university of groningen

BACHELOR THESIS

A Systematic Mapping of Microservice Patterns

Author:
Mahir HIRO RAMCHAND

Supervisors:
Prof. Dr. A. Capiluppi
Dr. F.J. Blaauw
Dr. V. Degeler

July 20, 2021

UNIVERSITY OF GRONINGEN

Abstract

Bachelor of Science in Computing Science

A Systematic Mapping of Microservice Patterns

by Mahir HIRO RAMCHAND

Choosing the right architectural style and implementing it can save capital and human resources in a business environment. However, dealing with software architecture usually leads to more questions, such as the alternative patterns, what we should expect after adoption, and the negative aspects of this pattern. The difficulty of this process incentivizes teams to consider how to adopt particular mechanisms carefully. Nevertheless, each business has diverse needs. Thus, with an ever-growing rate of microservice patterns, this only leads to more confusion. This paper attempts to map microservice patterns based on specific metrics, including maintainability and scalability, efficiency, coupling, complexity. The right combination of patterns helps us build more loosely coupled, safer, more maintainable, and scalable services. This paper analyzes a case study for Researchable, aiming to maximize the maintainability of their application. The findings suggest the API gateway, asynchronous messaging, and database per service patterns contributed to a more maintainable architecture.

Acknowledgements

During the writing of this paper, I received a great deal of help.

I would first like to thank my first supervisor, Prof. Dr. A. Capiluppi, who guided me during my writing in formulating my paper and results. Your feedback really helped bring my work to a higher level. You answered my questions within such short time frames, which help speed up my writing process. I would also like to thank Dr. V. Degeler for being a part of this thesis and being available for questions whenever needed.

I would also like to give a great deal of recognition to Dr. F.J. Blaauw from Researchable. You helped me from decided on the topic all the way to my submission. Even with your busy schedule, you offered me all the support I needed without questions. You also helped me add a case study to my paper and be the only student to do their bachelor thesis at a company in 2020/2021. I could not have had a better supervisor, and source of inspiration.

Last but not least, I would like to thank my family: my parents Hiro Ramchand, and Deepa Lokwani, for all the help and support throughout the past 20 years. None of this would be possible without your wisdom and guidance. I would also like to thank my sister, Karishma Hiro, for constantly motivating me throughout this journey. Renly Baratheon once said:

"A man without friends is a man without power"

Thus, my thanks and appreciations go to my friends.

Contents

List of Abbreviations

API	Application Programming Interface
CD	Continuous Delivery
CQRS	Command–query separation
HTTP	Hypertext Transfer Protocol
SOA	Service-oriented architecture
REST	Representational State Transfer
VM	Virtual machine
PaaS	Platform as a service
ESB	Enterprise service bus
SPOF	Single point of failure
OSI	Open Systems Interconnection
DevOps	Development And Operations
DDD	Domain Driven Design
ACID	Atomicity, Consistency, Isolation, and Durability
GDPR	General Data Protection Regulation
SDV	Sports Data Valley
RPC	Remote Procedure Call
CUD	Create, Update, Delete

Chapter 1

Introduction

1.1 Motivation

At the start of the 20th century, monolithic architectures were widely used throughout the industry, and yet to this very day are still a good choice based on certain factors. Nevertheless, monolithic architectures had significant drawbacks: considerable downtimes, scalability, maintainability, extra cost for resource utilization, and difficulty adopting DevOps practices [51]. These disadvantages led to the development of service-oriented architectures (SOAs)

In the mid-2000s, SOA took the IT industry by storm [46]. Many companies adopted this architectural design to provide organizational agility, improve application adaptability and systems interoperability, and reuse legacy assets [29]. Unfortunately, these firms learned the hard way that SOA did not live up to most of these promises but instead led to a hugely complicated and expensive architecture style that took too long to design and implement [46]. The drawbacks of SOA's led to the development of Microservices.

Microservices have since gained attention. Uber, Netflix, and Amazon [58] are just a few companies that have adopted microservices, and have stated they have benefitted from their well-known advantages. Now there is much debate whether the microservices are a subset of Service-oriented architecture (SOA) [14]. However, what is clear is that SOA was introduced before microservices with the same intention to address changing business requirements faster and easier through the medium of services [44].

Deciding on the correct patterns becomes a challenging task that can cost the business extra human and financial capital if not implemented correctly. Thus, figuring out the suitable patterns based on business goals and specific metrics creates the motivation for this research area.

1.2 Example Study

On Dec 14, 2020, Google across the globe suffered from an outage that lasted for approximately 45 minutes [22]. Nobody could access most of Google's services using the features of Account login and authentication to all Cloud services. Even though the event only lasted for 45 minutes, this "outrage" went viral; because of the use of a service-oriented architecture, the entire infrastructure for Gmail, YouTube, Google Drive, Google Docs, Google Calendar, and Google Play were all still running since the issue was isolated to a single service (fault isolation is described in section 2.1 for

microservices). Had it been for a monolithic architecture, all these Google services would have been down. In layman's terms, Google handled the incident very well since most of its services were still running without its authentication service.

1.3 Research Question(s) and Contributions

As the concept of using services becomes even more popular, more companies aim to adopt it. Likewise, Researchable[1], a startup in Groningen, Netherlands, has recently adopted the challenge of implementing a microservice architecture in one of their projects. Researchable's adoption of microservices was to maximize maintainability (more on this in the case study). However, with many patterns for different use cases, it can be challenging to digest the information and compare and contrast the different patterns, precisely what this paper does. That being said, the research question of this thesis is as follows:

Which microservice patterns maximize maintainability?

Answering such a research question should result in a more maintainable codebase consisting of services that can be increase performance, fix bugs quickly and increase usability. Moreover, the thesis aims to explain the building blocks that go into developing a microservice architecture. The goal is not to dive into deep detail of all the patterns but rather just the ones relevant to the case study.

1.4 Thesis Organization

- **Chapter 1:** *Introduction*, this chapter will contain the motivation, research questions, contributions, and, last but not least, an overview of this thesis's structure.
- **Chapter 2:** *Background*, this chapter aims to briefly describe the various alternatives to microservices in terms of software architectures.
- **Chapter 3:** *Supporting Technologies*, this chapter aims to discuss what other components are involved in developing a microservice architecture.
- **Chapter 4:** *Architectural Patterns*, this chapter aims to describe the building blocks of microservices and the available options based on different business requirements.
- **Chapter 5:** *Case Study*, this section will also extensively cover the context and implementations that Researchable will consider adopting in re-developing its new architecture.
- **Chapter 6:** *Conclusion*, this chapter will summarize the paper to simplify the points stated.
- **Chapter 7:** *Future Work*, this chapter will describe what the field of microservice patterns entails and what would have been worked on given more time.

[1] See Researchable's company page for more information regarding the company `https://researchable.nl/`

Chapter 2

Background

The following few subsections aim to compare and contrast the different available architectures used to date. It is essential to understand why the advantages and disadvantages are for each respective architecture. The following subsections will also contain architectural diagrams for simplicity to convey a high-level diagram of the architectural choices.

2.1 Microservices Architecture

Microservices are a relatively new concept. The term was introduced at a conference in 2011 at a Venice workshop of software architects to describe what the participants saw as a standard architectural style that many of them had been recently exploring [18]. In today's age, software is becoming increasingly bulkier and complex, leading to difficulty maintaining, updating, and managing projects. The microservices approach addresses applications becoming increasingly complex and more challenging to manage, update and maintain. The solution is to split up complex software into loosely coupled services. These modular services can be deployed and controlled individually, making large applications more robust to change.

As stated by Martin Fowler [21], "A microservice architectural style is an approach to developing a single application as a suite of small services, each running in its 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". However, the current consensus is that no clear-cut definition of a microservice architecture is. Nevertheless, there are however common characteristics around the term, which are stated by Mohammad Hamzehloui [23] which are as follows:

- Small in size [7, 9, 36]
- Single-responsibility [26, 4]
- Loosely coupled [40, 26, 50]
- Explicitly published interfaces [26, 24]
- Lightweight [26, 8]

However, everything comes with a cost, and microservices are complicated to implement. There exist many possibilities for adopting a microservice-oriented architecture. We will focus on specific patterns to encourage the key driver for Researchable, namely, maintainability. The upcoming sections will elaborate on the advantages

and disadvantages of different architectural choices and then provide some more information about the components of building a microservice architecture. Figure 2.1 shows a high-level overview of how a microservice architecture would look. The central aspect to note is how each service can communicate with each other and their respective data stores through other services.

