to building microservices independently of where and how they will be deployed, that is, in the cloud or on-premise. Some of these principles will be covered in this book, such as config, processes, and logs, but not all.

The Twelve Factors

The twelve-factor methodology is a popular set of application development principles compiled by the creators of the Heroku cloud platform.

The Twelve Factors

Codebase	One codebase tracked in revision control, many deploys
Dependencies	Explicitly declare and isolate dependencies
Config	Store config in the environment
Backing services	Treat backing services as attached resources
Build, release, run	Strictly separate build and run stages
Processes	Execute the app as one or more stateless processes

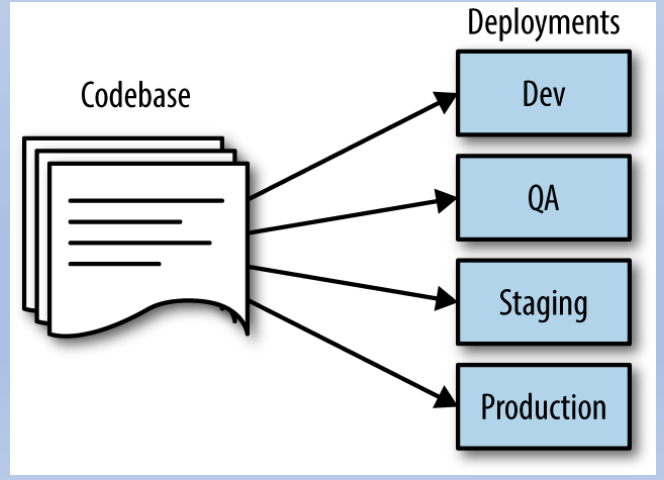
The Twelve Factors

Port binding	Export services via port binding
Concurrency	Scale out via the process model
Disposability	Maximize robustness with fast startup and graceful shutdown
Dev/prod parity	Keep development, staging, and production as similar as possible
Logs	Treat logs as event streams
Admin processes	Run admin/management tasks as one-off processes

Codebase

ONE CODEBASE TRACKED IN REVISION CONTROL, MANY DEPLOYS

Source code repositories for an application should contain a single application with a manifest to its application dependencies. There should be no need to recompile or package an application for different environments.



Dependencies

EXPLICITLY DECLARE AND ISOLATE DEPENDENCIES

Application dependencies should be explicitly declared, and any and all dependencies should be available from an artifact repository that can be downloaded using a dependency manager, such as Apache Maven.

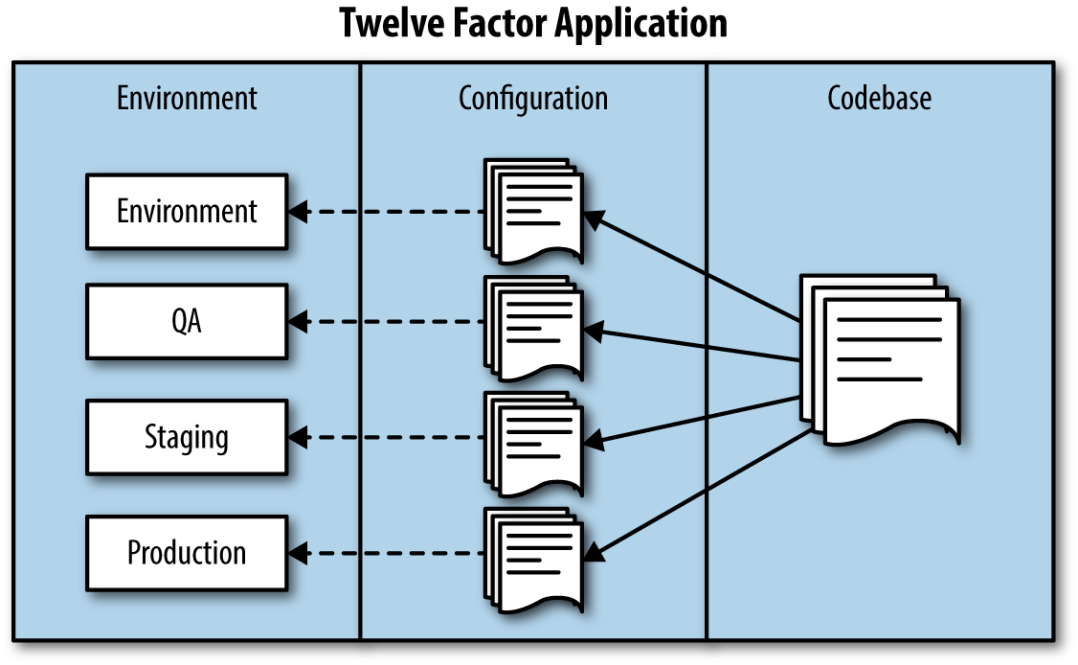
Twelve-factor applications never rely on the existence of implicit systemwide packages required as a dependency to run the application. All dependencies of an application are declared explicitly in a manifest file that cleanly declares the detail of each reference.

Config

STORE CONFIG IN THE ENVIRONMENT

Application code should be strictly separated from configuration. The configuration of the application should be driven by the environment.

Application settings such as connection strings, credentials, or hostnames of dependent web services should be stored as environment variables, making them easy to change without deploying configuration files.



Backing Services

TREAT BACKING SERVICES AS ATTACHED RESOURCES

A backing service is any service that the twelve-factor application consumes as a part of its normal operation. Examples of backing services are databases, API-driven RESTful web services, an SMTP server, or an FTP server.

Backing services are considered to be resources of the application. These resources are attached to the application for the duration of operation. A deployment of a twelve-factor application should be able to swap out an embedded SQL database in a testing environment with an external MySQL database hosted in a staging environment without making changes to the application’s code

Build, Release, Run

STRICTLY SEPARATE BUILD AND RUN STAGES

Build Stage: The build stage takes the source code for an application and either compiles or bundles it into a package. The package that is created is referred to as a build.

Release Stage: The release stage takes a build and combines it with its config. The release that is created for the deploy is then ready to be operated in an execution environment.

Run Stage: The run stage, commonly referred to as the runtime, runs the application in the execution environment for a selected release.

By separating each of these stages into separate processes, it becomes impossible to change an application’s code at runtime.

Processes

EXECUTE THE APP AS ONE OR MORE STATELESS PROCESSES

Twelve-factor applications are created to be stateless in a share-nothing architecture. The only persistence that an application may depend on is through a backing service. Examples of a backing service that provides persistence include a database or an object store. All resources to the application are attached as a backing service at runtime. A litmus test for whether or not an application is stateless is that the application’s execution environment can be torn down and recreated without any loss of data.

Twelve-factor applications do not store state on a local filesystem in the execution environment.

Port Bindings

EXPORT SERVICES VIA PORT BINDING

Twelve-factor applications are completely self-contained, which means that they do not require a web server to be injected into the execution environment at runtime in order to create a web-facing service. Each application will expose access to itself over an HTTP port that is bound to the application in the execution environment. During deployment, a routing layer will handle incoming requests from a public hostname by routing to the application’s execution environment and the bound HTTP port.

Concurrency

SCALE OUT VIA THE PROCESS MODEL

Applications should be able to scale out processes or threads for parallel execution of work in an on-demand basis. JVM applications are able to handle in-process concurrency automatically using multiple threads.

Applications should distribute work concurrently, depending on the type of work that is used. Most application frameworks for the JVM today have this built in. Some scenarios that require data processing jobs that are executed as long-running tasks should utilize executors that are able to asynchronously dispatch concurrent work to an available pool of threads.

The twelve-factor application must also be able to scale out horizontally and handle requests load-balanced to multiple identical running instances of an application.

Disposability

MAXIMIZE ROBUSTNESS WITH FAST STARTUP AND GRACEFUL SHUTDOWN

The processes of a twelve-factor application are designed to be disposable. An application can be stopped at any time during process execution and gracefully handle the disposal of processes.

Processes of an application should minimize startup time as much as possible. Applications should start within seconds and begin to process incoming requests. Short startups reduce the time it takes to scale out application instances to respond to increased load.

If an application’s processes take too long to start, there may be reduced availability during a high-volume traffic spike that is capable of overloading all available healthy application instances. By decreasing the startup time of applications to just seconds, newly scheduled instances are able to more quickly respond to unpredicted spikes in traffic without decreasing availability or performance.

Dev/Prod Parity

KEEP DEVELOPMENT, STAGING, AND PRODUCTION AS SIMILAR AS POSSIBLE

The twelve-factor application should prevent divergence between development and production environments. There are three types of gaps to be mindful of.

Time Gap: Developers should expect development changes to be quickly deployed into production.

Personnel Gap: Developers who make a code change are closely involved with its deployment into production, and closely monitor the behavior of an application after making changes.

Tools Gap: Each environment should mirror technology and framework choices in order to limit unexpected behavior due to small inconsistencies

Logs

TREAT LOGS AS EVENT STREAMS

Twelve-factor apps write logs as an ordered event stream to stout. Applications should not attempt to manage the storage of their own logfiles. The collection and archival of log output for an application should instead be handled by the execution environment.

