

Microservice Design Patterns

[Document subtitle]

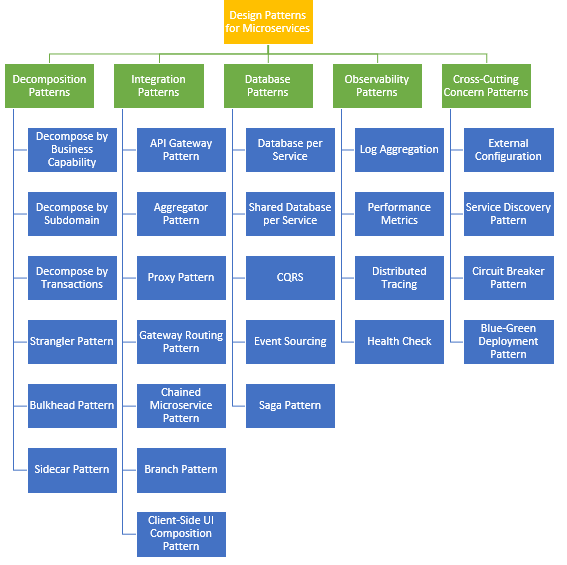


[Date]

[Company name]

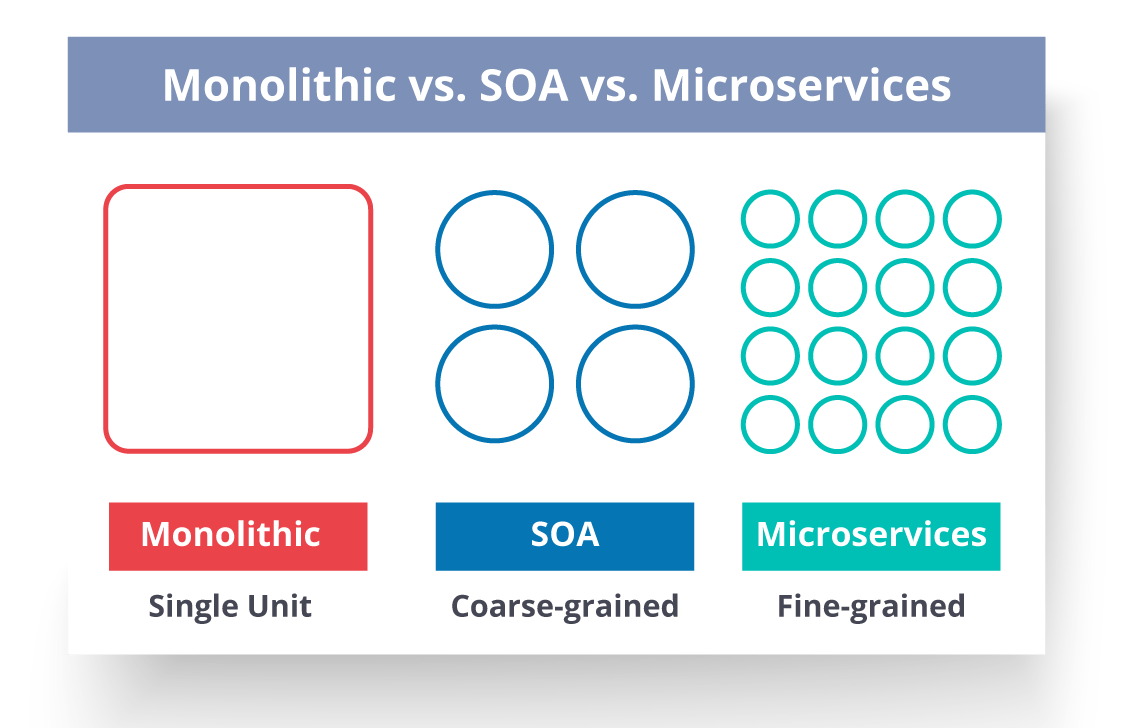
[Company address]

**Design-Patterns**

[](https://user-images.githubusercontent.com/9671419/81451093-592b6600-91a1-11ea-891b-a10b8c5b78cb.png)

