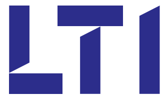


Let's Solve

A Larsen & Toubro Group Company

POV

Build Enterprise-Grade Microservices using Design Patterns

Author Rahul Kulkarni

Contents

Introduction To Microservices Patterns 3

How To Build Enterprise Grade Microservices Using Design Patterns 4

Microservice Architecture Aspects 5

Design Patterns 6

- Aggregator Pattern 6
- Api Gateway Design Pattern 7
- Chained Or Chain Of Responsibility Pattern 8
- Database Or Shared Data Pattern 9
- Command Query Responsibility Segregator Design Pattern 10
- Circuit Breaker Pattern 11
- Decomposition Pattern 12
- Strangler Pattern Or Vine Pattern 13

References 14

Summary 15

Conclusion 16

Introduction to Microservices Patterns

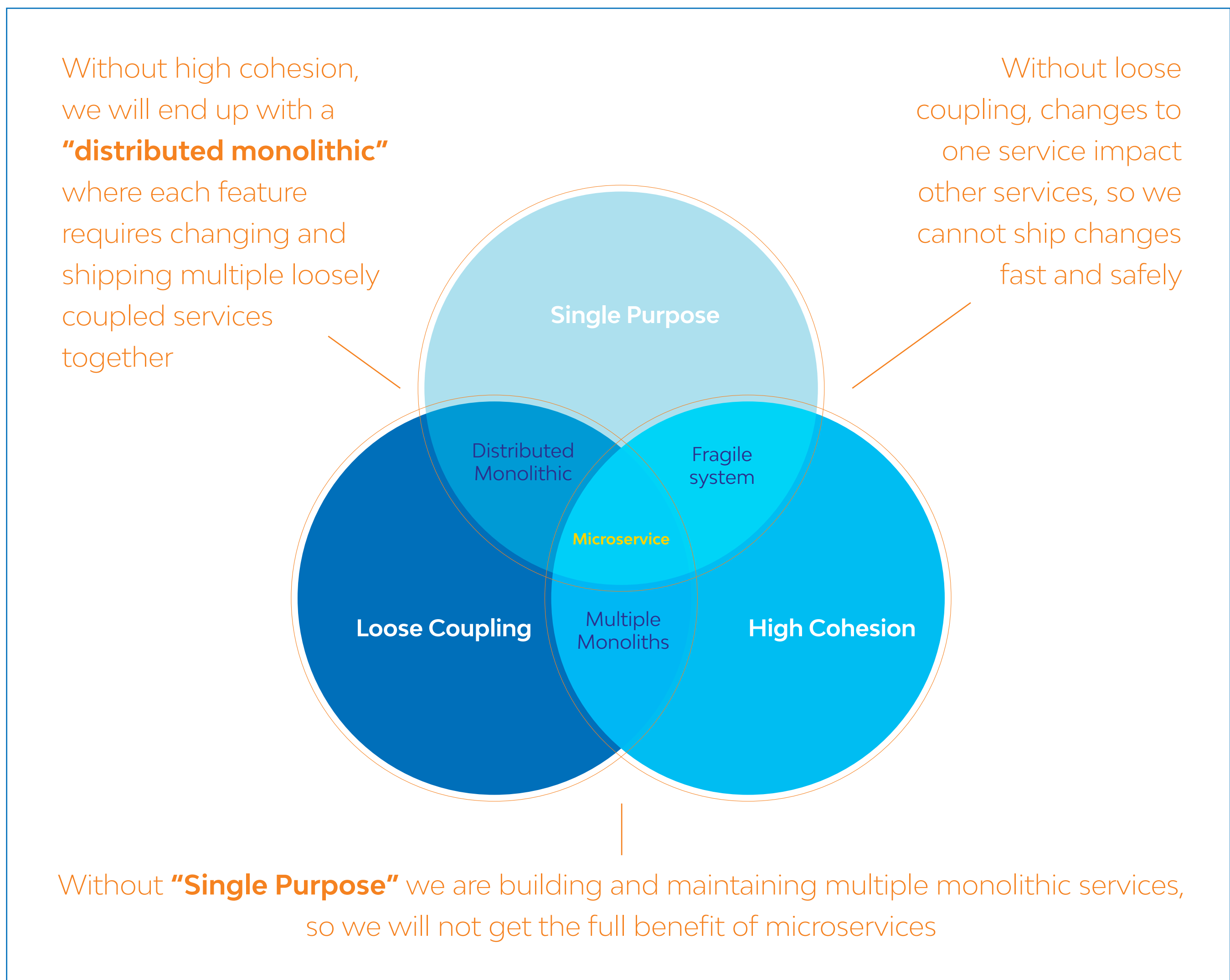
In today's world, Microservices have become a widely adopted pattern for building enterprise-grade applications. Various non-functional challenges can be addressed using the microservice- based architecture and developers can explore common patterns in the designs and create reusable solutions to improve the performance of the application. This whitepaper discusses specific design patterns that can be implemented for business requirements to help provide best configuration and performance outcomes, resulting in successful microservice implementation. The design patterns discussed here will help technical architects understand various designs used for Microservice implementation.