Admin Processes

RUN ADMIN/MANAGEMENT TASKS AS ONE-OFF PROCESSES

It sometimes becomes the case that developers of an application need to run one-off administrative tasks. These kinds of tasks could include database migrations or running one-time scripts that have been checked in to the application's source code repository. They are considered to be admin processes. Admin processes should be run in the execution environment of an application, with scripts checked into the repository to maintain consistency between environments.

Components of Microservices

- There are the following components of microservices:
- Spring Cloud Config Server
- Netflix Eureka Naming Server
- Hystrix Server
- Netflix ZuulAPI Gateway Server
- Netflix Ribbon
- Zipkin Distributed Tracing Server
- Spring Cloud Admin Server

Components of Microservices

- **Spring Cloud Config Server**

Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation **@EnableConfigServer**.

- **Netflix Eureka Naming Server**

Netflix server for service discovery. The Eureka client also balances the client requests. Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called **Eureka Client**. Eureka client interacts with the Eureka

Components of Microservices

- **Hystrix Server**

Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the **Circuit Breaker mechanism**. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.

- **Netflix Zuul API Gateway Server**

Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.

Components of Microservices

- **Netflix Ribbon**
- Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing:
- Load balancing
- Fault tolerance
- Multiple protocols(HTTP, TCP, UDP)
- Caching and Batching
- Client-Side Load Balancing: The client holds the list of server's IPs so that it can deliver the requests. The client selects an IP from the list, randomly, and forwards the request to the server.

Components of Microservices

- **Zipkin Distributed Server**

Zipkin is an open-source project m project. That provides a mechanism for sending, receiving, and visualization traces.

One thing you need to be focused on that is port number.

Application	Port
Spring Cloud Config Server	8888
Netflix Eureka Naming Server	8761
Netflix Zuul API gateway Server	8765
Zipkin distributed Tracing Server	9411

Using Feign REST Client for Service Invocation

The Feign is a declarative web service (HTTP client) developed by **Netflix**. Its aim is to simplify the HTTP API clients. It is a Java to HTTP client binder. If you want to use Feign, create an interface, and annotate it. It provides pluggable annotation support, including Feign annotations and JAX-RS annotations.

It is a library for creating REST API clients. It makes web service clients easier. The developers can use declarative annotations to call the REST services instead of writing representative boilerplate code.

Spring Cloud OpenFeign

Spring Cloud OpenFeign provides OpenFeign integrations for Spring Boot apps through auto-configuration and binding to the Spring Environment. Without Feign, in Spring Boot application, we use **RestTemplate** to call the User service. To use the Feign, we need to add **spring-cloud-starter-openfeign** dependency in the pom.xml file.

Let’s simplify the previously developed code using Spring Cloud OpenFeign.

In the previous section, one of the things that we had already encountered is when we were developing currency-conversion-service; how difficult it was to call the REST service. There is a lot of manuals that we have to do to call a very simple service. But still we have written a lot of code for it.

When we work with microservices, there will be a lot of calls to other services. We need not to code like the previous one. Feign is a component that solves this problem. Feign makes it easy to invoke other microservices.

The other additional thing that Feign provides is: it integrates with the **Ribbon** (client-side load balancing framework).

Let's implement the Feign in our project and invoke other microservices using Feign.

What is an API Gateway?

- An API stands for Application Program Interface. It is a set of instructions, protocols, and tools for building software applications. It specifies how software components should interact.
- The API Gateway is a server. It is a single entry point into a system. API Gateway encapsulates the internal system architecture. It provides an API that is tailored to each client. It also has other responsibilities such as **authentication, monitoring, load balancing, caching, request shaping and management,** and **static response handling**.
- API Gateway is also responsible for **request routing, composition,** and **protocol translation**. All the requests made by the client go through the API Gateway. After that, the API Gateway routes requests to the appropriate microservice.

What is an API Gateway?

- The API Gateway handles the request in one of the two ways:
- It routed or proxied the requests to the appropriate service.
- Fanning out (spread) a request to multiple services.
- The API Gateway can provide each client with a custom API. It also translates between two protocols, such as **HTTP, WebSockets,** and **Web-Unfriendly** protocols that are used internally.
- **Example**
- The popular example of API Gateway is **Netflix API Gateway**. The Netflix streaming services are available on hundreds of different kinds of devices such as **televisions**, **set-top boxes, smartphones, tablets,** etc. It attempts to provide a **one-size-fits-all** API for its streaming service.

What is an API Gateway?

- An API Gateway includes:
- Security
- Caching
- API composition and processing
- Managing access quotas
- API health monitoring
- Versioning
- Routing

Working of API Gateway

- In microservices, we route all the requests through an API. We can implement common features like **authentication, routing, rate limiting, auditing,** and **logging** in the API Gateway.
- Consider a scenario in which we do not want to call a microservice more than five times by a particular client. We can do it as a part of the limit in the API Gateway. We can implement the common features across microservices in the API gateway. The **Zuul API Gateway** is a popular API Gateway implementation.
- We must implement the following features in all the microservices:
- **Service Aggregation**
- **Authentication, authorization and Security**
- **Rate Limits**
- **Fault Tolerance**

Working of API Gateway

- Suppose there is an external consumer who wants to call **fifteen** different services as part of one process. It is better to aggregate those fifteen services and provide one service call for the external consumer. These are the kinds of features that are common across all the microservices. These features are implemented at the level of API.
- Instead of allowing microservices to call each other directly, we would do all the calls through API Gateway. API Gateway will take care of common features like authentication, fault tolerance, etc. It also provides aggregation services around all microservices because all calls get routed through the API Gateway.

Distributed Tracing

- Introduction to Distributed Tracing
- Distributed tracing is a technique to **monitor** and **profile** the applications, especially those built using microservice architecture. It is also known as **distributed request tracing**. Developers use distributed tracing to **debug** and **optimize** the code.
- Distribute tracing provides a place where we can see that "what is happening with a specific request?" It is important because there are a variety of components that are involved in the microservices.
- If we want to solve a problem or debug a problem, we need a centralized server. So the term **distributed tracing** comes into existence.

Distributed Tracing

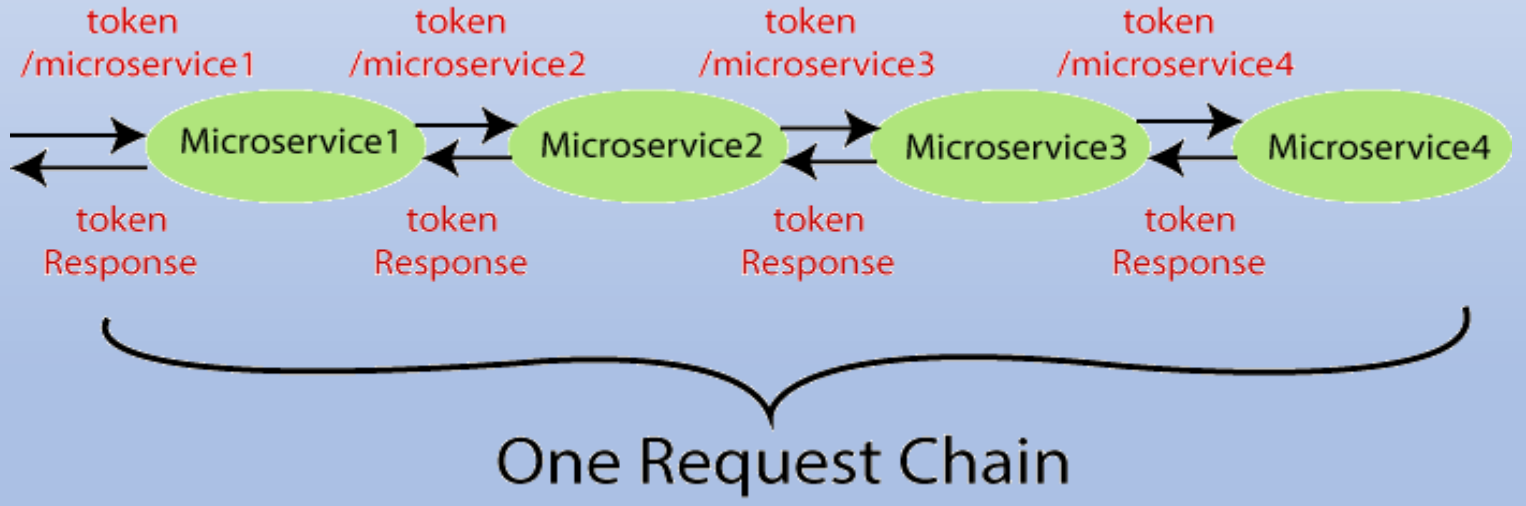
- **Spring Cloud Sleuth:**

Spring Cloud Sleuth is a **Spring Cloud library** that provides the ability to track the progress of subsequent microservices by adding **trace** and **span Ids** on the appropriate HTTP request headers. The Sleuth library is based on the **MDC** (Mapped Diagnostic Context) concept, where we can easily extract values, put to context, and display them in the log.