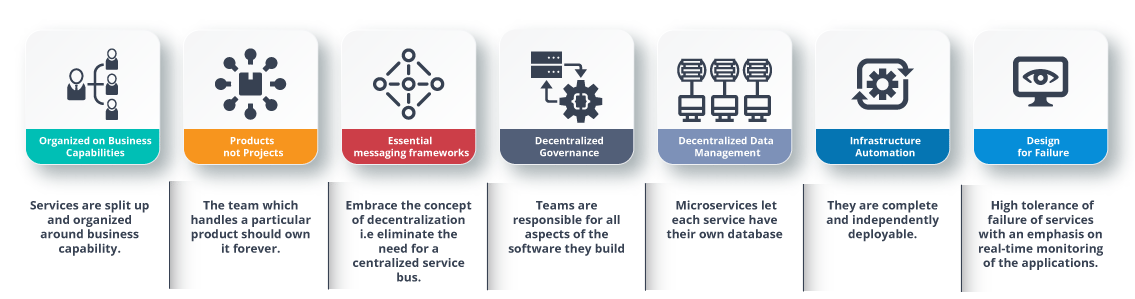
**Principles Used to Design Microservice Architecture**

The principles used to design Microservices are as follows:

* Independent & Autonomous Services
* Scalability
* Decentralization
* Resilient Services
* Real-Time Load Balancing
* Availability
* Continuous delivery through DevOps Integration
* Seamless API Integration and Continuous Monitoring
* Isolation from Failures
* Auto -Provisioning
* 
* **Monolithic Architecture** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
* A **Service-Oriented Architecture** is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
* **Microservice Architecture** is an architectural style that structures an application as a collection of small autonomous services, modelled around a business domain.

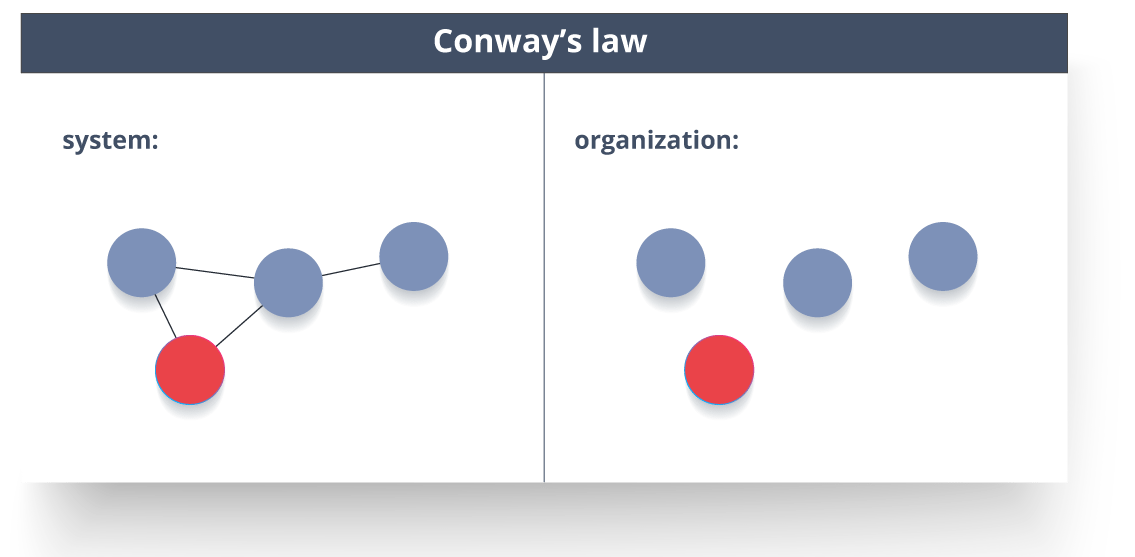
### ****What are the characteristics of Microservices?****