The current whitepaper contains how we address the microservices design using design patterns to address some common scenarios we encounter in the microservice implementation.

How to build enterprise-grade microservices using design patterns

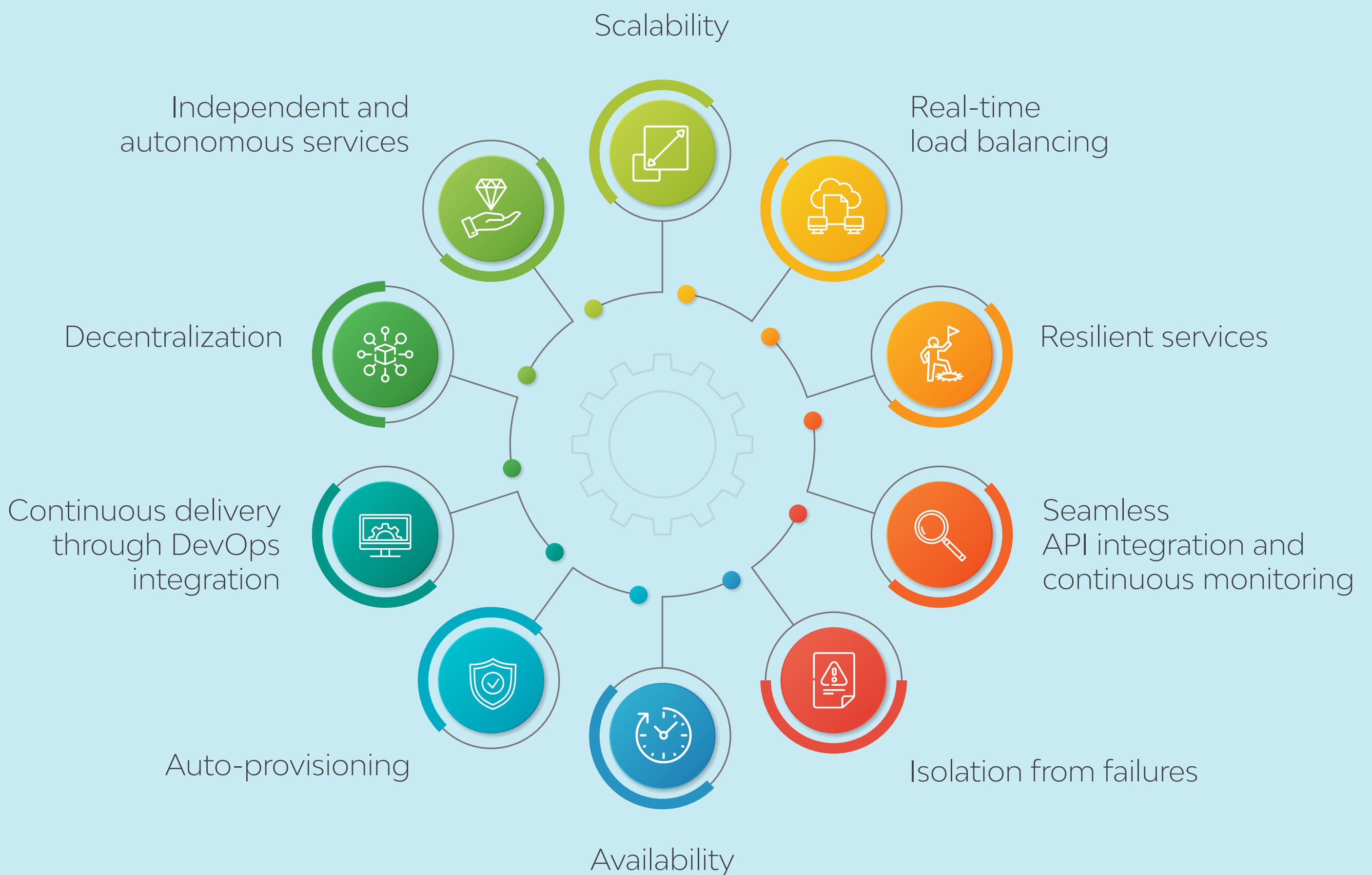
As you see in this diagram, Microservice is built on these three principles – Loose coupling, high cohesion and single purpose.

