Software Architectural Styles vs. Architectural Patterns vs. Design Patterns

An **architectural style** is a conceptual way of how the system will be created / will work.

Software Architectural Styles

Architectural Style	Definition
Client - Server	It is the two-tier architecture where the client make the service request to the server
Component-Based	Decomposes the application into reusable functional or logical components that are Location transparent and expose well defined communication interfaces.
Layered	Layering the app into groups such as presentation layer, business layer, data layer, services layer.
hexagonal	In this architecture, the application in the inside communications over some number of ports with systems on the outside.
Message-Bus	A software system where capable of Establishing the communication between systems based on rules. The systems do not have to worry about the receiving end.
N-tier/3-tier	Each segment is located physically in separate computer may be three or more tiers like the layered system.

Object-Oriented	Based on division of tasks for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Separated Presentation	Logic Separation between the user interface and the user data.
Service-Oriented Architecture	Collection of applications that expose and consume functionalities as services using contracts such as APIs and messages.
Micro services/ Containerization/ Multitenancy	"Microservices architectural style is an approach to developing *a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms*, often an HTTP resource API. These services are built around business capabilities and *independently deployable* by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." (James Lewis, 2014)
Cell Architecture	Emerging Architectural Style with architectural patterns with in the Enterprise.

An ***Architectural Pattern*** is a recurring solution to a recurring problem. In the case of Architectural Patterns, they solve the problems related to the Architectural Style.

For example, "*what classes will we have and how will they interact, in order to implement a system with a specific set of layers* ", or "*what high-level modules will have in our Service-Oriented Architecture and how will they communicate* ", or "*how many tiers will our Client-server Architecture have* ".

Architectural Patterns have an extensive impact on the code base, most often impacting the whole application either horizontally (ie. how to structure the code inside a layer) or vertically (ie. how a request is processed from the outer layers into the inner layers and back). Examples of Architectural Patterns:

- Three-tier
- Microkernel
- Event Driven
- Microservices

Design Patterns differ from Architectural Patterns in their scope, they are more localized, they have less impact on the code base, they impact a specific section of the code base, for example:

- How to instantiate an object when we only know what type needs to be instantiated at run time (maybe a Factory Class? – A design pattern in Java);
- How to make an object behave differently according to its state (maybe a state machine, or a Strategy Pattern?).

Focusing the Architecture Style / Architecture patterns / Design patterns with microservices

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Analogy:

Think of a House

1. Helicopter view of a house – or looking at the roof of the house from a top of a tree. This is something similar to architecture style
2. Just enter to a house from main door and take look at from outside. This is like Architectural pattern
3. Enter one of the specific rooms. example study room. and see the way u are studying ... like which subject you are going to read today is like design pattern.

Principles Used to Design Microservice Architecture – Architecture Patterns

1. **High cohesion along with loose coupling.**

   High Cohesion, Loose Coupling is a principle that is often mentioned in the context of microservices. Everything that logically belongs together should be in 1 microservice, and the different microservices should be loosely coupled. Loose coupling is harder to achieve than you might think, because there are many kinds of coupling:

2. **Seamless API Integration.**
3. **A unique source of Identification for every service.**
4. **Real-Time Traffic Management**
5. **Minimizing data tables to optimize load.**
6. **Performing constant monitoring over external and internal APIs.**
7. **Isolated data storage for each microservice** – This is very important to maintain limited access to data and avoid 'service coupling'. Classification of data based on the users is important and can be achieved through the Command and Query Responsibility Segregation (CQRS).

8. **Decentralization –** The first and foremost principle to design microservice architecture is the ability to break down the monolithic architecture into separate individual entities. These entities are known as microservices. These microservices work independent of the other system functions and all users to edit, delete or employ any functionality without affecting the system performance.

9. **Scalability –** Microservices are built with an aim in mind: Performance and efficiency. In real-world problem solving, expansion and large-scale systems are crucial to the performance of any microservice ecosystem. Scalability is crucial to design microservice architecture. With the possibility of multiple fragments functioning on multiple technologies, working with larger amounts of data can be a challenge. But, proper implementation and use of Application Controllers can make scalability with microservice architecture possible.

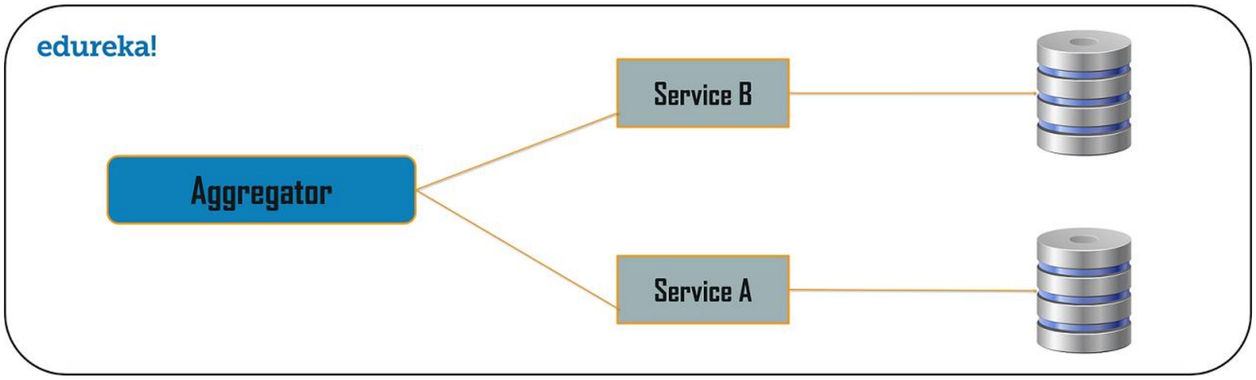
10. **Continuous delivery through DevOps Integration –** Those working in DevOps often receive microservice architecture well because of the ease of accessibility and integration of multiple technologies.