- **Zipkin**:
- Zipkin is an open-source, Java-based distributed tracing system. It has a management console that provides a mechanism for sending, receiving, storing, and visualizing traces details of the subsequent services.
- With the help of the Zipkin server, we can put all the logs of all the components in the MQ (RabbitMQ). We send the logs to the Zipkin server where the logs consolidate. After doing this, we can monitor different requests. We can also find what is happening to a specific request?

Implementing distributed tracing Using Spring Cloud Sleuth

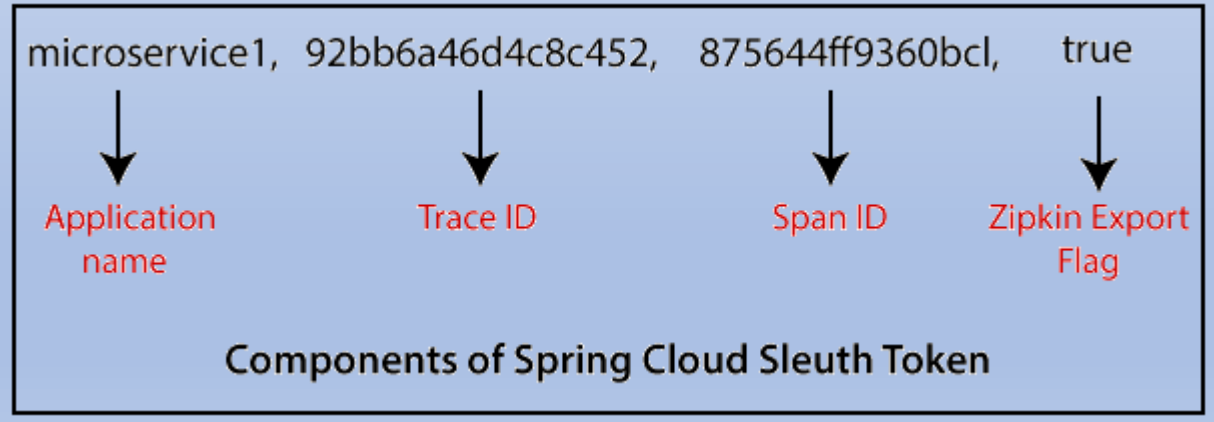
- In this step, we will add Spring Cloud Sleuth for all the microservices. It adds a unique Id to all the requests. It is used to generate and attach the trace Id, span Id to the logs so that tools (Zipkin) can use these ids.



Implementing distributed tracing Using Spring Cloud Sleuth

- The Spring Cloud Sleuth token has the following components:
- **Application name:** The name of the application that is defined in the **properties** file.
- **Trace Id:** The Sleuth adds the Trace Id. It remains the same in all services for a given request.
- **Span Id:** The Sleuth also adds the Span Id. It remains the same in a unit of work but different for different services for a given request.
- **Zipkin Export Flag:** It indicates a boolean value. It can be either **true** or

The following figure shows the Spring Cloud Sleuth token.



Implementing distributed tracing Using Spring Cloud Sleuth

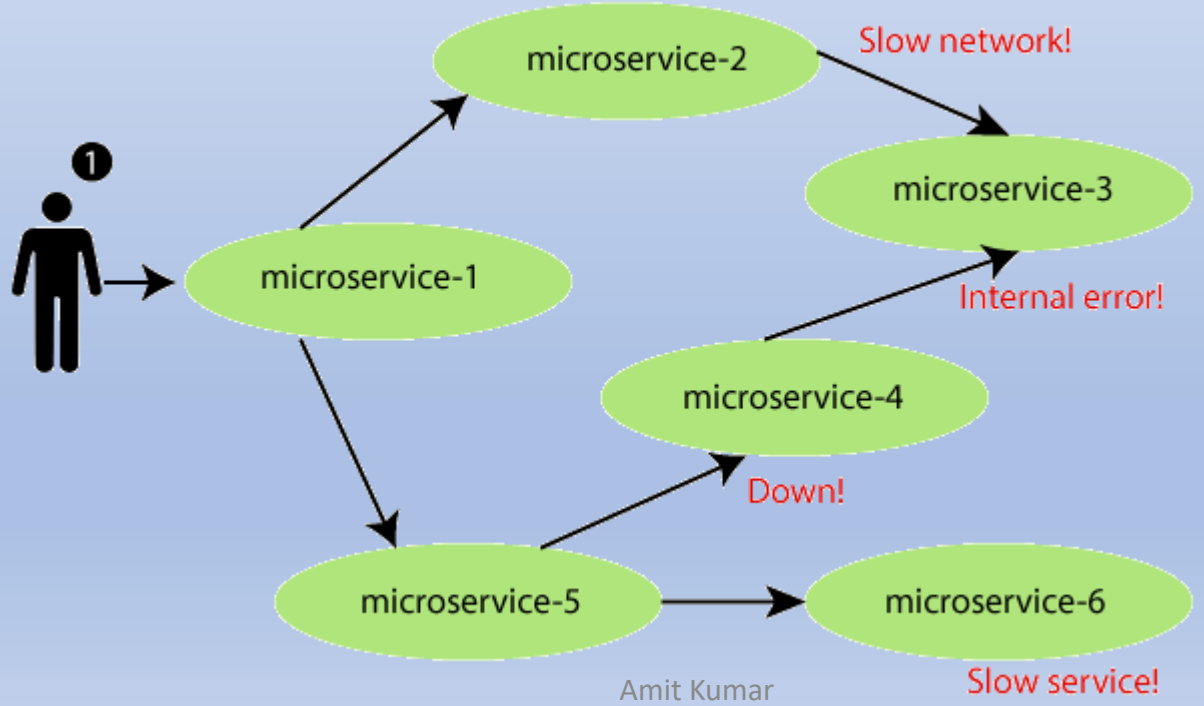
- **Sampler**
- Distributed tracing may have a very high volume of data, so the sampling is important in distributed tracing. Spring Cloud Sleuth provides a **Sampler** strategy. With the help of Sampler, we can implement the sampling algorithm that provides control of the algorithm. By default, we get a procedure that continuously performs the tracing if a **span (correlation: is an individual operation)** is already active.
- But the newly created spans are always marked as **non-exportable**. If all the applications are running with the Sampler, we can see trace (end-to-end latency graph that is composed of spans) in the log, not in any remote location. By default, Spring Cloud Sleuth sets all spans to **non-exportable**.
- When we export span data to the **Zipkin** or **Spring Cloud Stream**, Spring Cloud Sleuth provides **AlwaysSampler** class that exports everything to the Zipkin. It also provides a **PercentageBasedSampler** class that samples a fixed fraction of span.

Fault Tolerance with Hystrix

- Microservices must be extremely reliable because they depend on each other. The microservice architecture contains a large number of small microservices. These microservices communicate with each other in order to fulfill their requirements.
- The instances of microservices may go up and down frequently. **As the number of interactions between microservices increases, the chances of failure of the microservice also increases in the system.**
- Consider a scenario in which six microservices are communicating with each other. The **microservice-5** becomes down at some point, and all the other microservices are directly or indirectly depend on it, so all other services also go down.
- The solution to this problem is to use a **fallback** in case of failure of a microservice. This aspect of a microservice is called **fault tolerance**.

Fault Tolerance with Hystrix

- **Fault tolerance** can be achieved with the help of a **circuit breaker**. It is a pattern that wraps requests to external services and detects when they fail. If a failure is detected, the circuit breaker opens. All the subsequent requests immediately return an error instead of making requests to the unhealthy service. It monitors and detects the service which is down and misbehaves with other services. It rejects calls until it becomes healthy again.