All the design patterns for microservice architecture are built on these three principles.



Microservice architecture aspects

Some of the important aspects apart from the three principles used in Microservice design are:



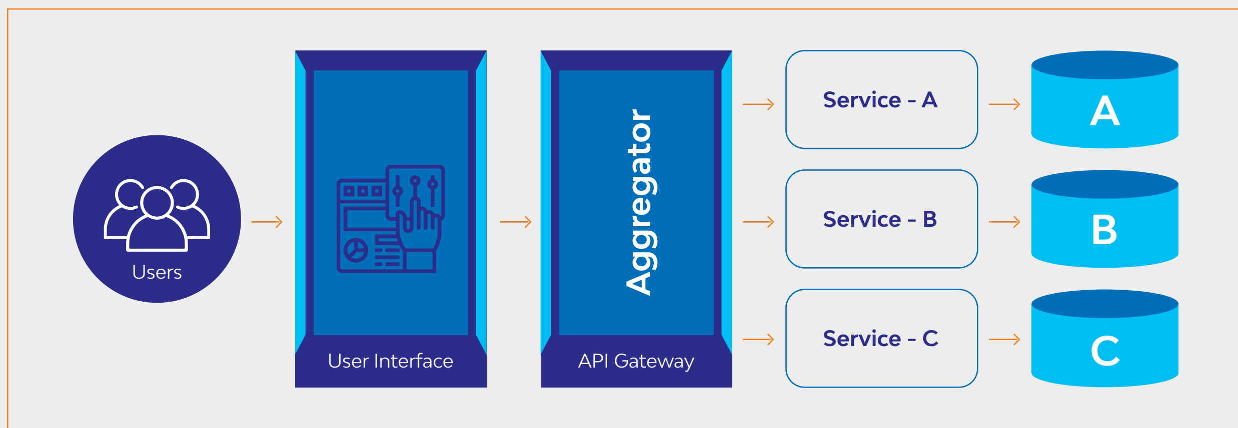
These aspects are a foundation of any stable microservice architecture and need to be considered while designing microservice-based design. All these principles speak of the independent and isolated behavior of microservice needed very much for its implementation.

Design patterns

Some of the common design patterns used in Microservice based architecture are -

Aggregator pattern

Aggregator pattern is a pattern of microservice implementation, where multiple services are invoked to achieve the desired functionality by the same UI page. This pattern is typically useful when we need output by combining data from various microservices. The following diagram displays Aggregator pattern in the microservice design.



As seen in the above diagram, an aggregator is a combination of multiple microservices for a composite functionality. It could be a combined service of some business logic API, or it could even be a UI, which needs functionality from three microservices. As the aggregator microservice is based on DRY principle, we can abstract logic in composite microservices and aggregate that logic in a single microservice.