Microservice Design Patterns for Effective Collaboration

With so many microservices running simultaneously, collaboration comes as a necessity for running an efficient microservice architecture. Today we will look at the collaboration patterns for designing a microservice.

1. Aggregator Microservice Design Pattern

Aggregator in the computing world refers to a website or program that collects related items of data and displays them. So, even in Microservices patterns, Aggregator is a basic web page which invokes various services to get the required information or achieve the required functionality.



Let’s imagine that we’ve been tasked with developing an internal API for our organization - a general practice clinic. The API needs to consume data from three existing microservices, each of which is used by other services within the practice’s architecture - some services call these individually, some call all three.

The requirements for the API are that it should return simple details for a patient, a list of their allergies and a list of medication that they are currently taking.

Our three existing microservices are as follows:

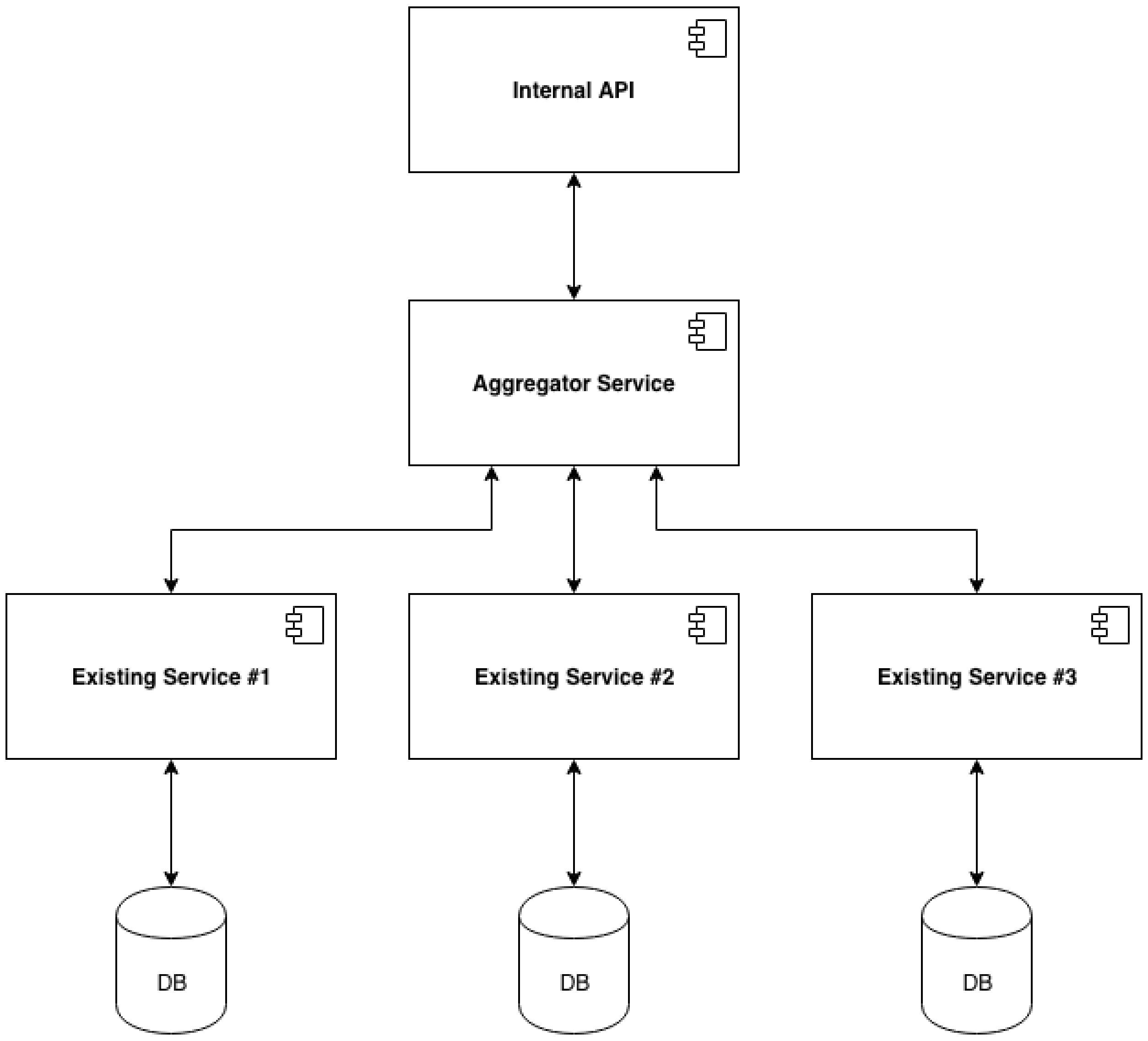
Existing Service #1 returns details about a patient - their name, age, etc.