Api Gateway

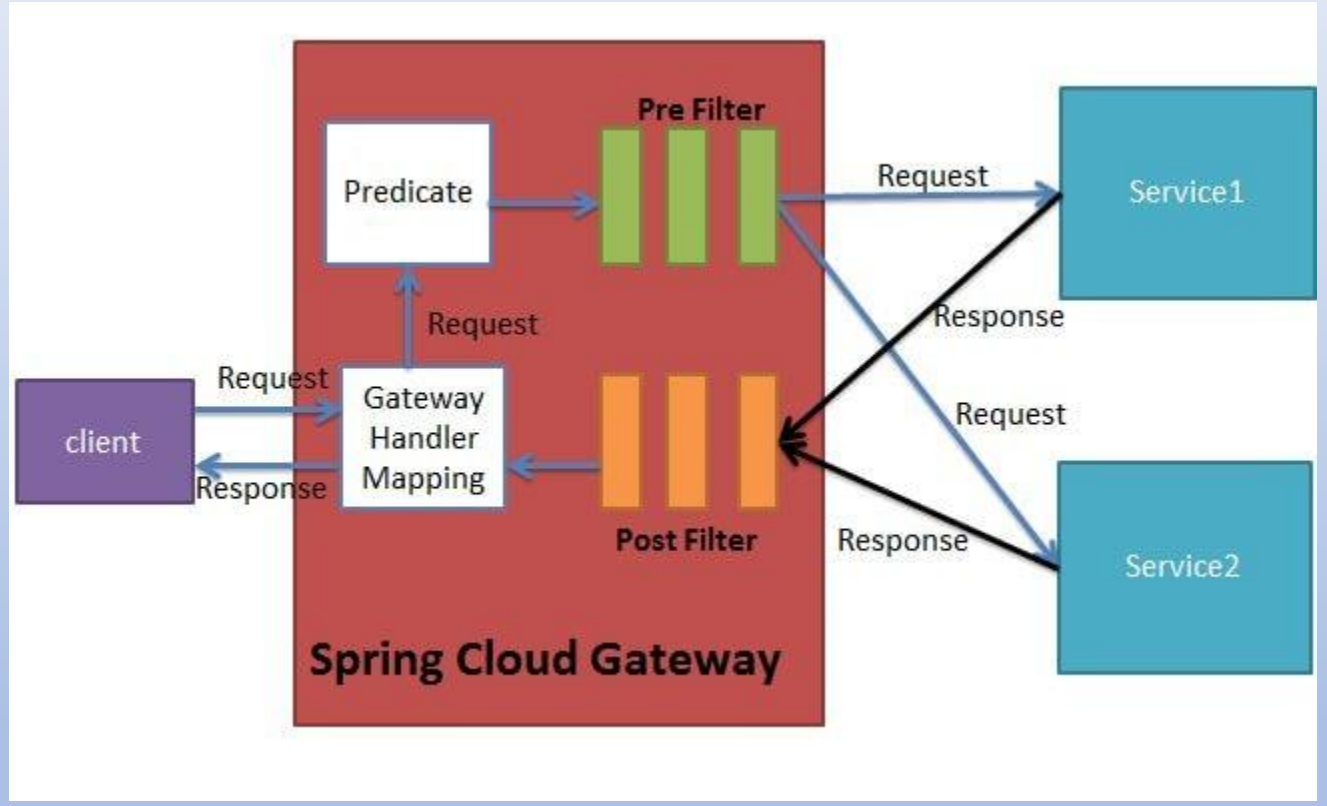
The advantages of this approach are as follows-This improves the security of the microservices as we limit the access of external calls to all our services.

The cross cutting concerns like authentication, monitoring/metrics, and resiliency will be needed to be implemented only in the API Gateway as all our calls will be routed through it.

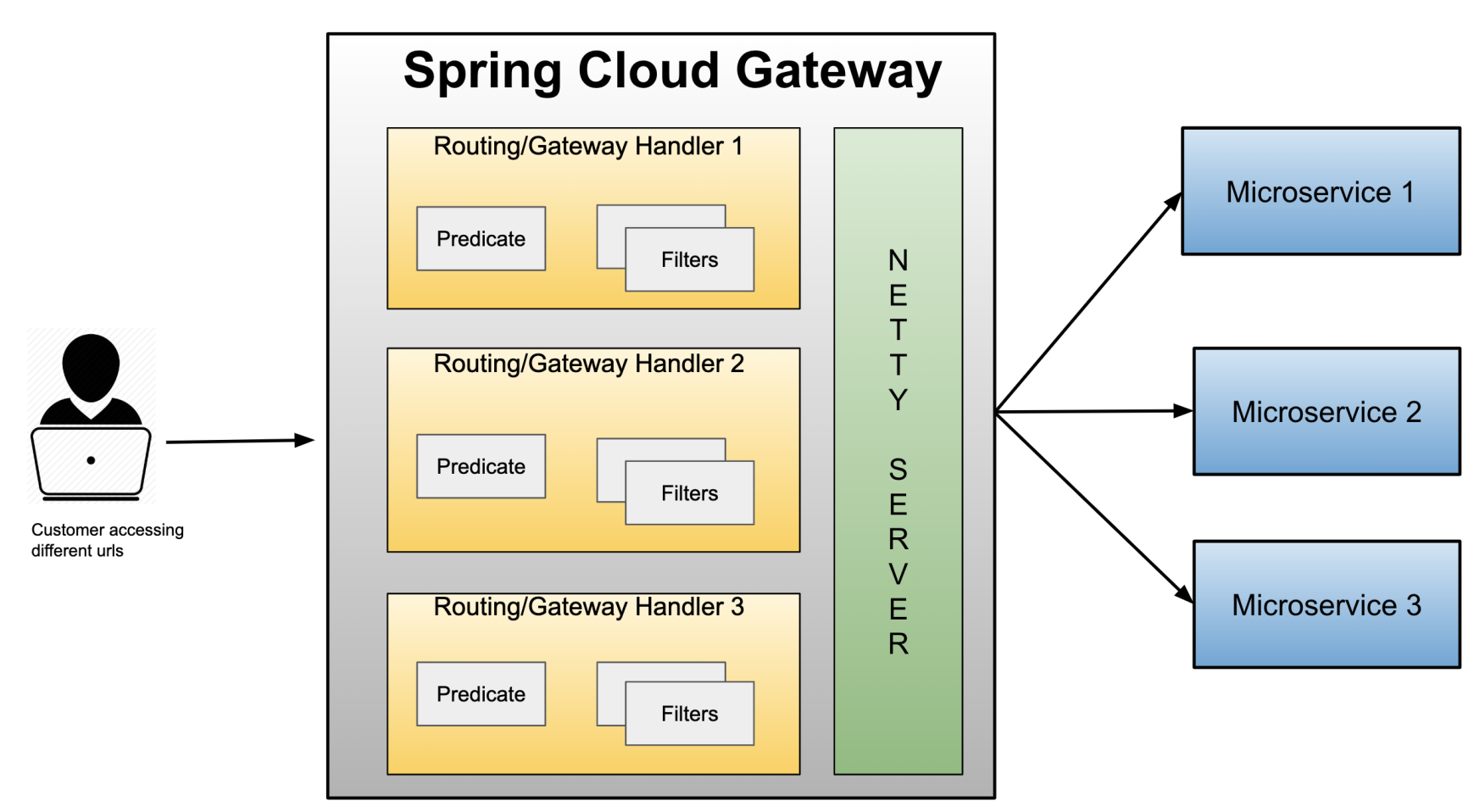
The client does not know about the internal architecture of our microservices system. Client will not be able to determine the location of the microservice instances.

Simplifies client interaction as he will need to access only a single service for all the requirements.

Api Gateway


Pre Filter
Predicate
Request
Service1
Response
Request
Request
Gateway
Handler
Mapping
client
Request
Response
Post Filter
Response
Service2
Spring Cloud Gateway

Api Gateway

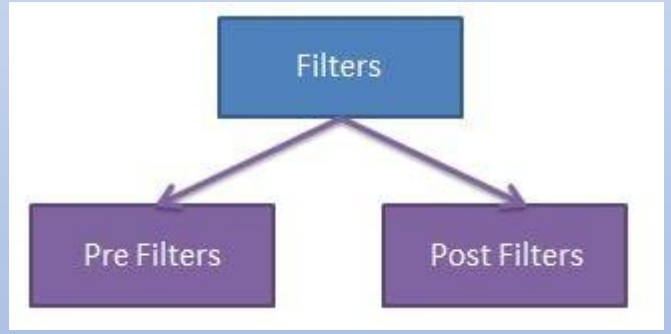

Spring Cloud Gateway
Routing/Gateway Handler 1
Predicate
Filters
Routing/Gateway Handler 2
Predicate
Filters
Routing/Gateway Handler 3
Predicate
Filters
NETTY SERVER
Customer accessing different urls
Microservice 1
Microservice 2
Microservice 3

Api Gateway

Using Predicates Spring Cloud Gateway determines which route should get called. Once decided the request is the routed to the intended microservice. Before routing this request we can apply some filters to the request. These filters are known as pre filters. After applying the filters the intended micoservice call is made and the response is returned back to the Spring Cloud Gateway which returns this response back to the caller. Before returning the response we can again apply some filters to this response. Such filters are called post filters.

Api Gateway

- **Implementing Spring Cloud Gateway Filters**
- Spring Cloud Gateway filters can be classified asSpring Cloud Gateway Pre Filters
- Spring Cloud Gateway Post Filters



Service Mesh

Microservices have taken the software industry by storm and rightly so. Transitioning from a monolith to a microservices architecture enables you to deploy your application more frequently, independently, and reliably.

However, everything is not green in a Microservice architecture, and it has to deal with the same problems encountered while designing distributed systems.

On this note, why not recap the Eight Fallacies of Distributed Computing:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology does not Change
- There is one Administrator
- Transport cost is zero
- The network is homogenous

Service Mesh

With a microservices architecture, a dependency on a network comes in and raises reliability questions. As the number of services increase, you have to deal with the interactions between them, monitor the overall system health, be fault tolerant, have logging and telemetry in place, handle multiple points of failure, and more. Each of the services needs to have these common functionalities in place so that the service to service communication is smooth and reliable. But this sounds like lot of effort if you have to deal with dozens of microservices, does it not?