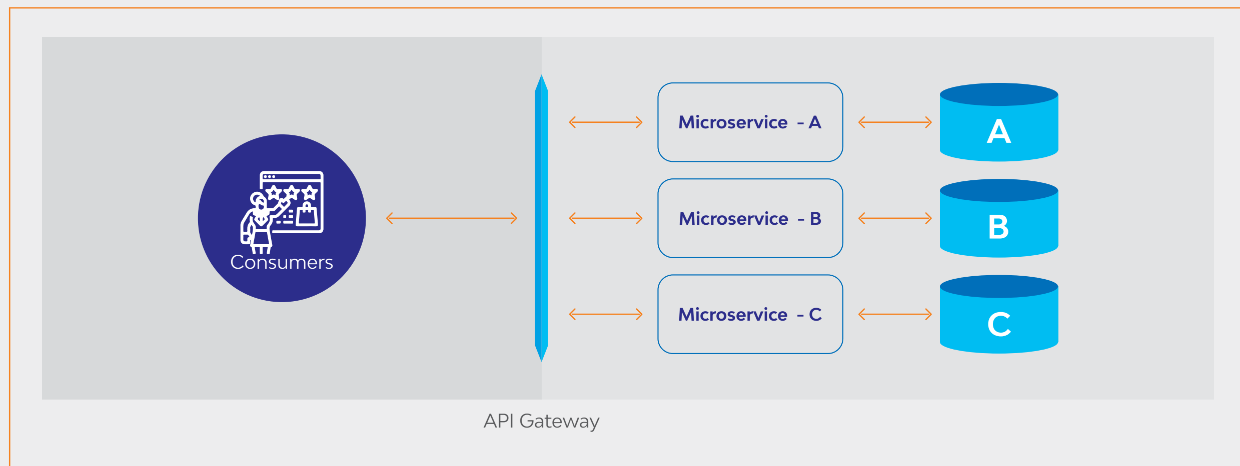
Real world example: Please consider an E-Commerce application, where a customer clicks buy button for purchase of an item. The item exists in his shopping cart. Here the purchase service uses the aggregator pattern to invoke product, purchase, cart, logistics microservices at one go. This means that using aggregator pattern, multiple microservices are called for performing the purchase action.

API gateway design pattern

Microservices are built of concepts of domain-driven design. This means that every microservice is a functionally independent service, which contains a complete functionality. But what happens when an application is further broken down in small autonomous services? Some of the common problems faced are:

- **How is the information requested from multiple microservices?**
- **How is data transformation achieved for various use cases?**
- **How is UI-based data management achieved?**
- **How are multiple protocol requests addressed?**

The common answer to all these questions is API gateway design pattern. 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.

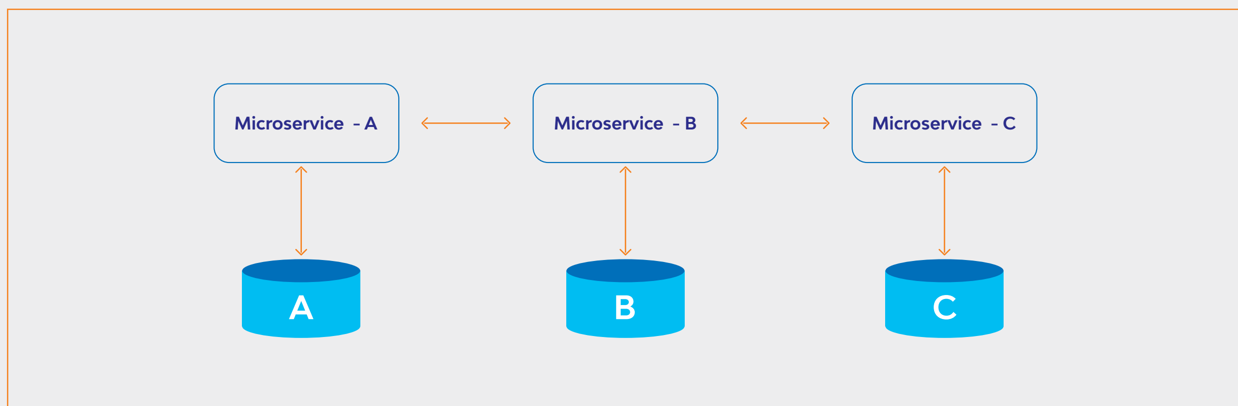


The diagram displays the API gateway design pattern. Microservices use service discovery, which acts as a guide to find the route of communication between each of them. Microservices then communicate with each other via a stateless server i.e. either by HTTP Request/Message Bus.

Real world example: Let's consider a scenario where an E-Commerce application is accessed from multiple devices using browser, as well as native app for viewing products. In such case, an API Gateway pattern can cater to multiple devices and provide the same set i.e. same functional features to these devices irrespective of their form.

