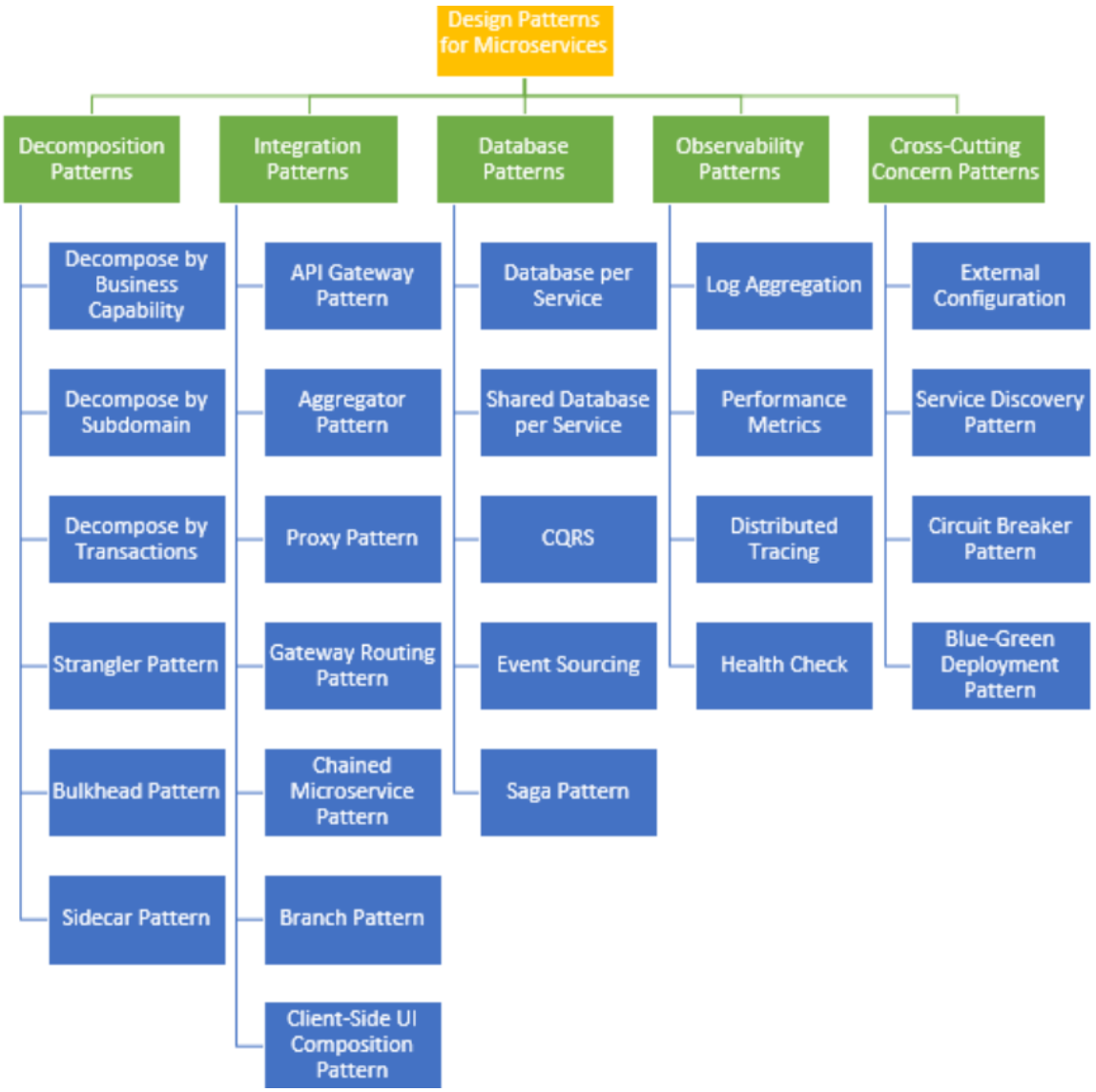
Microservice Design Patterns

Though MSA solves certain problems, it is not a silver bullet.

It has certain drawbacks and when using this architecture, there are numerous issues that must be addressed.

This brings about the need to learn common patterns in these problems and solve them with reusable solutions.