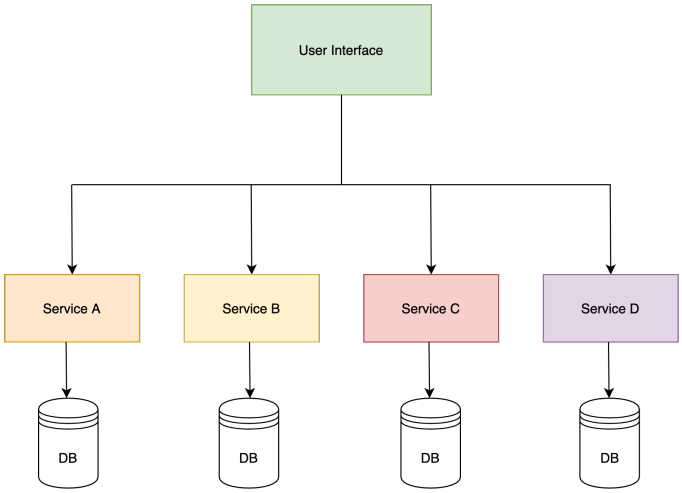

FIGURE 2.1: A microservice architecture in the simplest form

Advantages

The advantages we will look at will cover the main reasons for adoption. More specifically, we shall look at maintainability, fault tolerance, scalability, experimentation, reusability, and finally, continuous delivery.

1. **Services are easily maintainable:** As previously stated, a microservice architecture leads to a codebase with relatively small services that communicate with one another. Therefore, the code is more manageable for developers to understand and less stressful for the workforce about breaking the application when deploying the latest tweaks [6]. Having a smaller codebase in contrast to a monolithic application also allows for easier testability. *Figure* 2.2 demonstrates how different teams can seamlessly work in different teams and test and deploy their small, simple, and reliable applications.

2. **Better fault isolation:** In a monolithic application, one component with an issue will most likely bring down the whole system. Microservices act independently. Indicating that an issue in one service affects only that service, so all the other services will continue as intended. For example, a memory leak [47] would only affect one service.

3. **Scalability:** With an increasing rate of technological adaptation, scalability is a goal of many organizations. In simple terms, scaling a microservice architecture is as simple as adding multiple copies of a service receiving heavy traffic behind a load balancer.

4. **Allows for experimentation:** Due to the size nature of microservices; it can be implemented in a polyglot nature (i.e., different technologies and languages can be used for different services). In layman's terms, one project can use

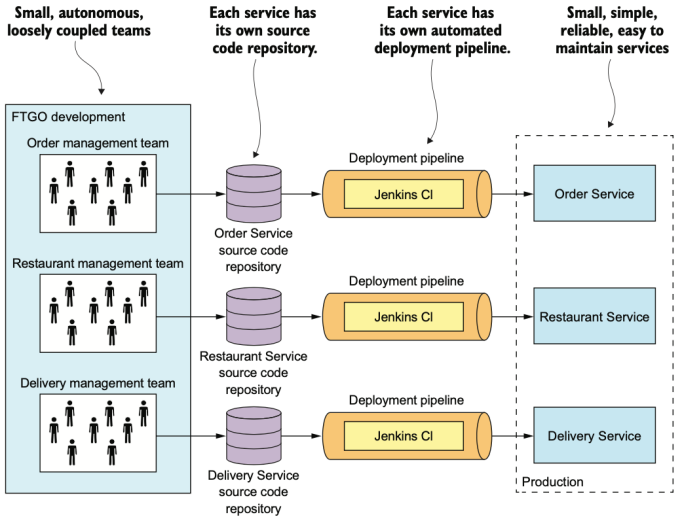

FIGURE 2.2: A microservice application consisting of a set of loosely coupled services [47]

different technological stacks for different services. For example, two separate teams in an organization with different knowledge in their respective tech stacks can work independently with knowledge of their own team's tech stack. This is in contrast to a monolithic system where the initial technological choices affect the future of the product in terms of freedom regarding new languages and frameworks [47].

5. **Reusability:** Reusability is not a promise microservices can consistently deliver. However, microservices can be implemented in a generic way which leads to a non-project-specific service. It is essential to note that this depends on the business workflow design because even a slight dependency on different services will break the microservice concept. Thus the ideal scenario here is to create an instance per custom build as we would like to avoid mixing client-specific sensitive configurations and data boundaries.

6. **Enables the continuous delivery of large, complex applications:** According to Chris Richardson, this is the biggest benefit of microservices [47]. The three reasons which enable this benefit are

   (a) *Testability:* An essential aspect of CD is automated testing. Automated tests are easier to write and faster to execute when using microservices because of their small nature. In theory, this should reduce the number of bugs in an application [47].

   (b) *Deployability:* The consensus is that the team who builds the service is responsible for only that service. Since services can be deployed independently of other services, no coordination is needed with other developers. This is important for companies like Amazon, with thousands of services, can now make a limitless above of changes per day by fixing old bugs, maintaining the codebase, or adding new features [53].

   (c) *Enables development teams to be autonomous and loosely coupled:* As stated previously, teams are usually responsible for the development and deployment of a single service. For example, a couple of services for a

supermarket delivery application would consist of a search recommendation engine, messaging platform, notifications, payments. The lower inter-team dependency encourages a faster development process which is a significant factor in adopting a microservice architecture.

Disadvantages

All architectures have their disadvantages; otherwise, there would be no need to develop other architectures. Microservices are not the default choice and should be used when they can be maintained and developed correctly.

1. **Splitting the application:** As Kalske et al. concluded, decomposition of a new system or an existing monolithic application can be challenging [27]. This is because each business has a different set of requirements and objectives, so there is no universal algorithm or tool to solve it.

2. **Data consistency complexity :** Microservices should not share databases[1]. Instead, they should act independently; this leads to a problem that many developers face as they have to manage multiple databases and, more importantly, data consistency amongst multiple services.

3. **Deploying features spanning multiple services:** Chris Richardson stated that another challenge faced by teams is the careful coordination between the various development teams when a feature spans multiple services. This leads to the creation of a rollout plan that orders service deployments based on the dependencies between services [47].

4. **Operational complexity:** Microservices require a high level of automation to ensure all the services are running smoothly and being monitored. A few of the tools and technologies needed are an automated deployment tool, PaaS, and a docker orchestration platform [47]

2.2 Monolithic Architecture

The most commonly implemented architecture in the past used by the industry is undoubtedly a monolithic architecture; companies like Amazon and eBay used it[2]. The architecture is intended to be tightly coupled, keeping all the logic for handling a request runs in a single process [43]. This architecture may sound outdated but should still be used for projects which do not need the added complexity based on the business requirements. Figure 2.3 shows a high-level overview of how a monolithic architecture would look; the central aspect to note is how the entire application uses a shared database and that the business logic encapsulates the monolith application.

Advantages

The advantages which shall be covered for a monolithic application will consist of ease in testing, deployment, development, and scaling.

[1] See this link to find more data considerations for microservices https://docs.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations

[2] Information about companies which started with a monolithic architecture http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html

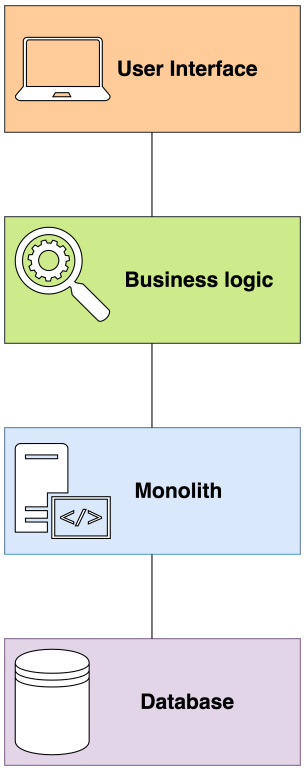

FIGURE 2.3: A monolith architecture in the simplest form

1. **Simplicity in integration testing:** Integration testing is a method of testing how separating functionalities in an application work synchronously with one another. Integration testing is arduous when using microservices because of the per database service used. In contrast, a monolithic application usually has one or a couple more databases with a clear structure (depending on the application).