Existing Service #2 returns a list of allergies that the patient has.

Existing Service #3 returns a list of medication that the patient is currently taking.

Usually, we’d expect an aggregator to make synchronous calls to relevant microservices, performing any necessary business logic on each result as it receives it and then packaging this up as an API endpoint for a consumer to use. This meets our requirements whilst opening up potential for re-use and decoupling.

Rather than increasing the number of services which call these microservices directly, we can make use of the aggregator pattern here.



Our new internal API will call our new aggregator microservice, which will call the three existing microservices before then pushing the necessary results back up to the internal API.

We can re-use our aggregator within other services which call all three existing services, decoupling these from direct interaction with the microservices, which will make it easier to replace one later down the line; if we want to suddenly commission a new allergies microservice, we only have to update the aggregator (and those services which don’t call *all three* existing services).

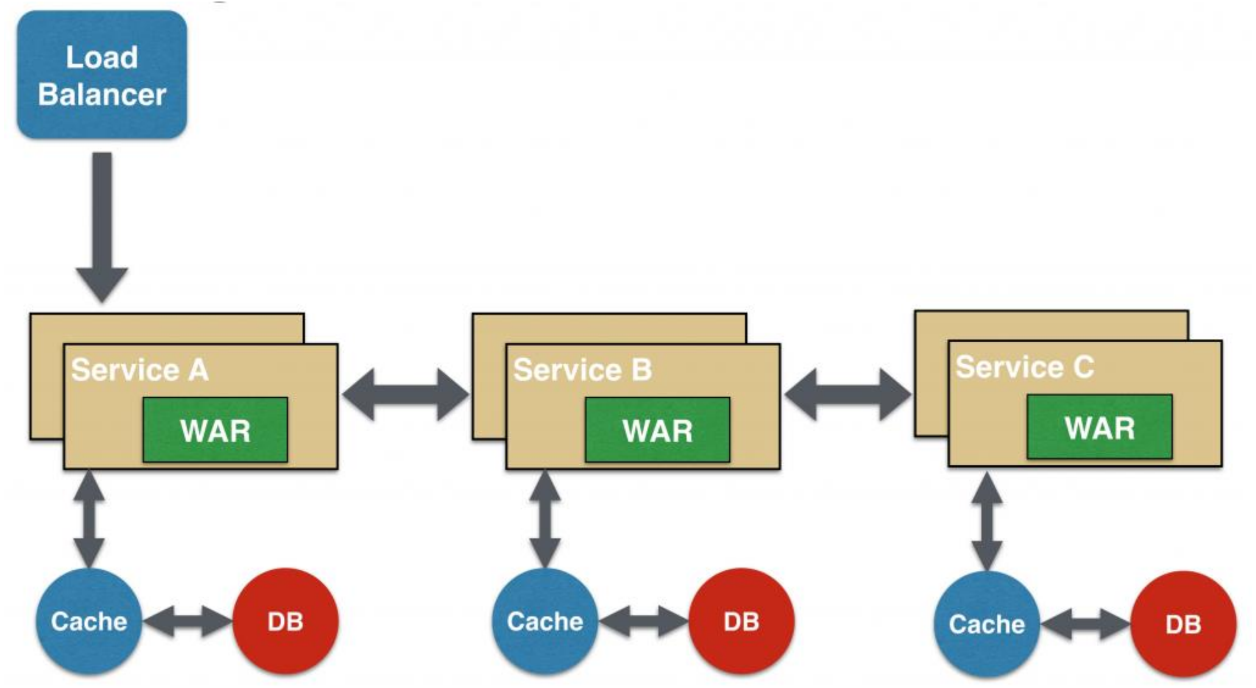
The Aggregate Design Pattern is based on the DRY principle. Based on this principle, you can abstract the logic into a composite microservices and aggregate that particular business logic into one service.

The premise of **microservices** is based on autonomous and fine-grained units of code that do one thing and one thing only. This is closely aligned with the **principle** of “don't repeat yourself” (**DRY**), which states that every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

https://github.com/iluwatar/java-design-patterns/tree/master/aggregator-microservices

2. 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.