Chained or chain of responsibility pattern

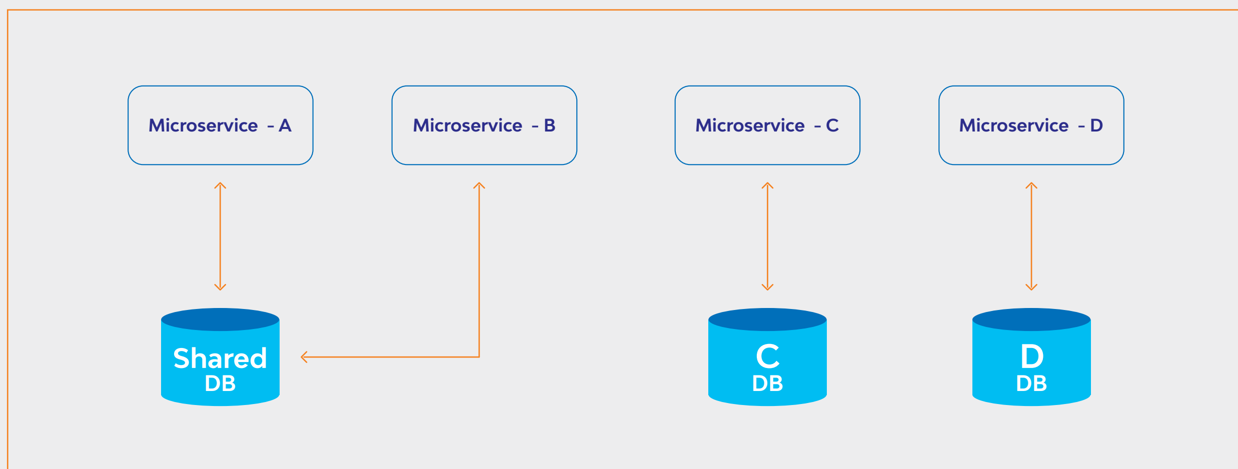
Let's say we have multiple services with dependency on each other. We have service A, B and C. Assuming there is a chain of calls i.e. client calls service A which calls service B and so on. This means that these services are lined up in chain. These services communicate synchronously. This pattern is called chain of responsibility pattern, see the diagram below.



Real world example: The best example of this pattern would be a product purchase options in the Ecommerce Application. The customer selects a product, adds it to the cart and creates an order. Here the responsibility pattern is used where the product, shopping cart and order microservices communicate with each other and complete the purchasing responsibility.

Database or shared data pattern

Microservices are designed using domain-driven design. This means that each microservice has its independent data management, making the functionality completely independent. For every application, there is humongous amount of data present. So, when we break down an application from its monolithic architecture to microservices, it is very important to note that each microservice has enough data to process a request. So, either the system can have a database per each service, or it can have shared database per service.



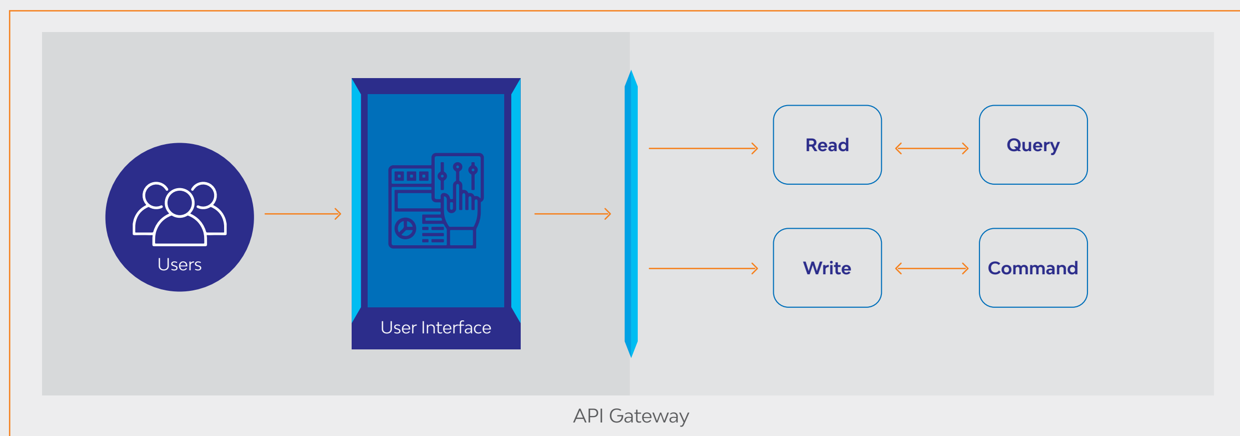
Various problems which may affect the decision of the patterns are as follows:

- Duplication of data and inconsistency
- Different services have different kinds of storage requirements
- Few business transactions can query the data, with multiple services
- De-normalization of data

Real world example: Sometimes, some business processes require data that is owned by multiple services. For e.g. View credit limit will query the customer service and orders will query the open orders. In this case, the queries must be joined to query multiple data and Shared DB pattern will be more helpful in this case. Sometimes the database structure i.e. schema needs to be frequently updated. If we have database per service, then we do not have to update entire database, rather we update only the database of that service. Some examples of database per service are Products Microservice, etc.