1. Scalability
2. Availability
3. Resiliency
4. Independent, autonomous
5. Decentralized governance
6. Failure isolation
7. Auto-Provisioning
8. Continuous delivery through DevOps



Applying all these principles brings several challenges and issues. Let's discuss those problems and their solutions.

1. Decomposition Patterns

a. Decompose by Business Capability

Problem

Microservices is all about making services loosely coupled, applying the single responsibility principle & cohesive. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

Solution

One strategy is to decompose by business capability. A business capability is something that a business does in order to generate value. For example, the capabilities of an insurance company typically include sales, marketing, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except it's business-oriented rather than technical.

b. Decompose by Subdomain

Problem

Decomposing an application using business capabilities might be a good start, but you will come across so-called "God Classes" which will not be easy to decompose. These classes will be common among multiple services. For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc. How do we decompose them?

Solution

For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue. It uses subdomains and bounded context concepts to solve this problem. DDD breaks the whole domain model created for the enterprise into subdomains. Each subdomain will have a model, and the scope of that model will be called the bounded context. Each microservice will be developed around the bounded context.

Note: Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

Noun, verbs also can be made as separate Services based on their size. For example making Payment by interacting

<<Single Responsibility Principle>>

c. Strangler Pattern

Problem

So far, the design patterns we talked about were decomposing applications, but 80% of the work we do may be with applications, which

are big monolithic applications. Applying all the above design patterns to them will be difficult because breaking them into smaller pieces at the same time it's being used live, is a big task.

Solution

The Strangler pattern comes to the rescue. The Strangler pattern is based on an analogy to a vine that strangles a tree that it's wrapped around. This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services. The idea is to do it one domain at a time.

Eventually, the newly refactored application "strangles" or replaces the original application until finally you can shut off the monolithic application.

Hence Strangler supports incremental refactoring of an application, by gradually replacing specific pieces of functionality with new services.

Side Car Design Pattern:

The sidecar design pattern allows you to add a number of capabilities to your application without additional configuration code for third-party components.

As a sidecar is attached to a motorcycle, similarly in software architecture a sidecar is attached to a parent application and extends/enhances its functionalities. A sidecar is loosely coupled with the main application.

Let me explain this with an example. Imagine that you have six microservices talking with each other in order to determine the cost of a package.

Each microservice needs to have functionalities like observability, monitoring, logging, configuration, circuit breakers, and more. All these functionalities are implemented inside each of these microservices using some industry standard third-party libraries.

2. Integration Patterns

a. API Gateway Pattern

Problem

When an application is broken down to smaller microservices, there are a few concerns that need to be addressed:

1. How to call multiple microservices abstracting producer information.
2. On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
3. Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
4. How to handle different type of Protocols some of which might not be supported by producer microservice.

Solution

An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.
2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.
4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.

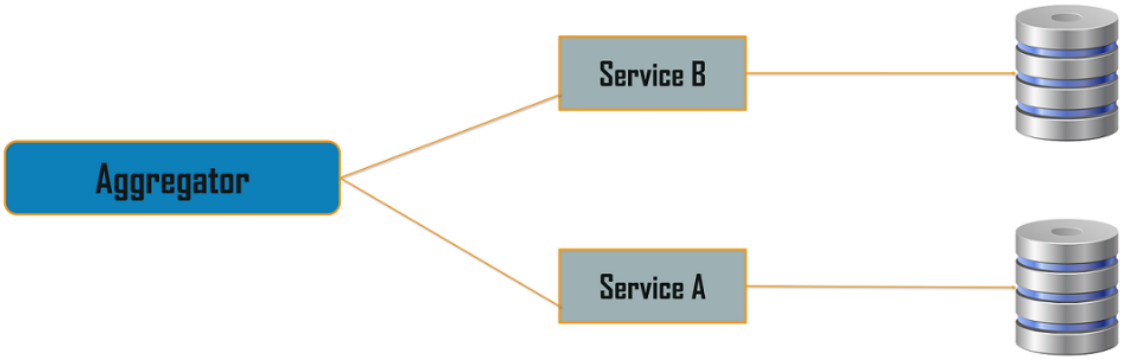
5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
6. It can also offload the authentication/authorization responsibility of the microservice.