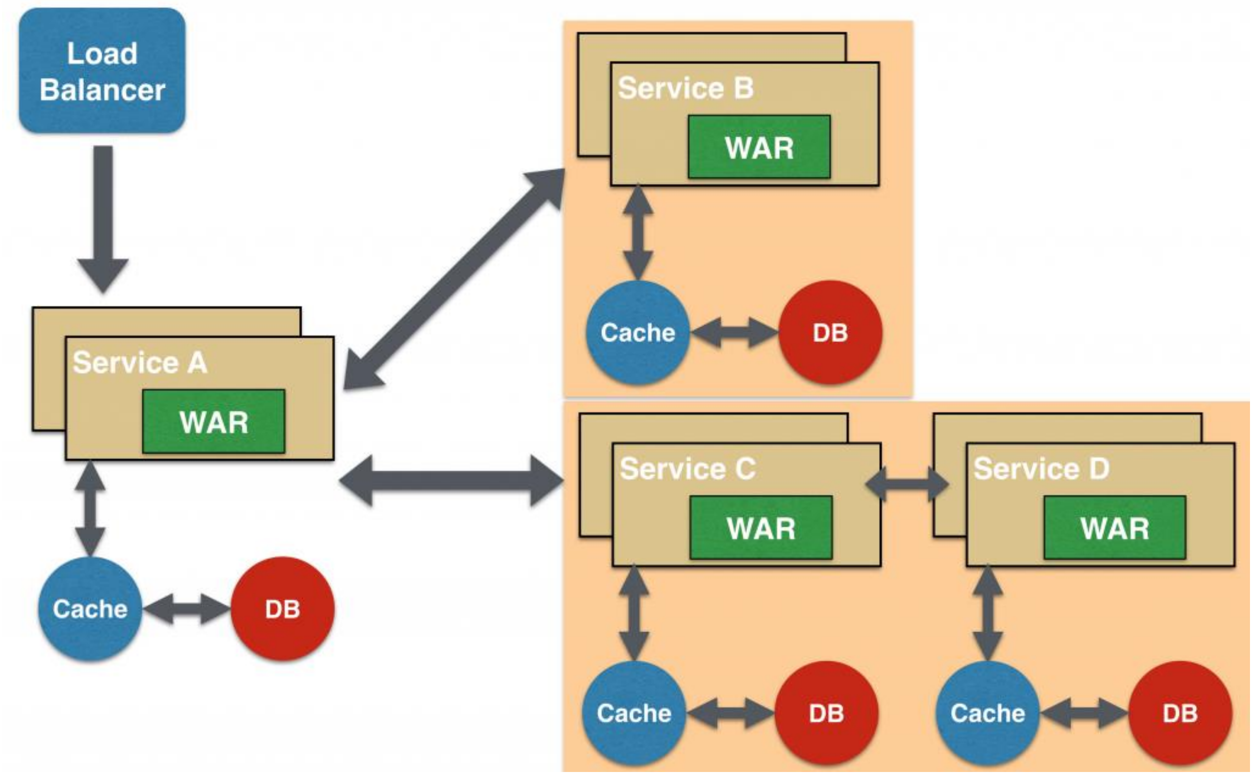


The key part to remember is that the client is blocked until the complete chain of request/response.

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

3 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.

.

4. Backend for Front-End/ API Gateway

API Gateway Design Pattern

Microservices are built in such a way that each service has its own functionality. But, when an application is broken down into small autonomous services, then there could be few problems that a developer might face. The problems could be as follows:

1. How can I request information from multiple microservices?
2. Different UI require different data to respond to the same backend database service
3. How to transform data according to the consumer requirement from reusable Microservices
4. How to handle multiple protocol requests?

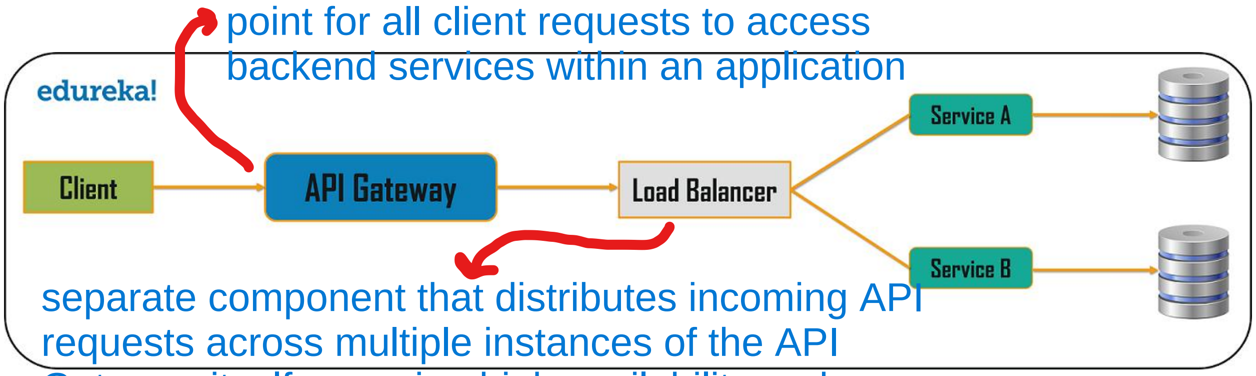
Well, the solution to these kinds of problems could be the API Gateway Design Pattern. The API Gateway Design Pattern address not only the concerns mentioned above but it solves many other problems. This microservice design pattern can also be considered as the proxy service to route a request to the concerned microservice. Being a variation of the Aggregator service, it can send the request to multiple services and similarly aggregate the results back to the composite or the consumer service. API

Gateway also acts as the entry point for all the microservices and creates fine-grained APIs' for different types of clients.

With the help of the API Gateway design pattern, the API gateways can convert the protocol request from one type to other. Similarly, it can also offload the authentication/authorization responsibility of the microservice.