1. **High Cohesion** - Small and focussed on doing one thing well. Small does not mean less number of lines of code because few programming languages are more verbose than others, but it means the smallest functional area that a single microservices caters to.
2. **Loose Coupling** - Autonomous - the ability to deploy different services independently, and reliability, due to the ability for a service to run even if another service is down.
3. **Bounded Context** - A Microservice serves a bounded context in a domain. It communicates with the rest of the domain by using an interface for that Bounded context.
4. Organisation around business capabilities instead of around technology.
5. Continuous Delivery and Infrastructure automation.
6. Versioning for backward compatibility. Even multiple versions of same microservices can exist in a production environment.
7. **Fault Tolerance** - if one service fails, it will not affect the rest of the system. For example, if a microservices serving the comments and reviews for e-commerce fails, the rest of the website should run fine.
8. Decentralized data management with each service owning its database rather than a single shared database. Every microservice has the freedom to choose the right type of database appropriate for its business use-case (for example, RDBMS for Order Management, NoSql for catalogue management for an e-commerce website)
9. **Eventual Consistency** - event-driven asynchronous updates.
10. **Security** - Every microservice should have the capability to protect its own resources from unauthorized access. This is achieved using stateless security mechanisms like JSON Web Token (JWT pronounced as jot) with OAuth2.

* 

### ****What is Conway’s law?****

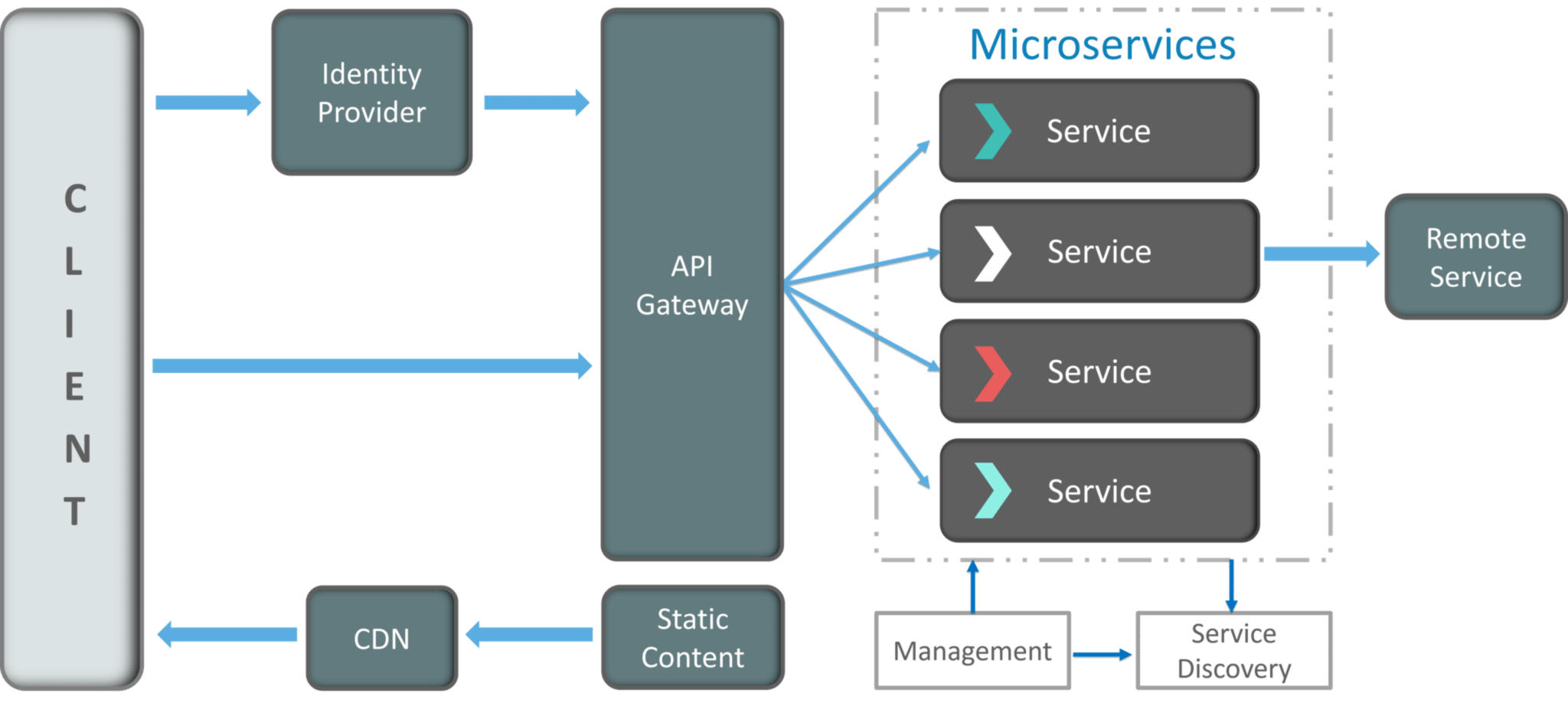
“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.” –***Mel Conway***