b. Aggregator Pattern

Problem

When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

Solution



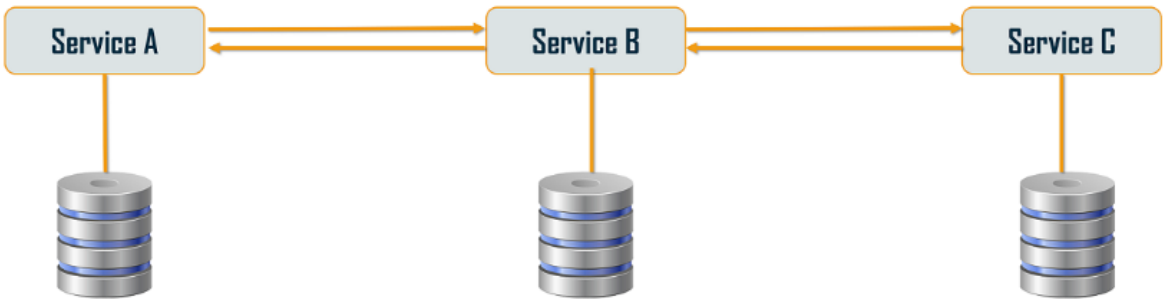
The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

Chained Design Pattern: One Service sends to request to another, which further sends request to next Service in the chain. All these services use synchronous HTTP request or response for messaging. Also, until the request passes through all the services and the respective responses are generated, the client doesn't get any output. So, it is always recommended to not to make a long chain, as the client will wait until the chain is completed

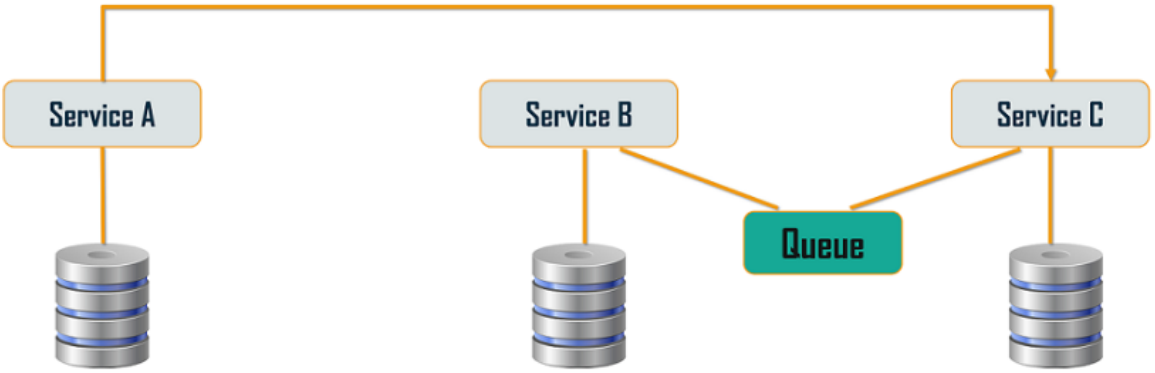


Branching Design Pattern:

Service A interacts with another Service B, after Service B returns Service A invokes another Service C.

Asynchronous Messaging Design Pattern:

if you do not want the consumer, to wait for a long time, then you can opt for the Asynchronous Messaging. In this type of microservices design pattern, all the services can communicate with each other, but they do not have to communicate with each other sequentially. So, if you consider 3 services: Service A, Service B, and Service C. The request from the client can be directly sent to the Service C and Service B simultaneously. These requests will be in a queue. Apart from this, the request can also be sent to Service A whose response need not have to be sent to the same service through which request has come.



c. Client-Side UI Composition Pattern

Problem

When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices. In the monolithic world, there used to be only one call from the UI to a backend service to retrieve all data and refresh/submit the UI page. However, now it won't be the same. We need to understand how to do it.

Solution

With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section(of web app/mobile app) will make a call to an individual backend microservice to pull the data. That is called composing UI components specific to service. Frameworks like AngularJS and ReactJS help to do that easily. These screens are known as Single Page Applications (SPA). This enables the app to refresh a particular region of the screen instead of the whole page.

3. Database Patterns

a. Database per Service

Problem

There is a problem of how to define database architecture for microservices. Following are the concerns to be addressed:

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.

2. Business transactions may enforce invariants that span multiple services.

3. Some business transactions need to query data that is owned by multiple services.

4. Databases must sometimes be replicated and sharded in order to scale.

5. Different services have different data storage requirements.

Solution

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly. For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service. Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

b. Shared Database per Service

Problem

We have talked about one database per service being ideal for microservices, but that is possible when the application is designed ideally. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?