2. **Deployment:** When an organization only has one application to deploy, the complexity is significantly reduced compared to microservices.

3. **Simplicity in development:** Since developers are used to programming monolithic applications and are less familiar with lots of inter-service communication, less time is spent on the learning process, and more time is used on the development process.

4. **Scaling:** To scale a monolithic architecture, all one needs to do is run multiple instances behind a load balancer that manages the load based on default or customer configurations set by the developer(s).

Disadvantages

One of the main reasons for adopting microservices is the drawbacks of monolith architectures since they often have bad fault isolation. They scale too much, have limited flexibility in terms of a tech stack, and finally, have large size and complexity limitations.

1. **Bad fault isolation:** Essentially, since the entire project is all running in one process, a bug in a single module, such as a memory leak, can potentially crash the entire application, which is catastrophic.

2. **Scaling too much:** As previously stated, to scale a monolithic application, multiple instances need to be running behind a load balancer. This could be considered a waste of resources if only specific system components need to be scaled. For example, in a social media application, the recommendations page may need to be scaled in contrast to the user profile; a monolithic architecture does not offer this flexibility.

3. **Size and complexity limitations:** It is no surprise that after a certain period, the application becomes too large in terms of size and complexity to understand how the code works. This makes it hard for existing employees and new employees, especially junior developers with a lack of experience in megalithic applications.

4. **No tech stack flexibility:** Flexibility in monolithic applications is minimal because the entire application has to be rewritten in a new language or framework, unlike a microservice architecture.

2.3 Service-Oriented Architecture

A Service-Oriented Architecture dates back to older than microservices since they originated from SOAs. Both architectures divide the system into multiple services. However, they do them in different ways [14]. The main goal with SOA is reusability at the enterprise scope. This contrasts with microservices, where it is preferred to reuse code by copying and accepting duplication to reduce dependencies and increase decoupling between components [15]. SOA's are primarily known for their common communication mechanism, namely, enterprise service buses. Despite the ESB, it is also a single point of failure similar to what may occur in a monolithic architecture. Figure 2.4 shows a high-level overview of how a SOA architecture would look. The key message here is how an ESB communicates the multiple services available, and yet these services combined share one data source.

Advantages

The three advantages described below are reusability, polyglot services, and the easability of introducing modifications.

1. **Reusability:** The primary objective of SOA is incorporation reuse, and at the enterprise level, aiming for some level of reuse is critical [15]. SOA focuses on application service reusability, whereas microservices are more focused on decoupling components of a system.

2. **Polyglot services:** Due to the separation in services, teams can develop applications through different technology stacks, which allows for experimentation.

3. **Scope of services allows for easy changes:** Due to the application structure, services in SOA are not as small as services in microservices. However, they are not as big as a monolithic application. This means they can easily be understood and edited by developers to add new features or fix bugs.

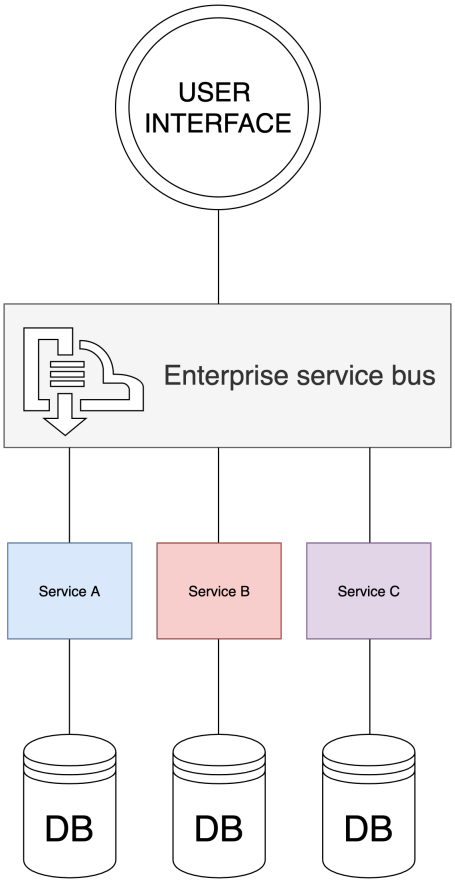

FIGURE 2.4: A SOA architecture in the simplest form

Disadvantages

The downsides of SOA's consist of poor fault tolerance, sizeable initial investment, and problems with the call stack.

1. **Bad fault tolerance:** Unlike in microservices, a bug or corruption that arrives in service(s) can take out the entire system [30].

2. **Large investment:** SOA's generally require a large upfront investment [30] through technology, development and staff deployment. Small/medium companies are less likely to adopt this type of architecture due to the business incentive from a business perspective.

3. **Call Stack:** Unlike monolithic applications where a call stack is a benefit in SOA's, it can turn into a hindrance for loosely coupled distributed system applications [25]. This is because they must wait for a remote component to complete a process before starting execution at the source, which causes distributed architectures to be unresponsive and fragile.

Černý and Donahoo examined the differences between these two architectures and their features, the table below lists the most important key aspects between the two architectures (visible in table 2.1).

Concern	*Microservices*	*SOA*
Deploy	Individual service deploy	Monolithic deploy, all at once
Architecture scope	One project	The whole company/enterprise
Flexibility	Fast independent service deploy	Business process adjustments on top of services
Management	Distributed	Centralized
Data storage	Per Unit	Shared
Scalability	Horizontally better scalable. Elastic	Limited compared to microservices. A bottleneck in the integration unit or a message parsing overhead. Limited elasticity
Unit	Autonomous, un-coupled, own container, independently scalable	Shared Database, units linked to serve business processes. Loosely coupled.
Service size	Fine-grained, small	Fine or coarse-grained
Versioning	Should be part of architecture, more open to changes	Maintaining multiple same services of different version

TABLE 2.1: A comparison of attributes between microservices and SOA. [14]

2.4 Serverless Architecture

Similar to microservices, serverless has no clear definition. However, a *(subjective)* would be as follows: *Serverless computing* can be defined by three characteristics [20]

- *Granular billing:* Costs will only be incurred while the functions are running
- *Near to none operational logic:* Resource management, auto-scaling, and similar operational logic is delegated to the infrastructure provider
- *Event-Driven:* Interactions with serverless applications are designed to be short-lived

The term serverless is not because there are no servers involved. It came about because the cloud provider manages and provisions the infrastructure. Thus, leaving more time for development and not for configurations. The powerful part about serverless is that if an organization currently has a microservice architecture, a hybrid architecture can be used to handle lighter tasks such as resizing an image (since this feature is likely needed once in a while instead of being full active).

Figure 2.5 shows a high-level overview of how a serverless architecture would function in practice. Here, the key message is how the API gateway stands in front of the lambda functions and redirects the flow accordingly. It is also important to note that the functions here are even smaller than microservices and have a smaller goal to complete. Finally, as seen in the figure 2.5, only one database currently exists, but that is just a case-specific implementation.

Advantages

The three main advantages of serverless architectures are the lack of management needed for server provisioning, auto-scaling handled by the provider, and only paying for resources when the functions are invoked.

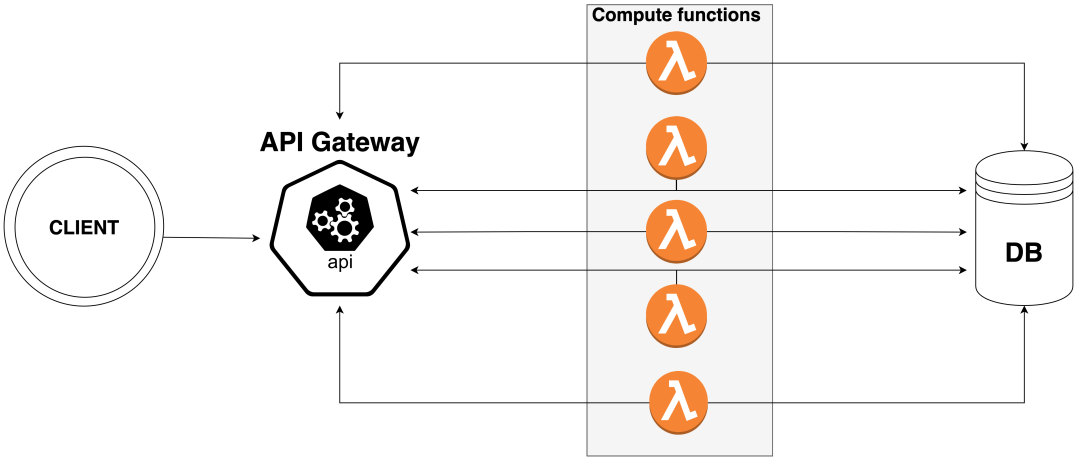

FIGURE 2.5: A serverless architecture in the simplest form

1. **No server provisioning:** A significant advantage of using a serverless architecture is that the cloud provider handles all the provisioning and management of the infrastructure. This leads to less time worrying about setting up a server, thus saving costs and reducing the time to production.