This law basically tries to convey the fact that, in order for a software module to function, the complete team should communicate well. Therefore the structure of a system reflects the social boundaries of the organization(s) that produced it.

#### **How does Microservice Architecture work?**

* **Clients** – Different users from various devices send requests.
* **Identity Providers** – Authenticates user or clients identities and issues security tokens.
* **API Gateway** – Handles client requests.
* **Static Content** – Houses all the content of the system.
* **Management** – Balances services on nodes and identifies failures.
* **Service Discovery** – A guide to find the route of communication between microservices.
* **Content Delivery Networks** – Distributed network of proxy servers and their data centers.
* **Remote Service** – Enables the remote access information that resides on a network of IT devices.



**Design Patterns of Microservices**

* Decomposition Design Pattern Microservices are developed with an idea on developers mind to create small services, with each having their own functionality. But, breaking an application into small autonomous units has to be done logically. So, to decompose a small or big application into small services, you can use the Decomposition patterns.

With the help of this pattern, either you can decompose an application based on business capability or on based on the sub-domains. For example, if you consider an e-commerce application, then you can have separate services for orders, payment, customers, products if you decompose by business capability.

But, in the same scenario, if you design the application by decomposing the sub-domains, then you can have services for each and every class. Here, in this example, if you consider the customer as a class, then this class will be used in customer management, customer support, etc. So, to decompose, you can use the Domain-Driven Design through which the whole domain model is broken down into sub-domains. Then, each of these sub-domains will have their own specific model and scope(bounded context). Now, when a developer designs microservices, he/she will design those services around the scope or bounded context.

Though these patterns may sound feasible to you, they are not feasible for big monolithic applications. This is because of the fact that identifying sub-domains and business capabilities is not an easy task for big applications. So, the only way to decompose big monolithic applications is by following the Vine Pattern or the Strangler Pattern.

* Strangler Pattern or Vine Pattern

The Strangler Pattern or the Vine Pattern is based on the analogy to a vine which basically strangles a tree that it is wrapped around. So, when this pattern is applied on the web applications, a call goes back and forth for each URI call and the services are broken down into different domains. These domains will be hosted as separate services.

According to the strangler pattern, two separate applications will live side by side in the same URI space, and one domain will be taken in to account at an instance of time. So, eventually, the new refactored application wraps around or strangles or replaces the original application until you can shut down the monolithic application

* Scalability
* Availability
* Resiliency
* Independent, autonomous
* Decentralized governance
* Failure isolation
* Auto-Provisioning
* 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. 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. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, 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.

**c. Strangler Pattern**

* Problem

So far, the design patterns we talked about were decomposing applications for greenfield, but 80% of the work we do is with brownfield 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. This creates two separate applications that live side by side in the same URI space. Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.

**2. Integration Patterns**

**a. API Gateway Pattern**

API Gateway is a special class of microservices that meets the need of a single client application (such as android app, web app, angular JS app, iPhone app, etc) and provide it with single entry point to the backend resources (microservices), providing cross-cutting concerns to them such as security, monitoring/metrics & resiliency.