Solution:

Service Mesh

Service Mesh

What Is a Service Mesh?

A service mesh can be defined as an infrastructure layer which handles the inter-service communication in a microservice architecture. Service mesh reduces the complexity associated with a microservice architecture and provides lot of the functionalities like:

- Load balancing
- Service discovery
- Health checks
- Authentication
- Traffic management and routing
- Circuit breaking and failover policy
- Security
- Metrics and telemetry
- Fault injection

Service Mesh

Why Is Service Mesh Necessary?

In a microservice architecture, handling service to service communication is challenging and most of the time we depend upon third-party libraries or components to provide functionalities like service discovery, load balancing, circuit breaker, metrics, telemetry, and more. Companies like Netflix came up with their own libraries like Hystrix for circuit breakers, Eureka for service discovery, Ribbon for load balancing which are popular and widely used by organizations.

However, these components need to be configured inside your application code and based on the language you are using, the implementation will vary a bit. Anytime these external components are upgraded, you need to update your application, verify it, and deploy the changes. This also creates an issue where now your application code is a mixture of business functionalities and these additional configurations. Needless to say, this tight coupling increases the overall application complexity, since the developer now needs to also understand how these components are configured so that he/she can troubleshoot in case of any issues.

Service Mesh

Service Mesh comes to the rescue here. It decouples this complexity from your application and puts it in a service proxy and lets it handle it for you. These service proxies can provide you with a bunch of functionalities like traffic management, circuit breaking, service discovery, authentication, monitoring, security, and much more. Hence from an application standpoint, all it contains is the implementation of business functionalities.

Say, in your microservice architecture, if you have five services talking with each other. Then, instead of building the common necessary functionalities like configuration, routing, telemetry, logging, circuit breaking, etc. inside every microservice, it makes more sense to abstract it into a separate component — called a 'service proxy.'

With the introduction of service mesh, there is a clear segregation of responsibilities. This makes the lives of developers easier. If there is an issue, developers can easily identify the root cause based on whether it is application or network related.

Service Mesh

How Is Service Mesh Implemented?

To implement a service mesh, you can deploy a proxy alongside your services. This is also known as the sidecar pattern.

The sidecars abstract the complexity away from the application and handle the functionalities like service discovery, traffic management, load balancing, circuit breaking, etc.

Envoy from Lyft is the most popular open source proxy designed for cloud native applications. Envoy runs along side every service and provides the necessary features in a platform agnostic manner. All traffic to your service flows through the Envoy proxy.

Service Mesh

What Is Istio?

Istio is an open platform to connect, manage, and secure microservices. It is very popular in the Kubernetes community and is getting widely adopted.

Istio provides additional capabilities in your microservices architecture like intelligent routing, load balancing, service discovery, policy enforcement, in-depth telemetry, circuit breaking and retry functionalities, logging, monitoring, and more.

Istio is one of the best implementations of a service mesh at this point. It enables you to deploy microservices without an in-depth knowledge of the underlying infrastructure.

As more and more organizations start breaking down their monoliths into a microservice architecture, they will reach a point where managing the increasing number of services becomes a burden. Service mesh comes to the rescue in such scenarios and abstracts away all the complexities without the need to make any changes to the application.

Service Mesh

What Is Istio?

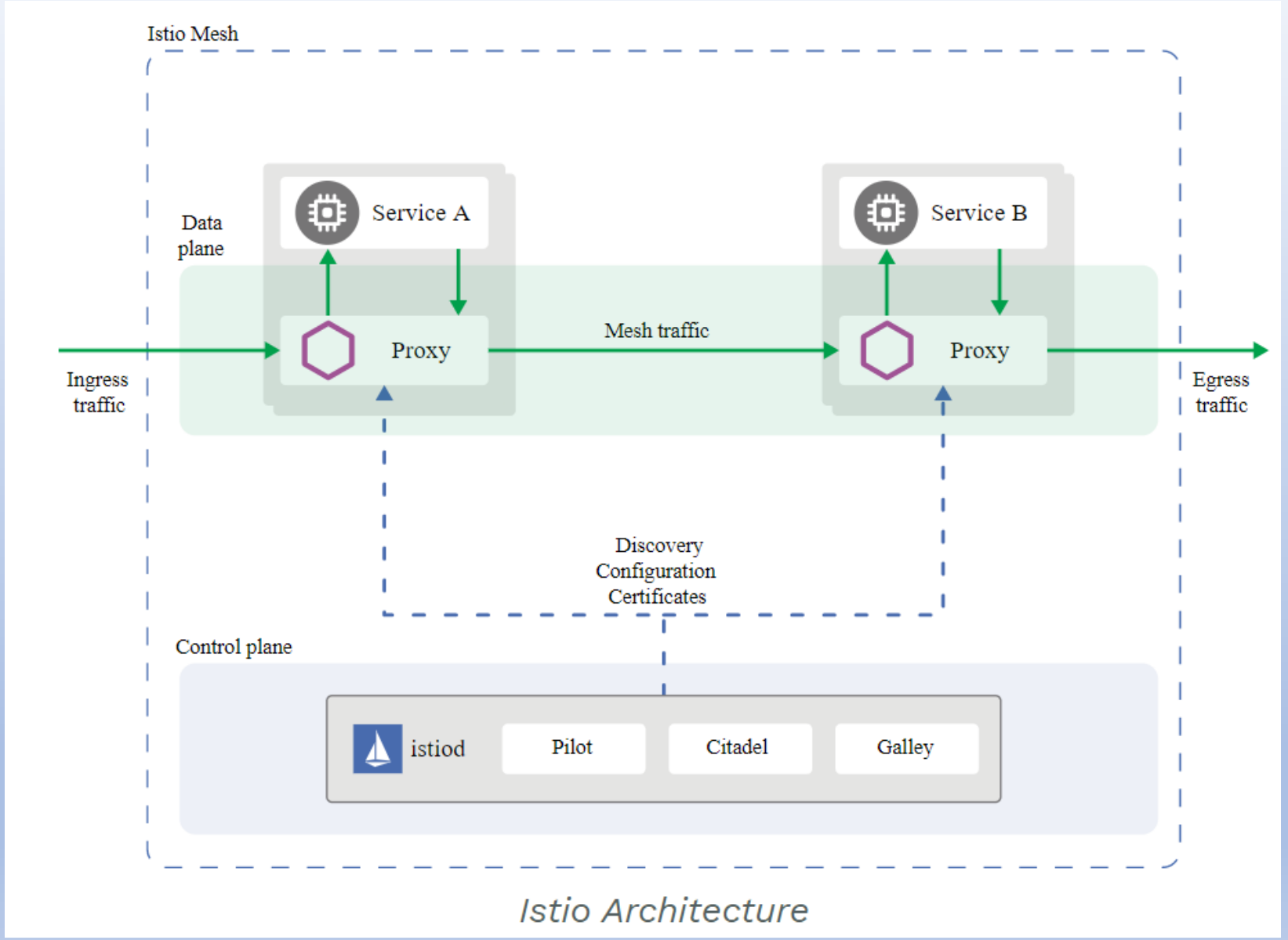
Istio is an open platform to connect, manage, and secure microservices. It is very popular in the Kubernetes community and is getting widely adopted.

Istio provides additional capabilities in your microservices architecture like intelligent routing, load balancing, service discovery, policy enforcement, in-depth telemetry, circuit breaking and retry functionalities, logging, monitoring, and more.

Istio is one of the best implementations of a service mesh at this point. It enables you to deploy microservices without an in-depth knowledge of the underlying infrastructure.

As more and more organizations start breaking down their monoliths into a microservice architecture, they will reach a point where managing the increasing number of services becomes a burden. Service mesh comes to the rescue in such scenarios and abstracts away all the complexities without the need to make any changes to the application.