So, once the client sends a request, these requests are passed to the API Gateway which acts as an entry point to forward the clients' requests to the appropriate microservices. Then, with the help of the load balancer, the load of the request is handled and the request is sent to the respective services. Microservices use Service Discovery which acts as a guide to find the route of communication between each of them. Microservices then communicate with each other via a stateless server i.e. either by HTTP Request/Message Bus.

a server that acts as a single entry point for all client requests to access backend services within an application



separate component that distributes incoming API requests across multiple instances of the API Gateway itself, ensuring high availability and scalability

https://github.com/iluwatar/java-design-patterns/tree/master/api-gateway

Microservice Design Patterns for Performance Monitoring

Monitoring the performance is an important aspect for a successful microservice architecture. It helps calculate the efficiency and understand any drawbacks which might be slowing the system down. Remember the following patterns related to observability for ensuring a robust microservice architecture design.

1. Log Aggregation

When we refer to a microservice architecture we are referring to a refined yet granular architecture where an application is consisting a number of microservices. These microservices run independently and simultaneously as supporting multiple services as well as their instances across various machines. Every service generates an entry in the logs regarding its execution.

How can you keep a track for numerous service-related logs? This is where log aggregation steps in.

As a best practice to prevent from chaos, you should be having a master logging service. This master logging service should be responsible for aggregating the logs from all the microservice instances. This centralized log should be searchable, making it easier to monitor.

2. Synthetic Monitoring / Semantic Monitoring

As explained previously, monitoring is a painful but indispensable task for a successful microservice architecture. With simultaneous execution of hundreds of services, it becomes troublesome to pinpoint the root area responsible for the failure in log registry. Synthetic monitoring gives a helping hand. When you perform automated test then synthetic monitoring helps to regularly map the results in comparison to the production environment. User gets alerted if a failure is generated. Using Semantic Monitoring you can aim for 2 things using a single arrow

- Monitoring automated test cases.
- Detecting Production failures in terms of business requirements.

3. API Health Check

What is an API

And Rest API

Creating a simple API is spring boot

Rest API

Microservice architecture design promotes services which are independent of each other to avoid any delay in the system. APIs as we know serve as the building blocks of an online connectivity. It is imperative to keep a health check on your APIs on regular basis to realize any roadblock. It is often observed that a microservice is up and running yet incapacitated for handling requests. This can be due to many factors:

- Server Loads
- User Adoption
- Latency
- Error Logging
- Market Share
- Downloads

In order to overcome this scenario, we should ensure that every service running must have a specific health check API endpoint. For example: HTTP/health when appended at the end of every service will return the health status for respective service instance. A service registry periodically appeals to the health check API endpoint to perform a health scan. The health check would provide you with the information on the below-mentioned:

1. A logic that is specific to your application.
2. Status of the host.
3. Status of the connections to other infrastructure or connection to any service instance.

Breaking it all down to Business Capability

The process of ‘decomposing’ a monolithic architecture into a microservice needs to follow certain parameters. These parameters have a different basis. Today we will look at the decomposition of the microservice design patterns which leave a lasting impact.

1. Unique Microservice for each Business Capability

A microservice is as successful as its combination of high cohesion and loose coupling. Services need to be loosely coupled while keeping the function of similar interests together. But how do we do it? How do we decompose a software system into smaller independent logical units?
We do so by defining the scope of a microservice to support a specific business capability.

Loose coupling implies that services are independent so that changes in one service will not affect any other. The more dependencies you have between services, the more likely it is that changes will have wider, unpredictable consequences.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

- Coupling: the degree to which components have knowledge of other components. Think well-defined interfaces, inversion of control etc

- Cohesion: the degree to which the elements within a component belong together. Think single responsibility principle and single reason to change

High Cohesion, Loose Coupling is a principle that is often mentioned in the context of microservices. Everything that logically belongs together should be in 1 microservice, and the different microservices should be loosely coupled. Loose coupling is harder to achieve than you might think, because there are many kinds of coupling:

For Example –
In every organization, there are different departments that come together as one. These include technical, marketing, PR, sales, service, and maintenance. To picture a microservice structure these different domains would each be the microservices and the organization will be the system.
So an Inventory management is responsible for all the inventories. Similarly, Shipping management will handle all the shipments and so on.

To maintain efficiency and foresee growth, the best solution is to decompose the systems using business capability. This includes classification into various business domains which are responsible to generate value in their own capabilities.

2. Microservices around similar Business Capability

Despite segregating on the basis of business capabilities, microservices often come up with a greater challenge. What about the common classes among the services? Well, decomposing these classes known as 'God Classes' needs intervention. For example, in case of an e-commerce system, the order will be common to several services such as order number, order management, order return, order delivery etc. To solve this issue, we turn to a common microservice design principle known as Domain-Driven Design (DDD).

In Domain-Driven Design, we use subdomains. These subdomain models have defined scope of functionality which is known as bounded context. This bounded context is the parameter used to create each microservice thus overcoming the issues of common classes.