Client Application can access tens or hundreds of microservices concurrently with each request, aggregating the response and transforming them to meet the client application’s needs. Api Gateway can use a client-side load balancer library (Ribbon) to distribute load across instances based on round-robin fashion. It can also do protocol translation i.e. HTTP to AMQP if necessary. It can handle security for protected resources as well.

Features of API Gateway

* Spring Cloud DiscoveryClient integration
* Request Rate Limiting (available in Spring Boot 2.x)
* Path Rewriting
* Hystrix Circuit Breaker integration for resiliency
* Problem

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

How to call multiple microservices abstracting producer information.

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.

Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?

How to handle different type of Protocols some of which might not be supported by producer microservice.

API Gateway

Client

Load Balancer

Service A

Service B

* Solution

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

An API Gateway is the single point of entry for any microservice call.

It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.

It can fan out a request to multiple services and aggregate the results to send back to the consumer.

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.

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.

It can also offload the authentication/authorization responsibility of the microservice.

**b. Aggregator Pattern**

* Problem

We have talked about resolving the aggregating data problem in the API Gateway Pattern. However, we will talk about it here holistically. 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.

Service B

Aggregator

Service A

* 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.

**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 greenfield and to be developed with DDD. 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 for brownfield 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.

**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**

* 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:

Choreography — 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.

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

**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 behaviour through logs for a particular request?

* Solution

We need a centralized logging service that aggregates logs from each service instance. Users can search and analyse 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

Pull — the metrics services pulls metrics from the service e.g. Prometheus

**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 Slueth, 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.

**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.

**b. Service Discovery Pattern**

* Problem

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

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.

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?

Down Service

**Remote Service**

**Load Balancer**

**Client**

**Service A**

**Service A**

**Service A**

Circuit Breaker

* 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

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the 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.

**Aggregator Microservice Design Pattern**

The first, and probably the most common, is the aggregator microservice design pattern.

In its simplest form, Aggregator would be a simple web page that invokes multiple services to achieve the functionality required by the application. Since each service (Service A, Service B, and Service C) is exposed using a lightweight REST mechanism, the web page can retrieve the data and process/display it accordingly. If some sort of processing is required, say applying business logic to the data received from individual services, then you may likely have a CDI bean that would transform the data so that it can be displayed by the web page.

Another option for Aggregator is where no display is required, and instead it is just a higher level composite microservice which can be consumed by other services. In this case, the aggregator would just collect the data from each of the individual microservice, apply business logic to it, and further publish it as a REST endpoint. This can then be consumed by other services that need it.

This design pattern follows the DRY principle. If there are multiple services that need to access Service A, B, and C, then its recommended to abstract that logic into a composite microservice and aggregate that logic into one service. An advantage of abstracting at this level is that the individual services, i.e. Service A, B, and C, and can evolve independently and the business need is still provided by the composite microservice.

Note that each individual microservice has its own (optional) caching and database. If Aggregator is a composite microservice, then it may have its own caching and database layer as well.

Aggregator can scale independently on X-axis and Z-axis as well. So if its a web page then you can spin up additional web servers, or if its a composite microservice using Java EE, then you can spin up additional WildFly instances to meet the growing needs.

**Proxy Microservice Design Pattern**

Proxy microservice design pattern is a variation of Aggregator. In this case, no aggregation needs to happen on the client but a different microservice may be invoked based upon the business need.

Just like Aggregator, Proxy can scale independently on X-axis and Z-axis as well. You may like to do this where each individual service need not be exposed to the consumer and should instead go through an interface.

The proxy may be a dumb proxy in which case it just delegates the request to one of the services. Alternatively, it may be a smart proxy where some data transformation is applied before the response is served to the client. A good example of this would be where the presentation layer to different devices can be encapsulated in the smart proxy.

**Chained Microservice Design Pattern**

Chained microservice design pattern produce a single consolidated response to the request. In this case, the request from the client is received by Service A, which is then communicating with Service B, which in turn may be communicating with Service C. All the services are likely using a synchronous HTTP request/response messaging.