Service Mesh



Istio Architecture

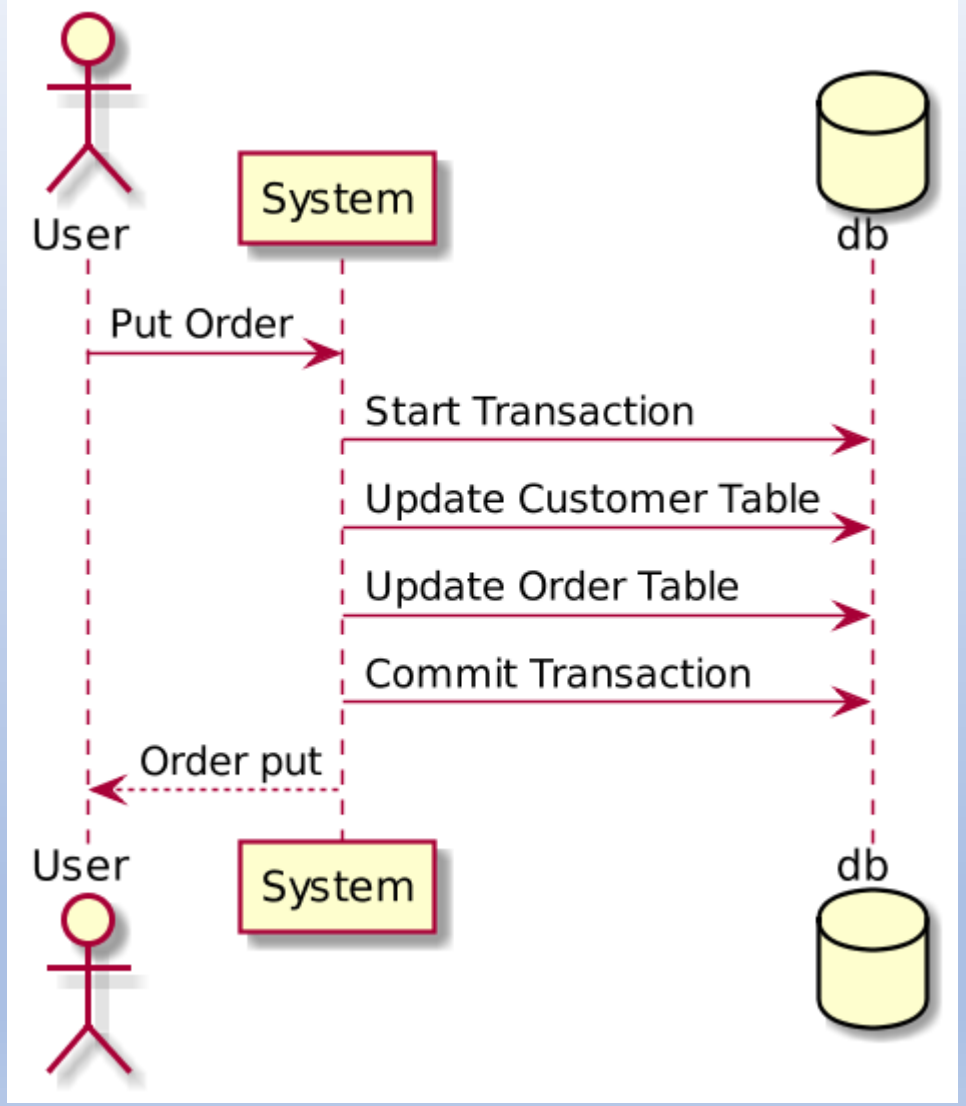
Distributed Transaction

Microservices architecture (MSA) has become very popular.. However, one common problem is how to manage distributed transactions across multiple microservices. This post is going to share my experience from past projects and explain the problem and possible patterns that could solve it.

When a microservice architecture decomposes a monolithic system into self-encapsulated services, it can break transactions. This means a local transaction in the monolithic system is now distributed into multiple services that will be called in a sequence.

Distributed Transaction

Here is a customer order example with a monolithic system using a local transaction:



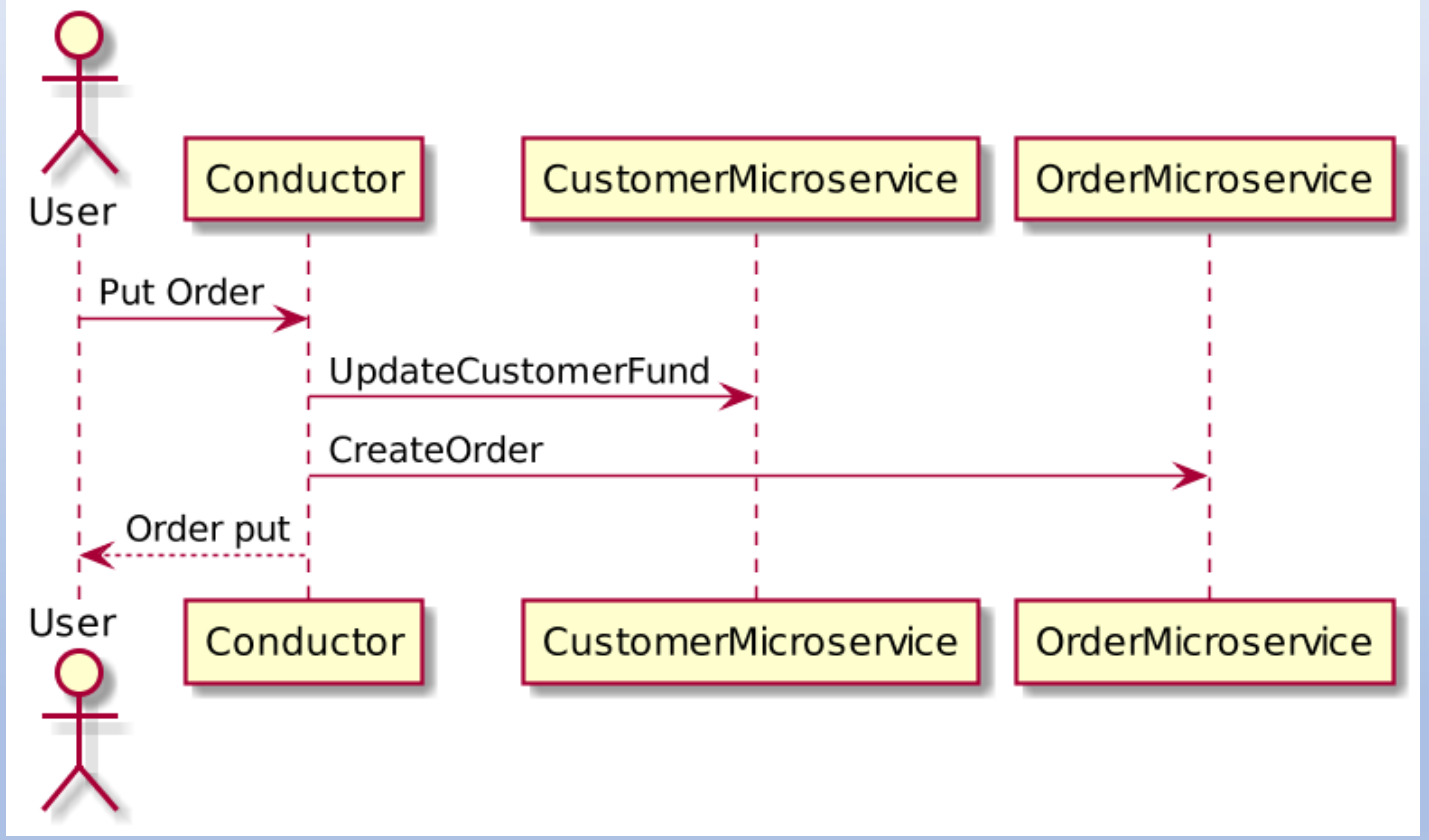
Distributed Transaction

In the customer order example above, if a user sends a Put Order action to a monolithic system, the system will create a local database transaction that works over multiple database tables. If any step fails, the transaction can roll back. This is known as ACID (Atomicity, Consistency, Isolation, Durability), which is guaranteed by the database system.

When we decompose this system, we created both the CustomerMicroservice and the OrderMicroservice, which have separate databases. Here is a customer order example with microservices:

Distributed Transaction

When a Put Order request comes from the user, both microservices will be called to apply changes into their own database. Because the transaction is now across multiple databases, it is now considered a distributed transaction.



Distributed Transaction

Problem:

How do we keep the transaction atomic?

-In a database system, atomicity means that in a transaction either all steps complete or no steps complete. The microservice-based system does not have a global transaction coordinator by default. In the example above, if the CreateOrder method fails, how do we roll back the changes we applied by the CustomerMicroservice?

Do we isolate user actions for concurrent requests?

If an object is written by a transaction and at the same time (before the transaction ends), it is read by another request, should the object return old data or updated data? In the example above, once UpdateCustomerFund succeeds but is still waiting for a response from CreateOrder, should requests for the current customer's fund return the updated amount or not?