3. Strangler Vine Pattern

While we discuss decomposition of a monolithic architecture, we often miss out the struggle of converting a monolithic system to design microservice architecture. Without hampering the working, converting can be extremely tough. And to solve this problem we have the strangler pattern, based on the vine analogy. Here is what the Strangler patterns mean in Martin Fowler's words:

"One of the natural wonders of this area [Australia] is the huge strangler vines. They seed in the upper branches of a fig tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host."
Strangler pattern is extremely helpful in case of a web application where breaking down a service into different domains is possible. Since the calls go back and forth, different services live on different domains. So, these two domains exist on the same URI. Once the service has been reformed, it **'strangles'** the existing version of the application. This process is followed until the monolith doesn't exist.

https://github.com/iluwatar/java-design-patterns/tree/master/strangler

Microservice Design Patterns for Optimizing Database Storage

For a microservice architecture, loose coupling is a basic principle. This enables deployment and scalability of independent services. Multiple services might need to access data not stored in their unit. But due to loose coupling, accessing this data can be a challenge. Mainly because different services have different storage requirements and access to data is limited in microservice design. So, we look at some major database design patterns as per different requirements.

1. Individual Database per Service

Usually applied in Domain Driven Designs, one database per service articulates the entire database to a specific microservice. Due to the challenges and lack of accessibility, a single database per service needs to be designed. This data is accessible only by the microservice. This database has limited access for any outside microservices. The only way for others to access this data is through microservice API gateways.

https://microservices.io/patterns/data/database-per-service.html https://github.com/iluwatar/java-design-patterns/tree/master/dao

https://github.com/eventuate-tram/eventuate-tram-sagas-examples-customers-and-orders

2. Shared Database per Service

In Domain Driven Design, a separate database per service is feasible, but in an approach where you decompose a monolithic architecture to microservice, using a single database can be tough. So while the process of decomposition goes on, implementing a shared database for a limited number of service is advisable. This number should be limited to 2 or 3 services. This number should stay low to allow deployment, autonomy, and scalability.

3. Event Sourcing Design Pattern

According to Martin Fowler

"Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.
The problem here lies with reliability. How can you rely on the architecture to make a change or publish a real-time event with respect to the changes in state of the application?
Event sourcing helps to come up from this situation by appending a new event to the list of the events every time a business entity changes its state. Entities like Customer may consist of numerous events. It is thus advised that an application saves a screenshot of the current state of an entity in order to optimize the load.

https://github.com/iluwatar/java-design-patterns/tree/master/event-sourcing

3. Command Query Responsibility Segregation (CQRS)

In a database-per-service model, the query cannot be implemented because of the limited access to only one database. For a query, the requirements are based on joint database systems. But how do we query then?

Based on the CQRS, to query single databases per service model, the application should be divided into two parts: Command and Query. In this model, command handles all requests related to create, update and delete while queries are taken care of through a materialized view. These views are updated through a stream of events.

These events, in turn, are created using an event sourcing pattern which marks any changes in the data. These changes eventually become events.

https://github.com/iluwatar/java-design-patterns/tree/master/cqrs

Microservice Design Patterns for Seamless Deployment

When we implement microservices, there are certain issues which come up during the call of these services. When you design microservice architecture, certain cross-cutting patterns can simplify the working.

1. Service Discovery

The use of containers leads to dynamic allocation of the IP address. This means the address can change at any moment. This causes a service break. In addition to this, the users have to bear the load of remembering every URL for the services, which become tightly coupled.

To solve this problem and give users the location of the request, a registry needs to be used. While initiation, a service instance can register in the registry and de-register while closing. This enables the user to find out the exact location which can be queried. In addition, a health check by the registry will ensure the availability of only working instances. This also improves the system performance.

2. Blue-Green Deployment

In a microservice design pattern, there are multiple microservices. Whenever updates are to be implemented or newer versions deployed, one has to shut down all the services. This leads to a huge downtime thus affecting productivity. To avoid this issue, when you design microservice architecture, you should use the blue-green deployment pattern.

In this pattern, two identical environments run parallelly, known as blue and green. At a time only one of them is live and processing all the production traffic. For example, blue is live and addressing all the traffic. In case of new deployment, one uploads the latest version onto the green environment, switches the router to the same and thus implement the update.

Microservice Design Patterns for Performance Monitoring

Monitoring the performance is an important aspect for a successful microservice architecture. It helps calculate the efficiency and understand any drawbacks which might be slowing the system down. Remember the following patterns related to performance monitoring for ensuring a robust microservice architecture design.

1. Log Aggregation

When we refer to a microservice architecture we are referring to a refined yet granular architecture where an application is consisting a number of microservices. These microservices run independently and simultaneously as supporting multiple services as well as their instances across various machines. Every service generates an entry in the logs regarding its execution. How can you keep a track for numerous service related logs? This is where log aggregation steps in. As a best practice to prevent from chaos, you should be having a master logging service. This master logging service should be responsible for aggregating the logs from all the microservice instances. This centralized log should be searchable, making it easier to monitor.