The key part to remember is that the client is blocked until the complete chain of request/response, i.e. Service <-> Service B and Service B <-> Service C, is completed. The request from Service B to Service C may look completely different as the request from Service A to Service B. Similarly, response from Service B to Service A may look completely different from Service C to Service B. And that’s the whole point anyway where different services are adding their business value.

Another important aspect to understand here is to not make the chain too long. This is important because the synchronous nature of the chain will appear like a long wait at the client side, especially if its a web page that is waiting for the response to be shown. There are workarounds to this blocking request/response and are discussed in a subsequent design pattern.

A chain with a single microservice is called singleton chain. This may allow the chain to be expanded at a later point.

**Branch Microservice Design Pattern**

Branch microservice design pattern extends Aggregator design pattern and allows simultaneous response processing from two, likely mutually exclusive, chains of microservices. This pattern can also be used to call different chains, or a single chain, based upon the business needs.

Service A, either a web page or a composite microservice, can invoke two different chains concurrently in which case this will resemble the Aggregator design pattern. Alternatively, Service A can invoke only one chain based upon the request received from the client.

This may be configured using routing of JAX-RS or Camel endpoints, and would need to be dynamically configurable.

**Shared Data Microservice Design Pattern**

One of the design principles of microservice is autonomy. That means the service is full-stack and has control of all the components – UI, middleware, persistence, transaction. This allows the service to be polyglot, and use the right tool for the right job. For example, if a NoSQL data store can be used if that is more appropriate instead of jamming that data in a SQL database.

However a typical problem, especially when refactoring from an existing monolithic application, is database normalization such that each microservice has the right amount of data – nothing less and nothing more. Even if only a SQL database is used in the monolithic application, denormalizing the database would lead to duplication of data, and possibly inconsistency. In a transition phase, some applications may benefit from a shared data microservice design pattern.

In this design pattern, some microservices, likely in a chain, may share caching and database stores. This would only make sense if there is a strong coupling between the two services. Some might consider this an anti-pattern but business needs might require in some cases to follow this. This would certainly be an anti-pattern for greenfield applications that are design based upon microservices.

This could also be seen as a transition phase until the microservices are transitioned to be fully autonomous.

**Asynchronous Messaging Microservice Design Pattern**

While REST design pattern is quite prevalent, and well understood, but it has the limitation of being synchronous, and thus blocking. Asynchrony can be achieved but that is done in an application specific way. Some microservice architectures may elect to use message queues instead of REST request/response because of that.

In this design pattern, Service A may call Service C synchronously which is then communicating with Service B and D asynchronously using a shared message queue. Service A -> Service C communication may be asynchronous, possibly using WebSockets, to achieve the desired scalability.

A combination of REST request/response and pub/sub messaging may be used to accomplish the business need.

**List down the advantages of Microservices Architecture.**

|  |  |
| --- | --- |
| Advantages of Microservices Architecture | |
| **Advantage** | **Description** |
| *Independent Development* | All microservices can be easily developed based on their individual functionality |
| *Independent Deployment* | Based on their services, they can be individually deployed in any application |
| *Fault Isolation* | Even if one service of the application does not work, the system still continues to function |
| *Mixed Technology Stack* | Different languages and technologies can be used to build different services of the same application |
| *Granular Scaling* | Individual components can scale as per need, there is no need to scale all components together |

**[What are the challenges in microservices?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3587)**

1. DevOps is a must, because of the explosion of a number of processes in a production system. How to start and stop the fleet of services?
2. The complexity of distributed computing such as “network latency, fault tolerance, message serialization, unreliable networks, handling asynchronous o/p, varying loads within our application tiers, distributed transactions, etc.”
3. How to make configuration changes across the large fleet of services with minimal effort?
4. How to deploy multiple versions of single microservice and route calls appropriately?
5. How to disconnect a microservice from ecosystem when it starts to crash unexpectedly?
6. How to isolate a failed microservice and avoid cascading failures in the entire ecosystem?
7. How to discover services in an elastic manner considering that services may be going UP or DOWN at any point in time?
8. How to aggregate logs/metrics across the services? How to identify different steps of a single client request spread across a span of microservices?
9. 