Command Query Responsibility Segregator (CQRS) design pattern

We have seen two patterns in the data patterns design pattern i.e. shared database for multiple microservices or independent database per microservice. Digging deeper in the independent database per service pattern, if we want to trigger a complex query, which spans multiple service databases, it's not possible as the data access is only limited to one single database. In such a scenario, CQRS design pattern can be used effectively. In this pattern, the application will be divided in two parts, command and query. The command part will handle all the requests related to Create, Update, Delete - while the query part will take care of the materialized views. This means that that command part will own the create, update and delete while the query part will own the reading statements. The following diagram should help understand the CQRS design pattern.



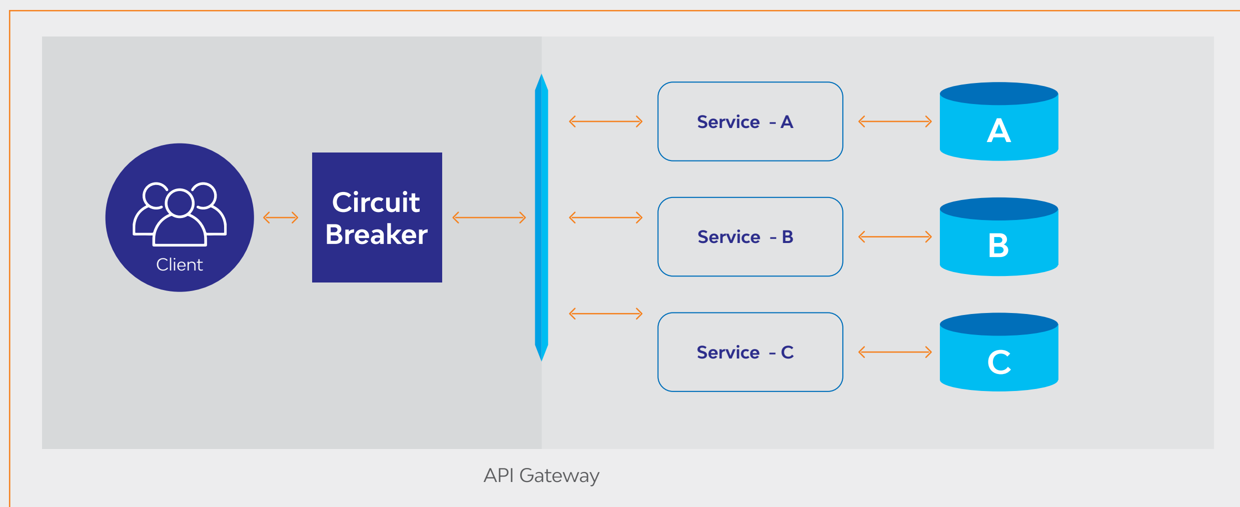
The above diagram displays how CQRS pattern is designed for incoming requests. The read request will go to Query and the edit requests to command.

Real world example: In an Ecommerce application, there are microservices like Products, where we need to populate the products page i.e. query the products service for list of products based on condition. We are not doing any changes to the products here; we are simply retrieving information or products from the catalogue. With multiple users querying the products list, this can become quite a heavy load for the servers, which are performing all actions i.e. CRUD. A better way is to implement CQRS pattern to have query for all the reads separately and command like create, update and delete separately, which are not so frequently used. This also helps in security of the products catalogue in a way, where only legitimate users are given access to the command, while rest all users can use query for read operations.

Circuit breaker pattern

For any working process to be interrupted, Circuit Breaker pattern is used to stop the process of request and response if the service is not working. This is important when we are dealing with service-oriented concepts like Microservices. The caller calls a microservice without knowing if the service endpoint is active, or if service is stopped. When no response is received, it retries the call continuously. The problem gets worse when a client calls multiple microservices, or if there is a workflow involving multiple microservices and one of these services do not respond.

To avoid such scenarios, a circuit breaker pattern is used where the client invokes a proxy rather than a service directly. The proxy will behave as a circuit barrier. So, when the number of failures crosses the threshold number, the circuit breaker trips for a time period. Then, all the attempts to invoke the remote service will fail in this timeout period. Once that time period is finished, the circuit breaker will allow a limited number of tests to pass through and if those requests succeed, the circuit breaker resumes back to the normal operation. Else, if there is a failure, then the time out period begins again.