Solution

A shared database per service is not ideal, but that is the working solution for the above scenario. Most people consider this an anti-pattern for microservices, but in certain applications, this is a good start to break the application into smaller logical pieces. This should not be applied for greenfield applications. In this pattern, one database can be aligned with more than one microservice, but it has to be restricted to 2-3 maximum, otherwise scaling, autonomy, and independence will be challenging to execute.

Heterogeneous database(MySQL, MOngo, Cassandra) can be used by different Microservices based on ACID compliant, scalability requirements.

c. Command Query Responsibility Segregation (CQRS)

Problem

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. Then, how do we implement queries in microservice architecture?

Solution

CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views. The **event sourcing pattern** is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

d. Saga Pattern(Distributed Transactions)

Problem

When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

Solution

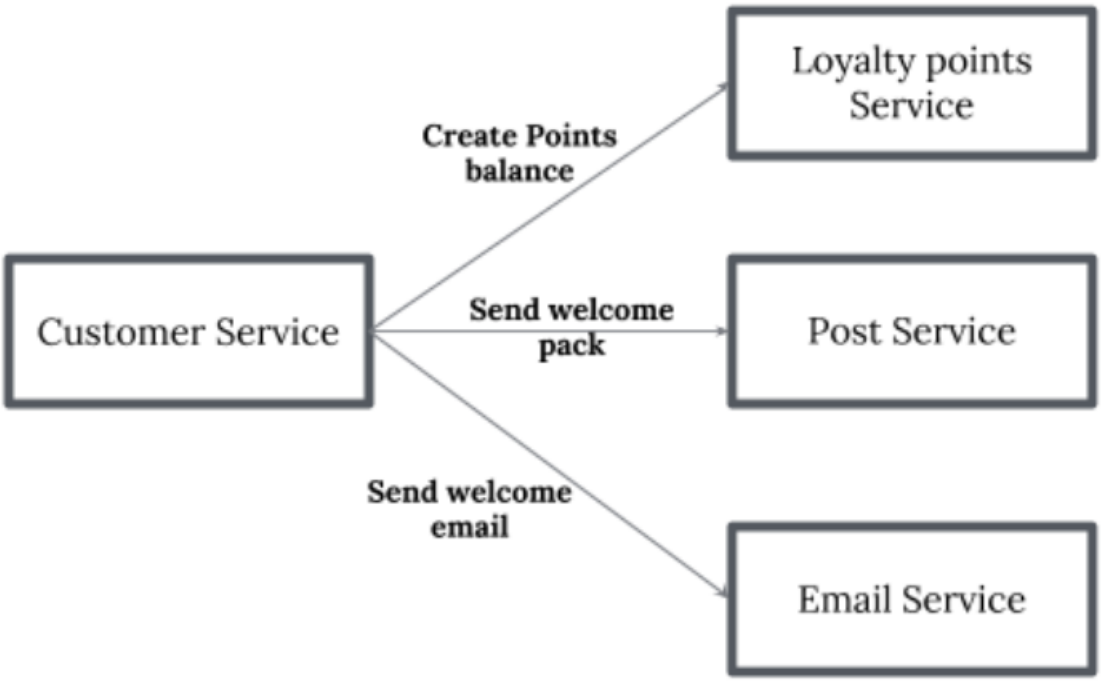
A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

1. Choreography(Each Service's self responsibility) — When there is no central coordination, each service produces and listens to another service's events and decides if an action should be taken or not.

   https://dzone.com/articles/saga-pattern-how-to-implement-business-transaction

2. Orchestration(Orchestrator's responsibility) — An orchestrator (object) takes responsibility for a saga's decision making and sequencing business logic.

   https://dzone.com/articles/saga-pattern-how-to-implement-business-transaction-1



4. Observability Patterns

a. Log Aggregation

Problem

Consider a use case where an application consists of multiple service instances that are running on multiple machines. Requests often span multiple service instances. Each service instance generates a log file in a standardized format. How can we understand the application behavior through logs for a particular request?

Solution

We need a centralized logging service that aggregates logs from each service instance. Users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs. For example, PCF does have Loggeregator, which collects logs from each component (router, controller, diego, etc...) of the PCF platform along with applications. AWS Cloud Watch also does the same.

b. Performance Metrics,

Problem

When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens. How should we collect metrics to monitor application performance?