2. **Auto scaling:** Another great advantage of having a cloud provider handle most of the *DevOps* section of a project means auto-scaling is handled directly by the provider based on the load each function receives.

3. **'Pay-per-use' model:** No resources are only charged when a function is invoked [19]. This indicates that using this architecture means payment is only needed for what is used. For example, when auto-scaling, the provider automatically scales up and down functions based on the load it receives, which is cost-effective instead of running multiple instances of a server or projects behind a load balancer.

Disadvantages

Latency, issues with time-consuming tasks, and difficulty with monitoring and debugging are the three main disadvantages of utilizing a serverless architecture.

1. **Latency:** A significant drawback in serverless computing is the startup latency per-invocation to handle a request. These startup latencies can range from tens to hundreds of milliseconds and possibly longer depending on if the invoked function was a 'warm' or 'cold' start and the underlying virtualization technology being used [3].

2. **Monitoring and Debugging:** Since serverless functions run for shorter periods, several orders of magnitude are running, yielding it more challenging to detect bottlenecks and problems. When the functions are completed, the only trace of their execution is the serverless platform monitoring [11].

3. **Issues with time-consuming tasks:** Another reason why serverless computing has not been adopted in massive projects is that serverless is more suited for short-term tasks. For example, AWS lambda offers clients 15 minutes to

execute a task, and if it takes longer, it has to invoke another function; this can be seen as an issue when performing tasks such as video uploading.

Chapter 3

Supporting technologies

3.1 Load Balancers

Load balancing is an efficient technique to distributed application traffic across multiple servers to increase the overall processing load. The easiest way to conceptualize a load balancer is to see a visual representation in figure 3.1. Fundamentally, when the client sends a request to a server, the load balancer first receives the request. Based on a distribution technique such as Round Robin, it forwards the request to a dedicated server, returning a response to the client through the load balancer. The

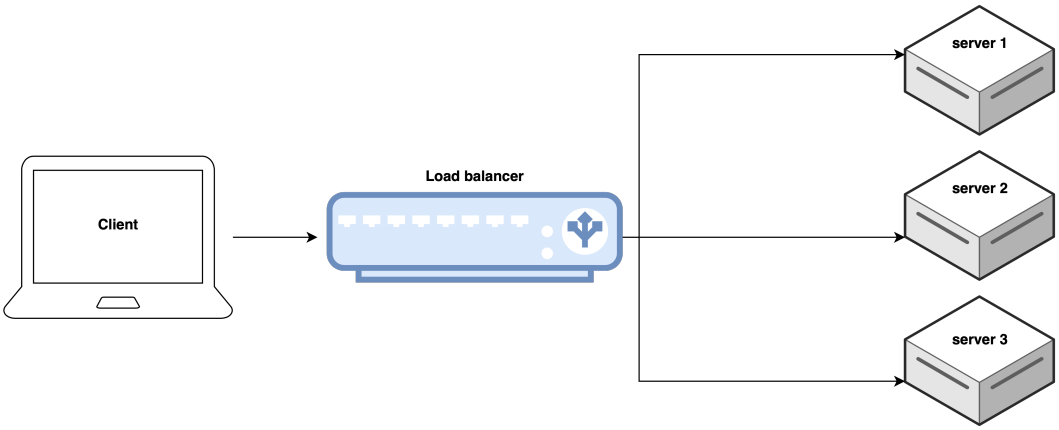

FIGURE 3.1: A simple load balancer distributing work to different servers

following lines show a load balancer for multiple clients trying to request information from a server using the round-robin approach.

1. Client A gets response from **server 1**
2. Client B gets response from **server 2**
3. Client C gets response from **server 3**
4. Client D gets response from **server 1**
5. Client E gets response from **server 2**
6. ...

Two of the main types of load balancers which exist are layer four and layer seven load balancer. A layer four load balancer operates at the transport layer. This approach handles traffic based on network information such as IP addresses and ports,

the routing here is done without reading the contents of the actual messages, so it can not make 'smart' routing decisions. Since it can not read the contents of the request, it can not be used for microservices since it does not know which specific set of servers to forward the request to. In contrast, it can be used for a monolithic architecture since it does not require knowledge about which server to forward the request to.

On the other hand, a layer seven load balancer that works at the OSI model's application layer can make decisions based on more information from the headers, message content, URL type, and cookie data. This means it is suitable for microservices since it can make content-based routing decisions. In summary, load balancing helps reducing downtime and helps make applications more flexible and scalable.

3.2 Messaging Queues

Messaging queues are a fundamental backbone to making asynchronous event-driven applications. For businesses with hundreds to thousands of users, messaging queues are not beneficial since a simple architecture can likely work just as well without a messaging queue. However, companies like Amazon and Alibaba, which receive hundreds of thousands of orders in a short span, require resiliency and scalability, which is precisely what messaging queues offer.

Why do we need it?

Regarding figure 3.2, the order flow with messaging queue would go as follows. If a customer places a new delivery request through the *orderingService*. Following this event, the *cookingService* to start cooking the respective order. An email or SMS will be sent through a *notificationService*, and then the *deliveryService* gets an alert to start the delivery.

1. **Scalability:** Imagine a scenario where *Tesla* had to confirm more than five hundred thousand orders to release a new car model. Utilizing a messaging queue, *Tesla* could increase the number of instances which is receiving the higher load, namely, *notificationService*, it is vital to keep in mind this is only possible due to the nature of asynchronous communication

2. **Reliability:** Consider a microservice application with no messaging queue but rather communication only through a synchronous method like HTTP in contrast to figure 3.2 which uses a messaging queue for communication, making the application asynchronous. If the *cookingService* went down for a while, the application without a messaging queue could result in a cancelation since all the steps are part of a single transaction. Thus a failure in one service would affect the other services causing the customer to reorder the desired order. Whereas, if a messaging queue were used as soon as the *cookingService* is available, it would find the pending order event and process it. Likewise, it can now continue processing all the similar requests while the *cookingService* was down.

3. **Flexibility:** Using queues allow more services to be added into the future with minimum downtime. Assuming a messaging broker like Kafka is implemented a new service, all that is left to do is subscribe to the topic(s) it wants to consume.