API Gateway

This pattern is typically useful if a problem occurs with a microservice and tracing needs to be done, as to which service is down by referring logs, circuit breaker pattern will help here.

Real world example: A circuit breaker pattern ensures stable performance in your microservices by monitoring for failures and providing an alternate service or error message. Let's say in Ecommerce application, user clicks on add a product to the cart, and the cart service is down! The system will freeze as the activity will wait for shopping cart service to register the product and if the service is down, the wait will continue forever, freezing the browser and ultimately the transaction. To avoid such kind of scenarios, circuit breaker pattern helps as it ruptures the call, showing user correct message or routing the user to a different service and logging right information to system.

Decomposition pattern

The basic purpose of Microservice is an independent functionality co-existing in a larger enterprise application designed with domain-driven model in consideration. One question we ask ourselves is "Can I break down my application in multiple sub-applications without losing the functionality and gaining advantages of upgradation, performance, teams and deployment?"

The answer to this question is Yes - Decomposition Patterns!

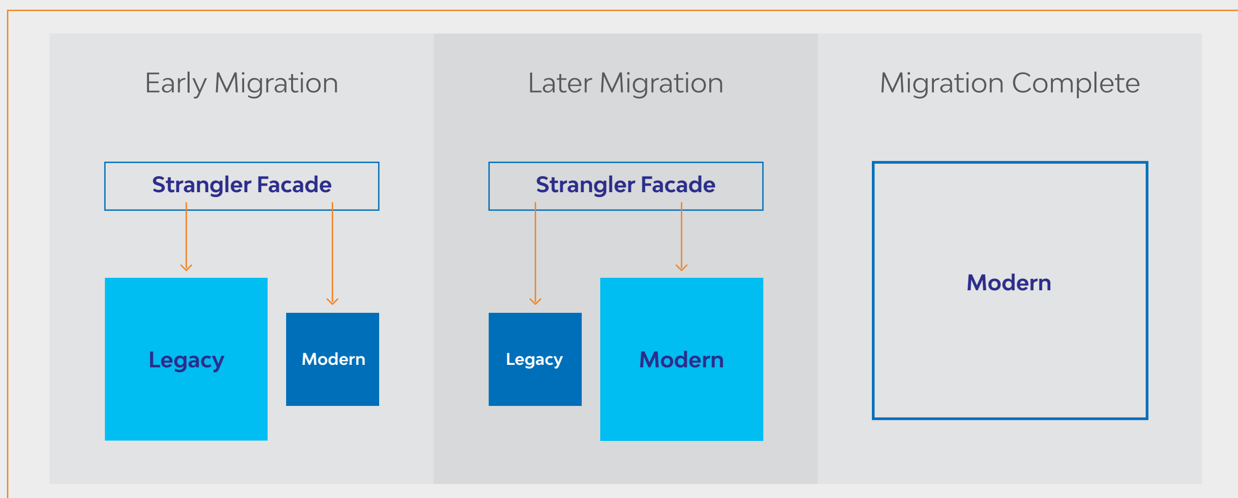
We can use the decomposition patterns based on domain-driven design i.e. business functionality or based on sub-domains. For example, please consider an Ecommerce application, we identify the domains and build services for every function like orders, payments, customers, products. etc. if we are decomposing by business capability.

In the same scenario if we design the application by decomposing by sub-domains, we can have each class-based service. This means that we can reuse the same class Customer for customer management, support, information, etc. Using the domain-driven design, we can break up the entire domain model in sub domains, which have their specific model and bounded context helping developers to create services around scope and this bounded context. The challenges for decomposition pattern we see are - conversions i.e. migrations of large monolithic applications. These challenges arise due to the complexity of these applications, the way they are designed and integrated as a single application. The only way to decompose large monoliths is Strangler pattern or Vine Pattern.

Real world example: Let's consider an application, a monolithic one, where a user visits a customer care website for one of his products. In such a case, user must register himself, his products and other information on the site for receiving support and other help needed to maintain the product. The help application being very large to be a monolith needs to be broken down into independent applications which can work together, however can co-exists separately for better performance and management. Here the decomposition pattern plays important role where the domain can be broken down in sub-domains and services created as per those sub domains.