Distributed Transaction

The problems above are important for microservice-based systems. Otherwise, there is no way to tell if a transaction has completed successfully. The following two patterns can resolve the problem:

2pc (two-phase commit)

Saga

Distributed Transaction

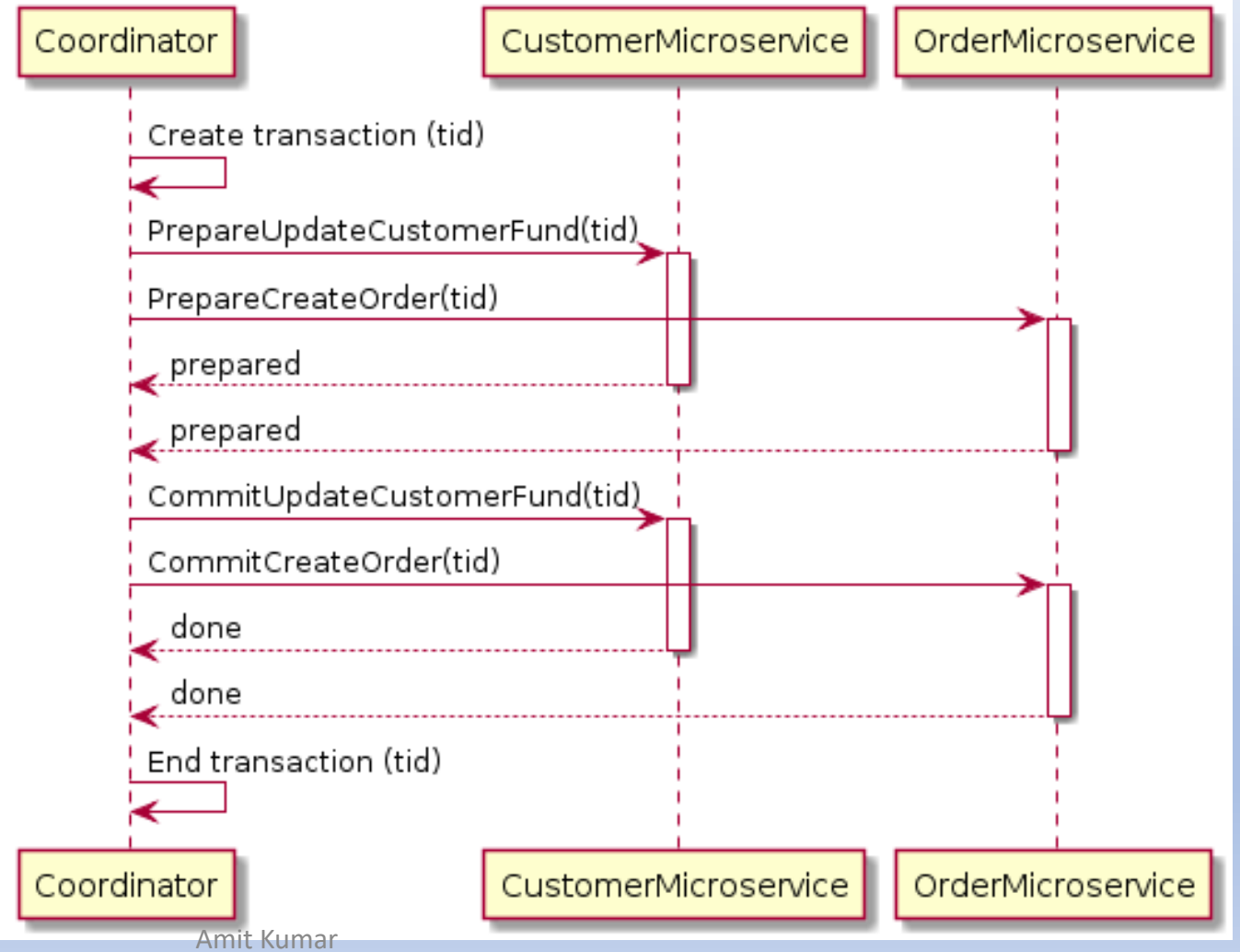
Two Phase Commit:

As its name hints, 2pc has two phases: A prepare phase and a commit phase. In the prepare phase, all microservices will be asked to prepare for some data change that could be done atomically. Once all microservices are prepared, the commit phase will ask all the microservices to make the actual changes.

Normally, there needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases.

Distributed Transaction

Here is a 2pc implementation for the customer order example:



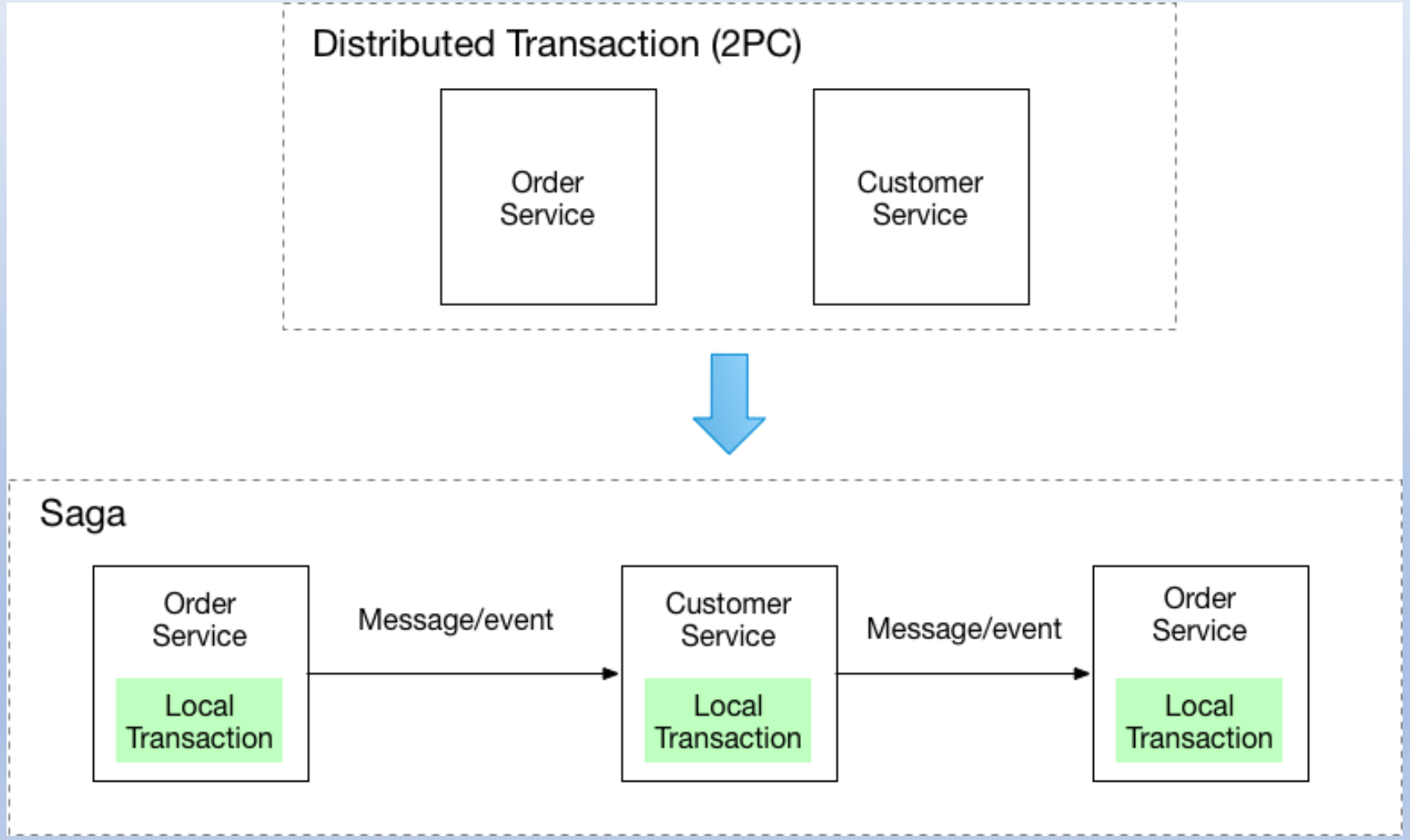
Distributed Transaction

Saga:

Implement each business transaction that spans multiple services is a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Distributed Transaction