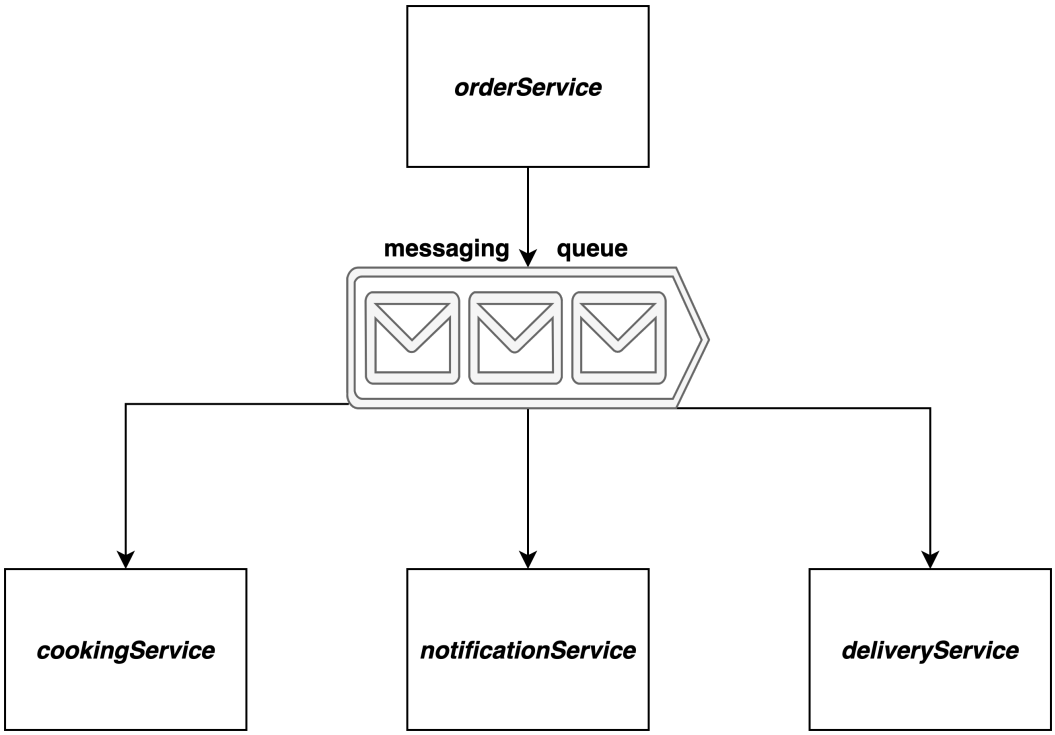

FIGURE 3.2: An application utilizing a messaging queue

3.3 Containerization

Before the invention of VMs, running a single main service was deployed to each machine, meaning if one server failed, other server availabilities would not be affected. The result leads to a massive increase of servers in data centers, which, as we know today, is unmanageable with many servers being underused, especially if a service was deployed and only dealt with light computations. The following method that arose was VMs when a physical server is divided into multiple virtual servers. There are many drawbacks to using VMs. While this paper will not elaborate on the issues, two of the main issues are as follows. Firstly, there is the overhead required to set up and manage a VM, and secondly, each VM is self-contained: higher memory and storage requirements and making software development more complex.

One method of deploying microservices is through containerization. Put containerization is a lightweight method of virtualization applications [41]. Sharing the kernel of the host OS in containerization also means avoiding infrastructure overhead of an entire OS, as only the necessary resources, such as installations, dependencies, and code, are provided. Essentially, containerization leads to faster starting and stopping timings which provided better performance results compared to VMs [28].

Figure 3.3 demonstrates an example of a containerized application. At the app, layer containers are an abstraction that packages code and dependencies together. The diagram shows that multiple containers can share the same OS and machine while running in isolation from the other apps.

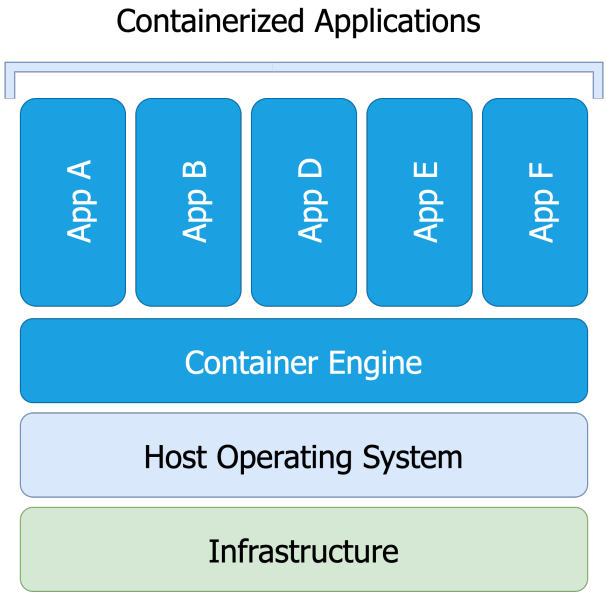

FIGURE 3.3: The architecture of a containerized application[1]

3.4 Orchestration

While many companies still rely on traditional deployment methods, Container orchestration is becoming an increasingly popular decision of any system since it automates deployment, scaling, networking, and management of containers. At first thought, this might sound redundant, especially when managing a few containers requires minimal effort. However, companies such as AWS, which deploy hundreds of thousands of containers at any given period, require orchestration. Microservices with containerization make security, scaling, resource allocation, load balancing, and similar features more easily operatable.

An ever-so-increasing rate of containers managing all these resources (memory, CPU power, and similar resources) becomes a significant challenge. Container orchestration work by providing several instances of services, and then the orchestration tools take care of the essential tasks. For example, Kubernetes has three essential functions for scalable microservice applications:

- **Load balancing:** Kubernetes can auto-scale and distribute the network traffic so that the deployment is stable.
- **Resource management:** Treats a cluster of machines as a pool of CPU, memory, and storage volumes, thereby combining the machines into a single machine [47].
- **Self-healing:** Kubernetes can auto-heal by replacing a failed container with a new one and killing containers that do not live up to user-defined health checks.

Since microservices are often deployed with containers, these containers need to be managed from a more prominent grouping method orchestration tools are an essential component when building microservices.

[1]See the url for the original figure `https://www.docker.com/resources/what-container`

3.5 Scaling Cube

X Axis Scaling:

X-Axis scaling, also known as horizontal scaling, is the easiest method of scaling. The idea is to run multiple instances of an application that stands behind a load balancer, as seen previously in section 3.1. This should lead to a decrease of the load on a single application and spread the load. Horizontal scaling is also how monolithic applications can scale since the other methods require attributes that only SOAs or microservices can offer. The drawback of allowing fast scaling of transactions is the cost of duplicate data and functionality. In summary, X-axis scaling is not suitable for data scalability but can allow for transaction scalability [1].

Y Axis Scaling:

Y-Axis scaling, also known as functional decomposition, works by splitting up an application, such as a monolith, into services. Functional decomposition is needed when there are large data sets without relations between each other. In summary, functional decomposition allows for efficient scaling of transactions, extensive data records, which can help is fault isolation [1].

Z Axis Scaling:

Z-axis scaling, also known as sharding, is similar to X-axis scaling because it also runs multiple instances of an application. However, the main difference is that each instance is now only responsible for a subset of data. Sharding is functional when there are significant and similar data sets with a quickly growing rate of records. Sharding works by identifying customer information such as `customerFirstName` based on this information; one can partition the data and service routing. In summary, Z-axis scaling is excellent for scaling customer bases and similar large data sets, which can not be solved using Y-axis scaling [1].

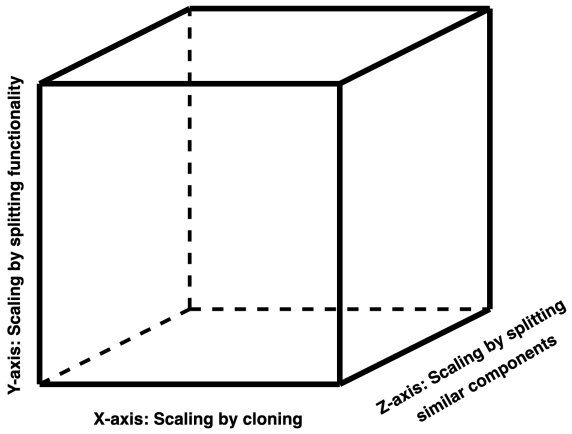

FIGURE 3.4: The cube depicts three distinct methods of scaling an application: X-axis scaling aims to distribute the load over multiple identical instances; Y-axis scaling aims to decompose an application into smaller components, namely, services; Z-axis scaling aims to distribute requests based on information contained in the request.

Figure 3.4 visualizes the three distinct methods of scaling an application. Overall, X

and Z-axis scaling help improve availability and an application's capacity with the cost of increasing application complexity [35]. Nevertheless, as previously seen to deal with application complexity, the solution is to use Y-axis scaling. It is essential to clearly state that the three methods are not alternatives but instead go hand in hand. Martin L. Abbott and Michael T. Fisher created figure 3.5 which perfectly demonstrates the effects of the three methods. In figure 3.5, X-axis split functions by duplicating instances here. Y-axis split works by separating functionality. Finally, Z-axis works by sending load to different instances based on the customers the first letter.