Strangler pattern or Vine Pattern

Strangler pattern quite represents a vine, which strangles a tree to which it is wrapped around. When a typical migration from legacy application system to any latest architecture is performed, very heavy code rewrite practices are needed by the system. In such cases, Strangler pattern can help deprecating a legacy system slowly over time and adding new functionality incrementally rather than getting entire system offline and do a complete overhaul. What this means is that two separate applications (Original and re-factored) will co-exist side by side in the same space and the refactored application wraps around or strangles the original application until you shut down the original application.



Real world example: The best example of strangler pattern is Monolith to microservices migration. We are moving ancient ecommerce application from monolithic design to microservice design. In such a case, let's consider product catalog as one example, where the strangler pattern can help. While the existing functionality is rendered by the product component, a product service can be created and can co-exist side by side with the main component. Gradually, the features can be switched off the monolith and turned on for Microservice. Gradually, the microservice becomes the active function and replaces the monolith function.

References

DZone Microservices

Design Patterns for Microservices - DZone Microservices

Description: Some design patterns described

Microservice Architecture

Microservice Architecture pattern (microservices.io)

Description: Architecture Patterns

Microservice Design Patterns

Microservices Design Patterns | Microservices Patterns | Edureka

Description: Design patterns for Microservices.

Summary

We have seen the way design patterns are particularly useful for specific design requirements of Microservices. The following table describes the patterns we have discussed and their benefits.

Design Pattern	Benefit
Aggregator Pattern	Combination of multiple microservices for composite functionality.
API Gateway Pattern	Common interface & management for multiple microservices APIs.
Chain of Responsibility Pattern	A chain of microservices with one service calling another.
Shared Data Pattern	Having a single database for multiple microservices with distinct schema per microservice.
CQRS Pattern	Read node responsible for select queries and edit node fo update/edit queries to divide the responsibility and gain performance benefits.
Decomposition Pattern	Domain-driven design helps in decomposing the monolithic application in multiple independent functionalities, which can be managed separately as per the purpose of microservices.
Strangler or Vine Pattern	Translating the legacy application in microservices piece by piece i.e. both the architectures will co-exist till the functionality is duplicated and legacy code is removed.
Circuit Breaker Pattern	Breaking the calling of Microservices in situations, where a microservice is not reachable or not working.

Conclusion

So finally, what we understand from these patterns is important from architect point of view for every Microservice based solutions we deliver. These patterns occur in the daily design activities for any architect while solutioning Microservice based architecture.

These patterns are used frequently while we are designing microservice based architecture and are considering non-functional requirements like discovery, availability, performance optimization etc. There are many more such patterns which you will implement as you design the architecture. I have tried collecting some basic ones which will help you get started with the microservice design.

While taking certain decisions on how the services will communicate with UI and business logic, security and data privacy, failsafe methods, versioning etc. which are speciality of the microservice architecture, these patterns need to be implemented to ensure smooth functioning of the overall system as the system is no more a single system, in fact a composition of several independent systems. These patterns will help you take such decisions.

While designing the microservices, please focus on how the services are going to accomplish the task of working together as a single application though maintaining independent existence and these patterns will be useful while taking important decisions in this process.

About the Author

Rahul Kulkarni

Associate Principal, Enterprise Architect, LTI

As an Enterprise Architect, conducting enterprise analysis, design, planning, and implementation, Rahul believes in leveraging a comprehensive approach for the successful development and execution of strategy. He provides consultation on Application Lifecycle Management (ALM), architecture strategies, opportunities, and problem resolution. Serving as an advisor to senior business management on business and information integration strategies, Rahul has worked on projects from various domains and technologies focused on service-oriented architecture, IoT and web-based solutions. Specialized in Microsoft technologies, Rahul has provided various enterprise solutions in presales as well as delivery solutions.

LTI (NSE: LTI) is a global technology consulting and digital solutions company helping more than 400 clients succeed in a converging world. With operations in 31 countries, we go the extra mile for our clients and accelerate their digital transformation with LTI's Mosaic platform enabling their mobile, social, analytics, IoT and cloud journeys. Founded in 1997 as a subsidiary of Larsen & Toubro Limited, our unique heritage gives us unrivalled real-world expertise to solve the most complex challenges of enterprises across all industries. Each day, our team of more than 33,000 LTItes enable our clients to improve the effectiveness of their business and technology operations and deliver value to their customers, employees and shareholders. Find more at http://www.Lntinfotech.com or follow us at @LTI_Global.

info@Lntinfotech.com

A Larsen & Toubro
Group Company