Saga vs 2PC



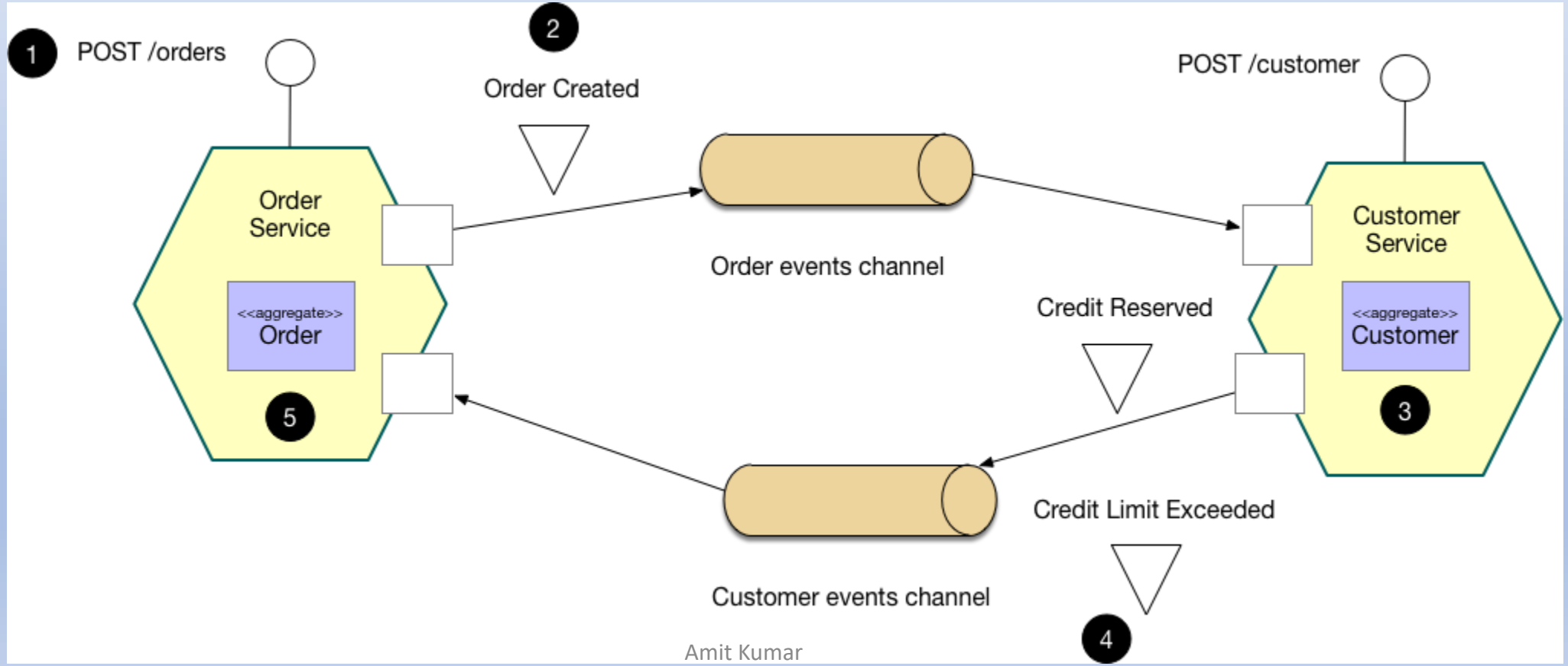
Distributed Transaction

Saga:

- There are two ways of coordination sagas:
- Choreography - each local transaction publishes domain events that trigger local transactions in other services
- Orchestration - an orchestrator (object) tells the participants what local transactions to execute

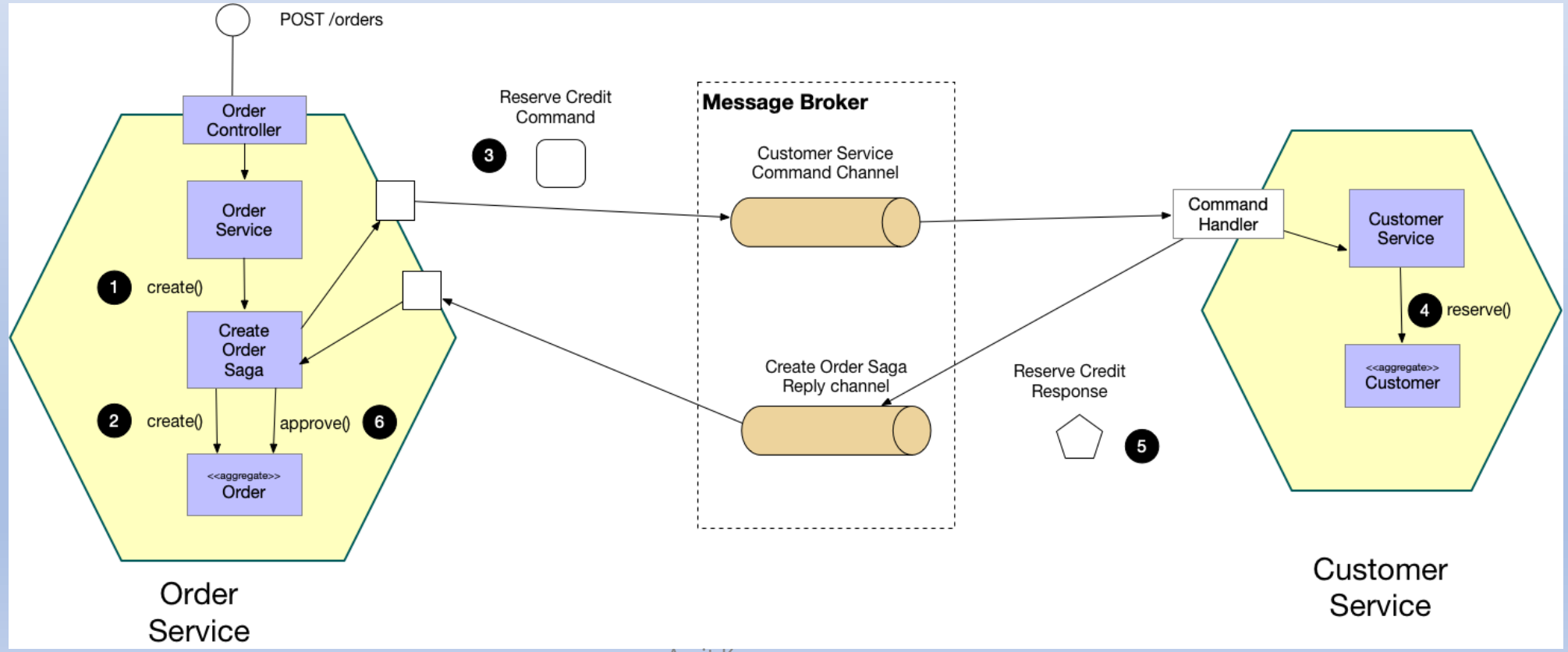
Distributed Transaction

Choreography-based saga

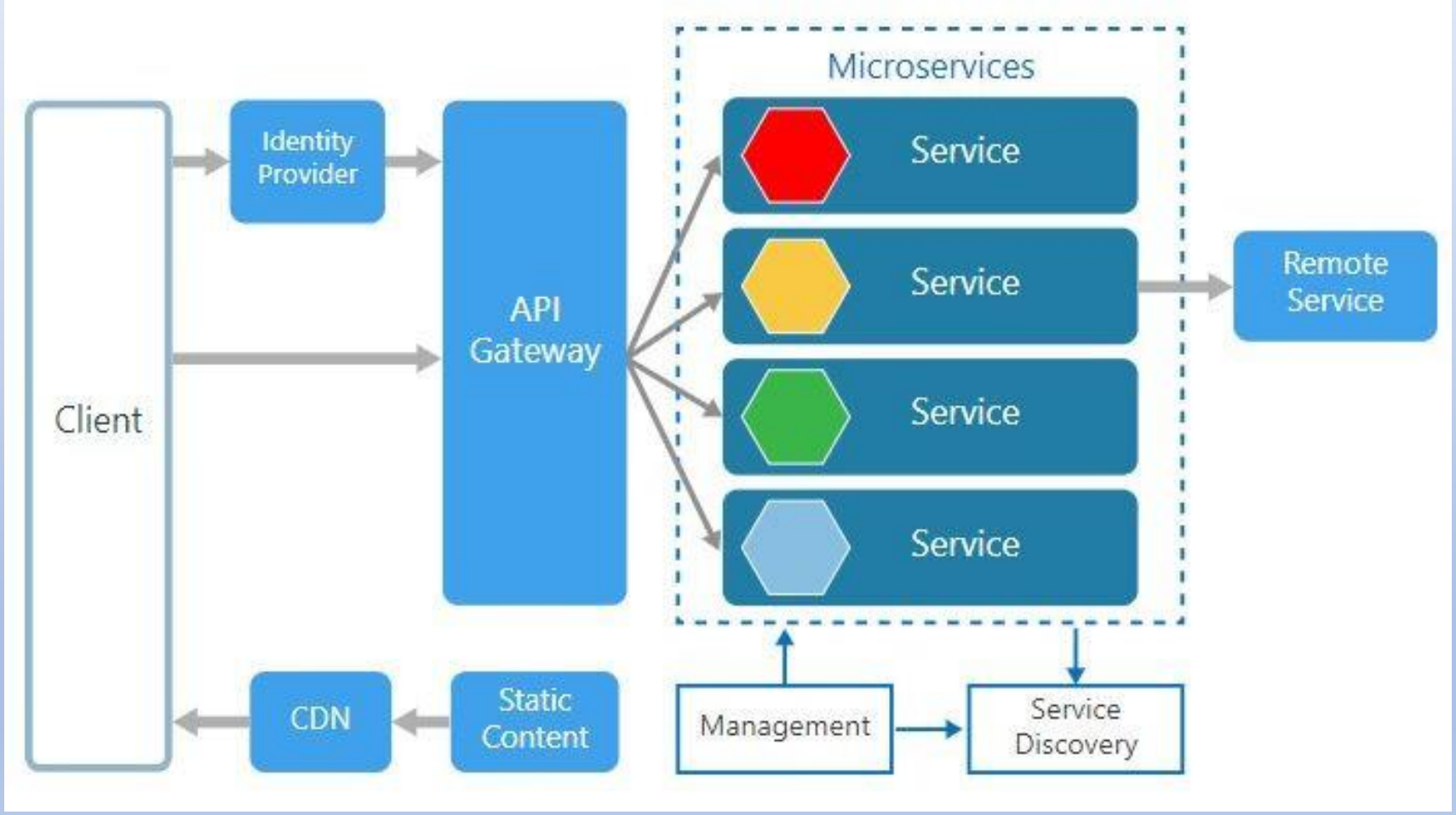


Distributed Transaction

Orchestration-based saga



Architecture



Architecture