2. Synthetic Monitoring a.k.a Semantic Monitoring

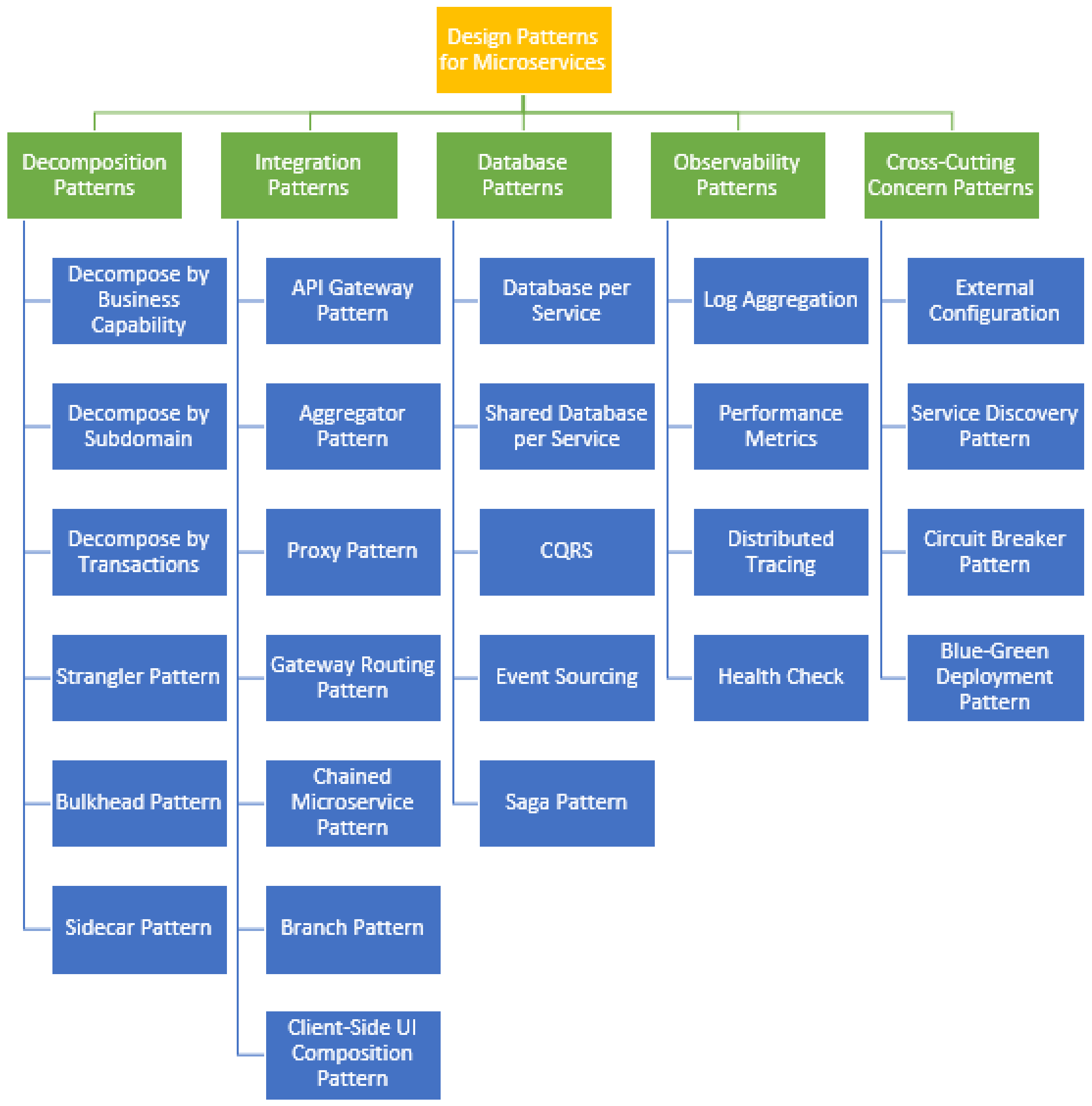
With the increase in load and microservices, it becomes important to keep a constant check on system performance. This includes any patterns which might be formed or addressing issues that come across. But more importantly, how is the data collected?