Solution

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

- Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics(Saas, On Premise)
- Pull — the metrics services pulls metrics from the service e.g. Prometheus - Open Source

c. Distributed Tracing

Problem

In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?

Solution

We need a service which

- Assigns each external request a unique external request id.
- Passes the external request id to all services.
- Includes the external request id in all log messages.
- Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Spring Cloud Sleuth, along with Zipkin server, is a common implementation.

d. Health Check

Problem

When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances? With a load balancing pattern implementation.

Solution

Each service needs to have an endpoint which can be used to check the health of the application, such as `/health`. This API should o check the status of the host, the connection to other services/infrastructure, and any specific logic.

Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well.

JMX

5. Cross-Cutting Concern Patterns

a. External Configuration

Problem

A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some

configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service. How do we avoid code modification for configuration changes?

Solution

Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at startup or on the fly.

Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties. These can be accessed by the application on startup or can be refreshed without a server restart.

Spring Cloud Config

b. Service Discovery Pattern

Problem

When microservices come into the picture, we need to address a few issues in terms of calling services:

1. With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes.
2. Each service URL has to be remembered by the consumer and become tightly coupled.

So how does the consumer or router know all the available service instances and locations?

Solution

A service registry needs to be created which will keep the metadata of each producer service. A service instance should register to the registry when starting and should de-register when shutting down. The consumer or router should query the registry and find out the location of the service. The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it. There are two types of service discovery: client-side and server-side. An example of client-side

discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

c. Circuit Breaker Pattern

Problem

A service generally calls other services to retrieve data, and there is the chance that the downstream service may be down. There are two problems with this: first, the request will keep going to the down service, exhausting network resources and slowing performance. Second, the user experience will be bad and unpredictable. How do we avoid cascading service failures and handle failures gracefully?

Solution

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.

Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

d. Blue-Green Deployment Pattern

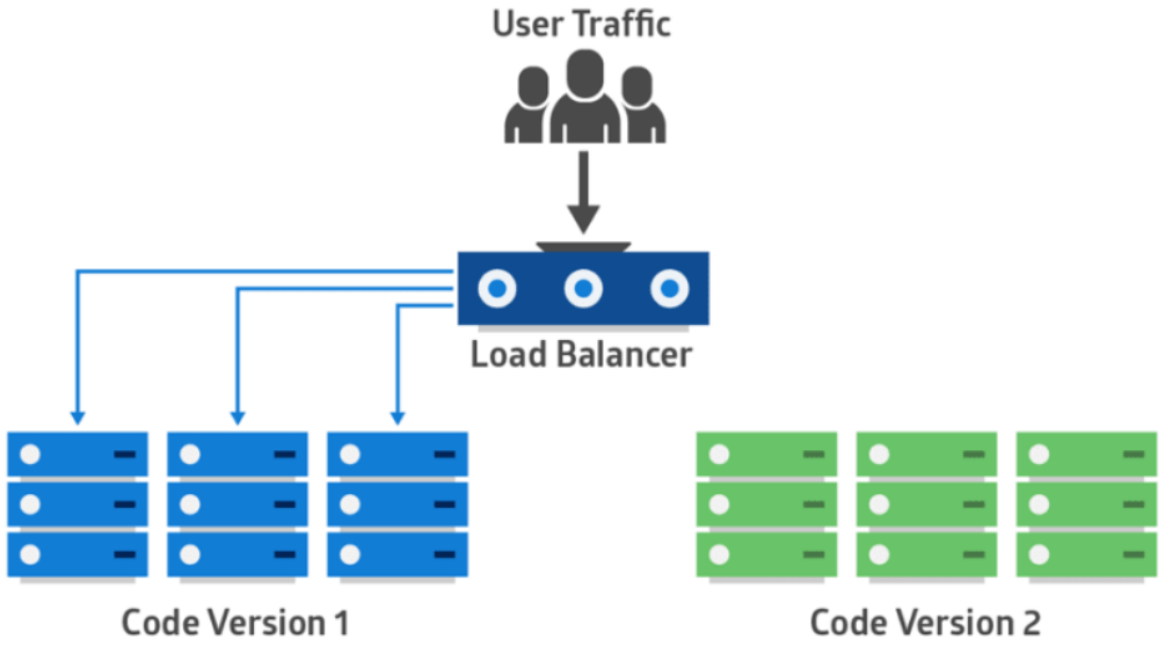
Problem

To deploy enhanced version, if we stop all the services, then downtime will be huge and can impact the business.