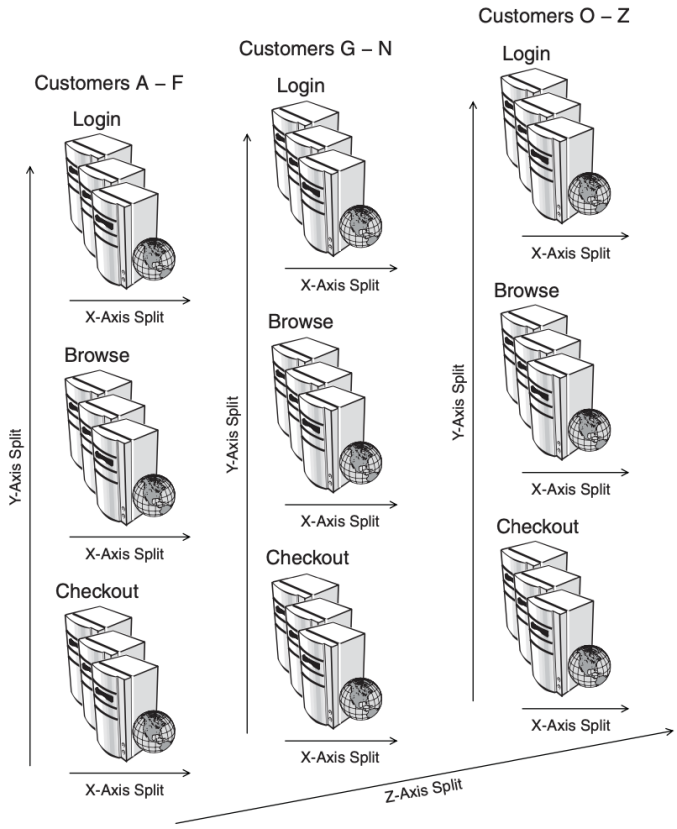

FIGURE 3.5: This figure shows an example of how load is split up on the three axes. [2]

Chapter 4

Architectural Patterns

4.1 Overview

The following sections many microservice patterns. In order to categorize each attribute with a simple value a mapping is created using the rules mentioned below:

- **GREEN** → this pattern encourages the respective attribute
- **RED** → this pattern discourages the respective attribute
- **YELLOW** → this pattern could encourage or discourage the respective attribute based on different aspects
- **GREY** → there is not enough information in the literature to make a decision

In the following subsection, we will describe a handful of patterns for different contexts. Each subsection has its patterns in comparison to its alternatives. By the end of this chapter, the following questions listed below should be answered in their respective sections.

1. How should one re-architecture a large codebase into a microservice-oriented architecture? → *decomposition patterns*
2. With so many services present, how should services interact with each other? → *integration patterns*
3. What different methods of communication should be used to store and query data in a safe, reliable, and efficient way? → *database patterns*
4. How should one make sure applications are operating reliably? → *observability patterns*
5. How can we design more secure and performant applications? Is there a safe way to update services? → *miscellaneous patterns*

Combining all the sections, we will have a collection of over more than twenty patterns ranging from the initial setup of microservices to maintaining them in production.

4.2 Decomposition Patterns

This section will elaborate on the different approaches to moving a monolithic application into a microservice-oriented application. However, some of these ideas can also be utilized to design microservices from scratch. However, this generally

leads to duplicating the existing functionality and additional risk from a business perspective. Hence, it is better to decompose an existing application, which is what this section aimed to solve. To put it simply,

"If you do a big bang rewrite, the only thing you are certain of is a big bang!" - Martin Fowler

The following section will discuss decomposition patterns: Decompose by Business Capability, Decompose by Subdomain, Decompose by Transactions, and Decompose by service per team pattern with the respective drivers.

Stability	Coupling	Easability of split	Reduction in the number of microservices	Improved Availability	Data consistency

Each of the following metrics was selected for a specific use case or requirement, which are described as follows:

- *Stability:* Is important in the case that business functions change in an organization which affects the structure of microservices.
- *Coupling:* Having a loosely coupled set of services is crucial to ensure an organization is benefiting from the advantages of microservices.
- *Easability of split:* Is important because the opportunity cost of moving from a monolith to microservices may be too costly even to consider adopting.
- *Reduction in the number of microservices:* Being able to complete transactions with fewer microservices is almost always a better alternative due to the effort extra effort otherwise needed to deploy, clone, and manage extra services.
- *Improved Availability:* Is crucial in microservices to avoid downtime because it is very costly for a business. If we assume the system has an uptime of 99.999% our downtime is calculated so follows:
  - 365 days a year and 24 hours per day = 8760 hours per year
  - 8760 hours per year $\times$ 60 minutes per year = 525,600 minutes per year
  - 99.999% uptime means the downtime accepted is = 0.001%
  - 525,600 minutes per year $\times$ 0.001% downtime accepted = 5.256 minutes
  - Assume there are a 100 services, 100 services $\times$ 5.256 minutes = 525.6 minutes = **8.76 hours** of downtime

  8.76 hours of downtime very expensive when the cost can range between $140K to $540K per hour[1].
- *Data consistency:* It is important to ensure the state of data in different microservices are synchronized often to not provide or work with stale data.

[1]See the link below for more information about the cost of downtime https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/

4.2.1 Decompose by Business Capability

A standard method of decomposing a monolith can be by business capabilities. The term 'business capability' has a variety of definitions. Thus Michell [32] concluded a uniform definition accounting from different industries; "Potential of a business resource to produce customer value by acting on their environment via a process using other tangible and intangible resources." Thus, the modularity of a service is defined by business capabilities. Figure 4.1 shows how an E-commerce monolith application be broken down into six different microservices based on the decomposition by business capability pattern.

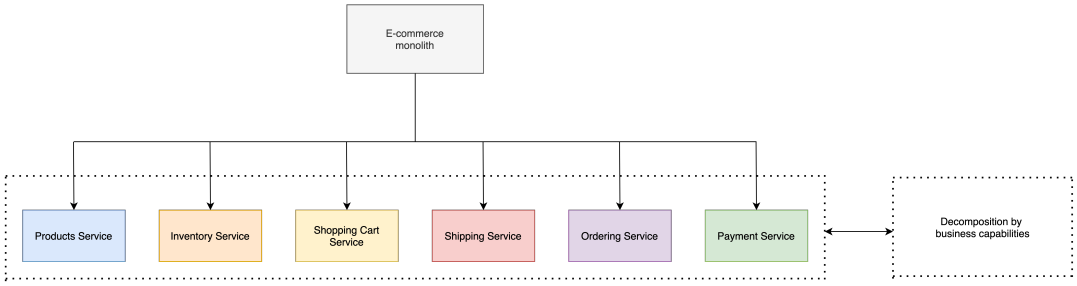

FIGURE 4.1: The architecture of splitting an application based on decomposition by business capability

The benefits are that services will remain in a loosely coupled manner. Additionally, given that business capabilities are relatively stable, the architecture of the microservices will similarly have a stable architecture (**stability**) [47]. Finally, development teams are created not around technical features but rather ones that deliver business value. On the other hand, understanding the business capabilities can be pretty complex and challenging to map to a set of services (**easability of split**). This pattern will also lead to a tightly coupled architecture with the overall business model (**coupling**). The result is table 4.1 which illustrates this pattern offers the attributes. The (**reduction in the number of microservices**) is implementation specific, so the metric outcome is neutral. Lastly, a widespread problem that is encountered in most decomposition patterns in maintaining data consistency across services [47] (**data consistency**).

Stability	Coupling	Easability of split	Reduction in the number of microservices	Improved Availability	Data consistency

TABLE 4.1: Ranking metrics based on decomposition by business capability attributes

Advice when to use: Decomposing by business capability should be used if a team has enough insight into the organization's business units with additional experts on subject matters for each business unit[2].

[2]See AWS documentation for more information about decomposition by business capability
https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/decompose-business-capability.html

4.2.2 Decompose by Subdomain

Decompose by Subdomain models microservices around Domain-Driven Design (DDD). The DDD approach to software development builds around Object-Oriented Analysis and Design principles. This pattern works well when there is an existing monolithic system that has well-defined boundaries between subdomain-related modules. Hence, one can repackage the existing modules in the monolith into microservices without the expense of rewriting code. Figure 4.2 is similar to figure 4.1 where the e-commerce application is split using decomposition by business capability pattern; however, after this, is done the e-commerce application is decomposed into subdomains. As seen below, the *Ordering Service* and *Payment Service* are broken down into smaller microservices.