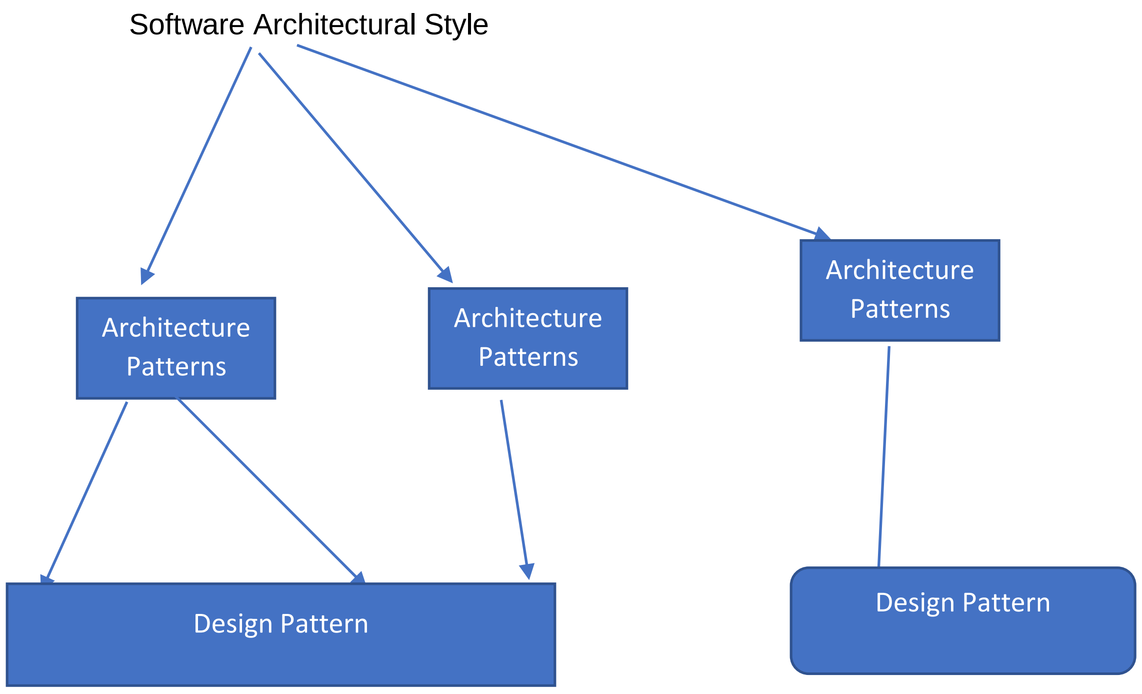
The answer lies with the use of a metric service. This metrics service is either in the Push form or the Pull form. As the name suggests, a Push service such as AppDynamics pushes the metrics to the service while a Pull service such as Prometheus pulls the data from the service.

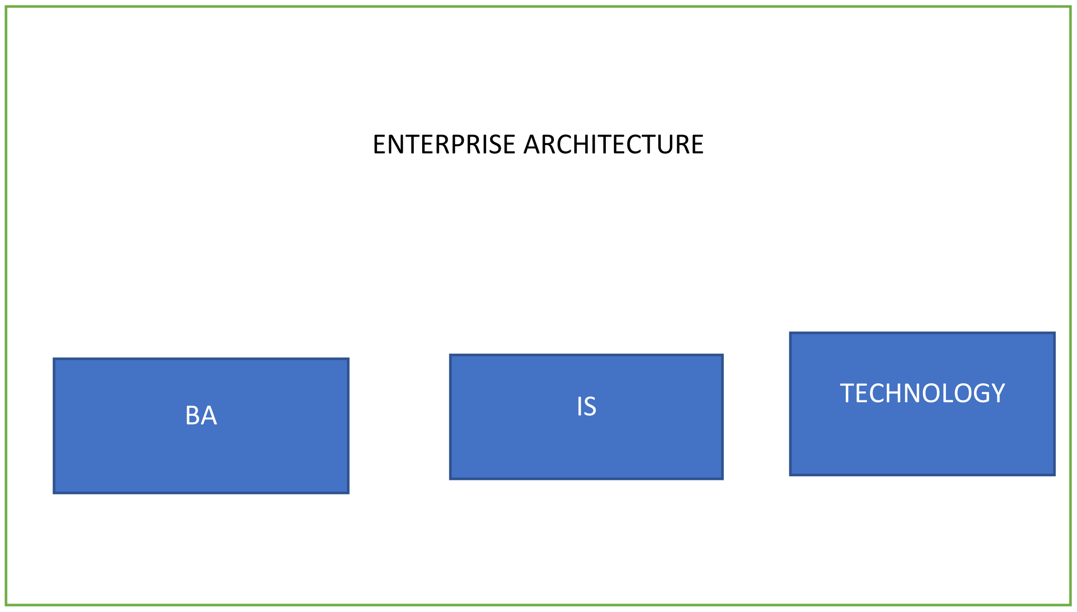
3. Running a Health Check

Microservice architecture design promotes services which are independent of each other to avoid any delay in the system. But, there are times when the system is up and running but it fails to handle transactions due to faulty services. To avoid requests to these faulty services, a load balancing pattern has to be implemented.

To achieve this, we use '/health' at the end of every service. This check is used to find out the health of the service. It includes the status of the host, its connection and the algorithmic logic.


Design Patterns for Microservices
Decomposition Patterns
Integration Patterns
Database Patterns
Observability Patterns
Cross-Cutting Concern Patterns
Decompose by Business Capability
Decompose by Subdomain
Decompose by Transactions
Strangler Pattern
Bulkhead Pattern
Sidecar Pattern
API Gateway Pattern
Aggregator Pattern
Proxy Pattern
Gateway Routing Pattern
Chained Microservice Pattern
Branch Pattern
Client-Side UI Composition Pattern
Database per Service
Shared Database per Service
CQRS
Event Sourcing
Saga Pattern
Log Aggregation
Performance Metrics
Distributed Tracing
Health Check
External Configuration
Service Discovery Pattern
Circuit Breaker Pattern
Blue-Green Deployment Pattern


Software Architectural Style
Architecture Patterns
Architecture Patterns
Architecture Patterns
Design Pattern
Design Pattern


ENTERPRISE ARCHITECTURE
BA
IS
TECHNOLOGY