Also, the rollback will be a nightmare.

How do we avoid or reduce downtime of the services during deployment?

Solution



The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let's assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic. All cloud platforms provide options for implementing a blue-green deployment. For more details on this topic, check out this article.

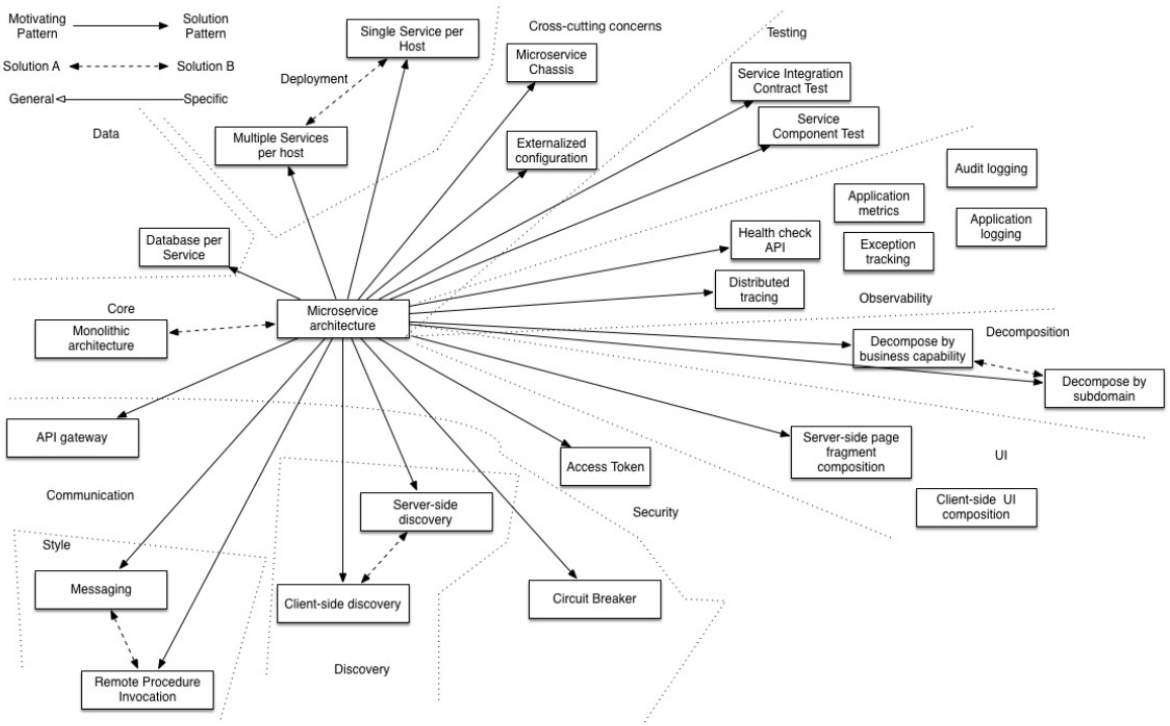
There are many other patterns used with microservice architecture, like Sidecar, Chained Microservice, Branch Microservice, Event Sourcing Pattern, Continuous Delivery Patterns, and more. The list keeps growing as we get more experience with microservices. I am stopping now to hear back from you on what microservice patterns you are using.

Other Misc, Design Patterns:

Caching

Authentication

Contract Testing


Motivating Pattern
Solution Pattern
Solution A
Solution B
General
Specific
Single Service per Host
Cross-cutting concerns
Testing
Deployment
Microservice Chassis
Service Integration Contract Test
Service Component Test
Data
Multiple Services per host
Externalized configuration
Audit logging
Application metrics
Application logging
Database per Service
Health check API
Exception tracking
Distributed tracing
Observability
Core
Microservice architecture
Monolithic architecture
Decomposition
Decompose by business capability
Decompose by subdomain
API gateway
Server-side page fragment composition
UI
Access Token
Client-side UI composition
Communication
Server-side discovery
Security
Style
Messaging
Client-side discovery
Circuit Breaker
Remote Procedure Invocation
Discovery