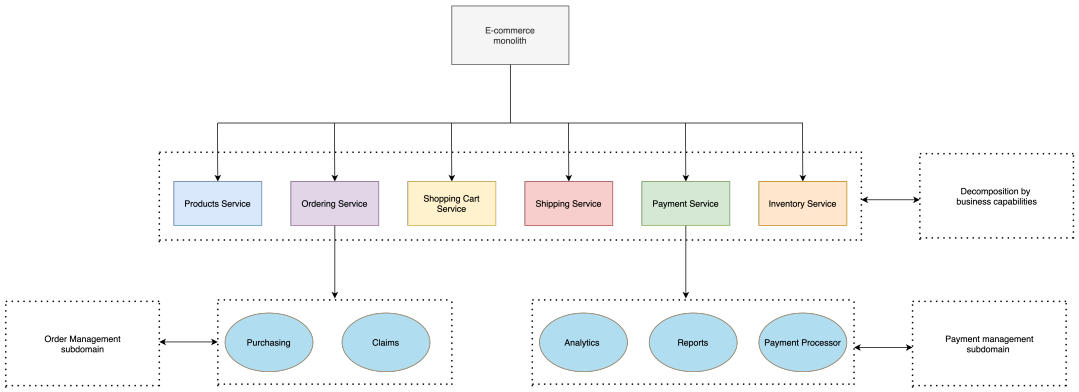

FIGURE 4.2: The architecture of splitting an application based on decomposition by subdomain

The benefits are similar to decomposition by business capability since the architecture is stable since the subdomains are relatively stable; likewise, services are cohesive and loosely coupled that, most importantly, provides maintainability (**stability & coupling**) [47]. The difference is that systems become more scalable and predictable due to the nature of subdomains. Finally, the last advantage is that development teams are created not around technical features but rather ones that deliver business value. On the other hand, splitting domains can lead to an excess in microservices, creating integration and service discovery issues, thus, affecting maintainability (**reduction in the number of microservices**). Last but not least, similar to decomposition by business capabilities, business subdomains are challenging to identify because they require an in-depth understanding of the overall business (**easability of split**). Ultimately, table 4.2 summaries the information into a digestible format. Similar to decomposition by business capability (**data consistency**) is an issue when there is an increase in the number of microservices present.

Stability	Coupling	Easability of split	Reduction in the number of microservices	Improved Availability	Data consistency

TABLE 4.2: Ranking metrics based on decomposition by subdomain attributes

Advice when to use: Since this pattern expands on the decomposition by business

capability pattern as described above, the team planning to adopt decomposition by subdomain must adhere to the previous attributes mentioned. Additionally, it is advised only when enough teams can be responsible for single services present in an organization to design such a complex architecture.

4.2.3 Decompose by Transactions

Microservices need to communicate with each other to complete a single transaction. This can introduce problems if one of the services goes down in the middle of a transaction, usually referred to as two-phase commit problems. The solution is to group microservices that are involved in a single transaction.

Figure 4.3 demonstrates how an e-commerce monolith can be broken down into multiple microservices based on transactions. In this case, the *Purchase Service* is comprised of many otherwise different services. An example of order flow is if a user clicks "checkout", then the cart first must be retrieved, followed by an order creation. After which, payment is required, which sends a notification, updates the inventory status, and initiates the shipping process. Note that the *Notification Service* is used after the payment is complete, and for the *Marketing Service* hence it is extracted into a separate service.

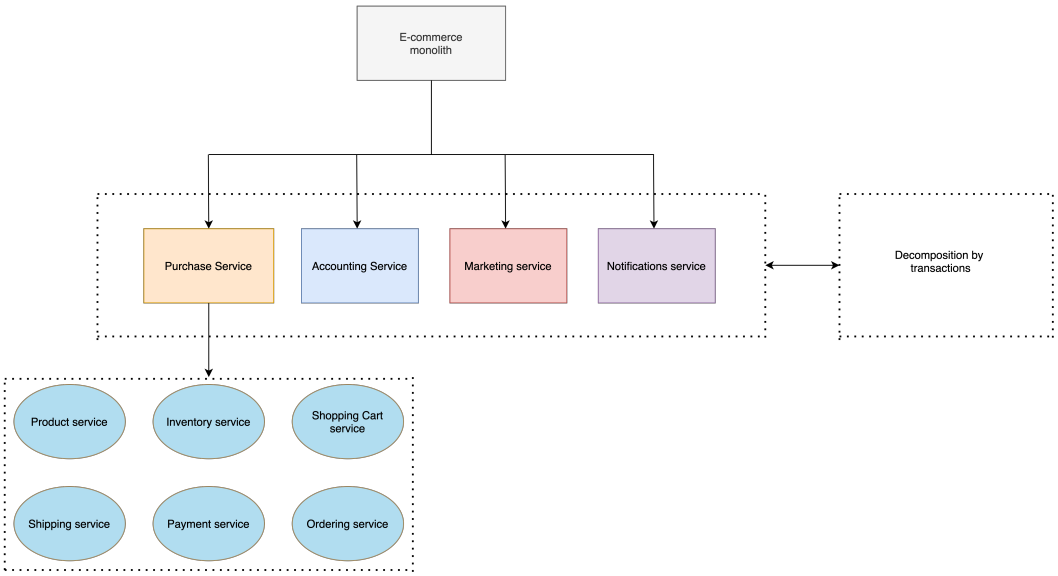

FIGURE 4.3: The architectural structure of the decomposition by transactions

The first benefit is that data consistency is less of an issue since services are grouped according to transactions (**data consistency**). Decomposition by transactions also reduces the number of microservice communications needed, which leads to a reduction in latency issues. Since there are fewer microservices alone but instead grouped (**reduction in the number of microservices**), there is an improvement in availability (**improved availability**). Hence, fewer microservices result in a smaller chance of downtime in a single service, which would otherwise interrupt the ongoing transaction causing a halt in the system, affecting availability. As seen in Figure 4.3, the more modules become packaged together, the more likely a distributed monolithic

application arises (**coupling**). There is also an increase in complexity due to multiple features being implemented in a single microservice. Last but not least, the final disadvantage is that transaction-oriented microservices will expand if the number of business domains and their dependencies is large (**stability**). In conclusion, table 4.3 summaries the information into a digestible format. AWS documented the reduction in the number of microservices, improved availability, and data consistency (all green metrics) in their *AWS Prescriptive Guidance*[3].

Stability	Coupling	Easability of split	Reduction in the number of microservices	Improved Availability	Data consistency

TABLE 4.3: Ranking metrics based on decomposition by transactions attributes

Advice when to use: An organization, should utilize this pattern if response times are crucial for customers (for example, in a reservation service). If different modules do not create a monolith after being packaged, this pattern is likely suitable for a team's needs[4].

4.2.4 Decompose by service per team pattern

Unlike the previous decomposition patterns, decomposition by service per team breaks down a monolith into microservices managed by individual teams. Each team is solely responsible for the code base of a business capability. In other words, each team is responsible for the development, deployment, testing for their capability and ideally interacts with other teams to negotiate APIs. Figure 4.4 shows how a monolith is split into microservices that are managed, maintained, and delivered by individual teams.

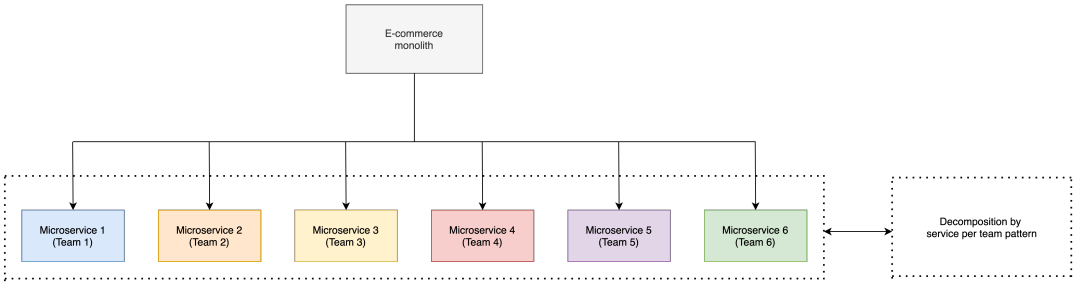

FIGURE 4.4: The architecture of splitting an application based on decomposition by service per team pattern

In this pattern, microservices are not shared by multiple teams, which is a big advantage. Meaning development can occur independently with minimal coordination (**coupling**) [48]. Which inevitability means teams can quickly innovate and iterate product features. Conversely, aligning teams to business capabilities can be challenging (**easability of split**). If there are circular dependencies between teams, extra

[3]See AWS prescriptive guidance for more information regarding the attributes of decompose by transactions https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/decompose-transactions.html

[4]See footnote 3

effort is required to deliver coordinated application increments. Table 4.4 summaries the content above into a simple format. The stability of the split is case specific, and thus, depends on the business functions and how they change over time (**stability**). Regarding a reduction in the number of microservices and improved availability, there is not enough research on the pattern. So much so, that there are no literature reviews or books which contain information on the pattern (**reduction in the number of microservices & improved availability**).

Stability	Coupling	Easability of split	Reduction in the number of microservices	Improved Availability	Data consistency

TABLE 4.4: Ranking metrics based on decomposition by service per team attributes

Advice when to use: Applications that have 5-9 people [48] working on them with multiple teams who want to be autonomous and loosely coupled should consider this pattern. On the one hand, it brings added complexity, but on the other hand, it brings the drawbacks discussed above in table 4.4.

4.3 Integration Patterns

After understanding the different methods of decomposing an application, integration patterns are the next important task to understand. This is essentially about adding more services or structuring an application such that the services can quickly, but more importantly, efficiently communicate with each other and the client. Thus the following section will discuss the following integration patterns: Aggregator Pattern, API Gateway Pattern, Chained Microservice Pattern, Branch Pattern, Shared Data Microservice Design Pattern, and finally, the Asynchronous messaging design pattern. The table below contains the key metrics, which will define the characteristics of each integration pattern.

Encapsulation	Scalability	Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability

Each of the following metrics was selected for a specific use case or requirement, which is described as follows:

- *Encapsulation:* helps to make sure that it does not expose the internals of a service which is important.
- *Scalability:* Is important if there is an increase in load, which starts to affect the service performance.
- *Reduced complexity to orchestrate data:* Handling transactions between multiple services can be very complex, so it is important to know which patterns minimize the complexity.
- *Low latency:* It is important to reduce the communication delay between services.
- *Debugging code:* It is important to be able to isolate issues in services quickly.

- *Maintainability:* Is an important requirement for a good codebase, which can be sustainable in the long run.

4.3.1 Aggregator Pattern

A microservice architecture can consist of tens to hundreds of services. The issue, however, is that to get information about a specific product requires many extra calls to other services to gather the information required. The *Aggregator Pattern* functions by collecting pieces of data from various micro services and returns an aggregate for processing. As seen in figure 4.5 the new aggreator service can keep a joint collection of information from different services in its own database. The various services can then push events onto a messaging queue/ bus to which is then recorded in the aggregator.

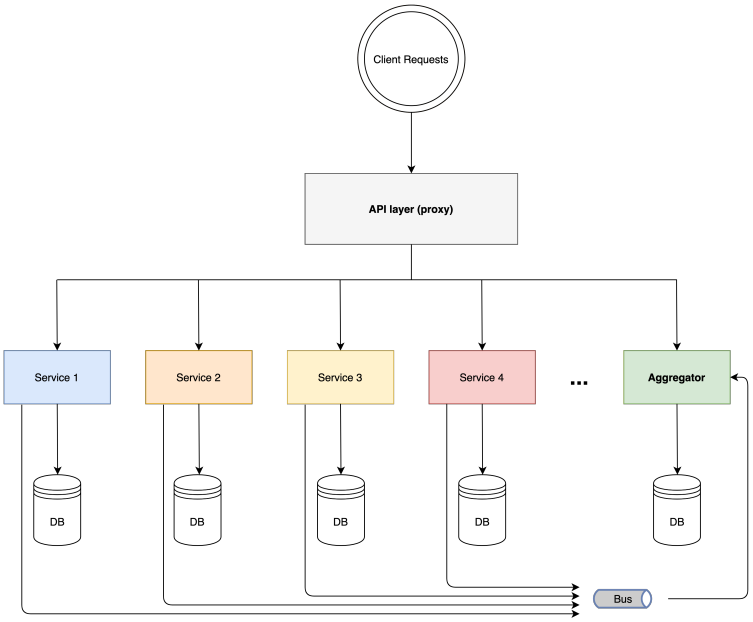

FIGURE 4.5: The architectural structure of the aggregator pattern

A significant benefit is that there is a single point of access for the microservices. This pattern also leads to a higher degree of encapsulation (**encapsulation**) [39]. However, it most importantly can scale in the x- and z-axis (**scalability**) [39]. The cost of implementing the aggregator pattern is that there is a higher level of latency in communications between microservices, which means there is an increase in complexity to orchestrate data (**low latency**). Last but not least, this pattern requires much experience to implement; thus, the aggregator pattern can turn into a bottleneck antipattern if it is not optimized. Unfortunately, the maintainability is not perfect here as there is an extra service to maintain (**debugging code**). However, from another perspective, there is a central service to add service calls so it could enhance maintainability (**maintainability**). Receiving data from n services with maintaining and updating a database to hold this information is generally very difficult. Thus this metric is not a good reason for adoption (**reduced complexity to orchestrate data**).

Advice when to use: A good idea when to utilize the aggregator pattern is when the outcome of possible outages result in a low business casualty. Apart from that,

Encapsulation	Scalability	Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability
	x y z				

TABLE 4.5: Ranking metrics based on the aggregator pattern

if the benefits outweigh the downfalls of this pattern, and the teams have experienced developers due to the complexity of orchestrating data the pattern is useful to consider.

4.3.2 API Gateway Pattern

An API Gateway is the single entry point that aggregates the calls to the individual microservices. While this may sound very similar to the Aggregator Pattern, there are some distinct features. Most importantly, this new service does not store data but instead becomes responsible for API composition, request routing, and new features such as authentication. From figure 4.6, the API gateway can serve different clients across several communication channels. The API gateway can expose an API depending on the client's needs. The gateway can then handle requests using API composition and then invoking multiple services and aggregating the results. Diving deeper, figure 4.7 demonstrates the multiple API calls made after the client requests for `getOrderDetails()`

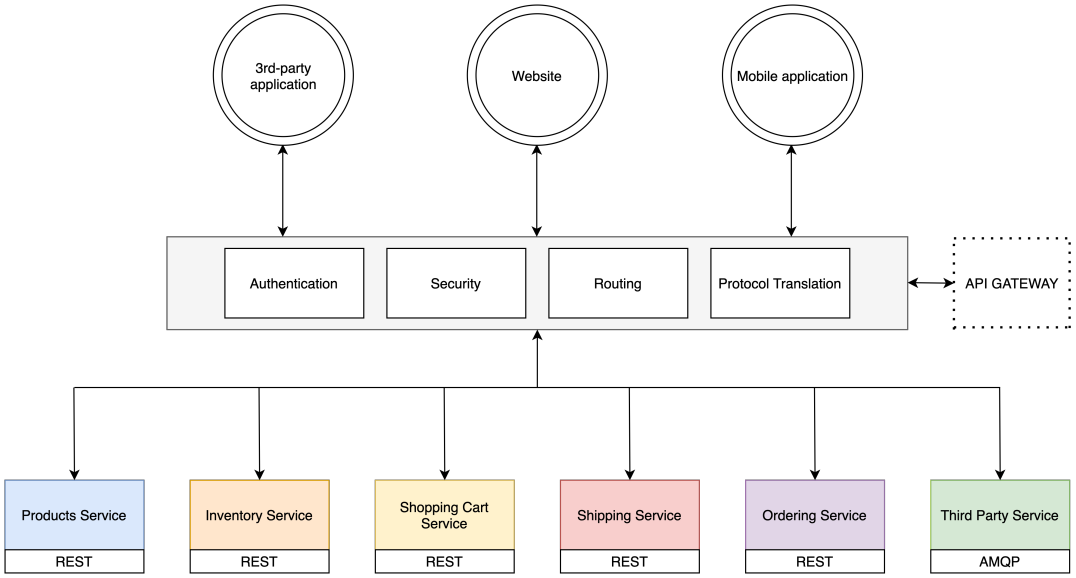

FIGURE 4.6: The architectural structure of the API gateway pattern

Similar to the aggregator pattern, encapsulation of the internal structure is offered by this pattern (**encapsulation**) [33]. When considering the (**low latency**)[5] metric, introducing a new layer to the overall architecture decreases the response time, however as services scale stating whether this amount is significant is problem specific. The API gateway simplifies the client-side code and reduces the number of network calls

[5]See AWS prescriptive guidance for more information regarding the attributes of decompose by transactions https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-integrating-microservices/api-gateway-pattern.html

needed between the client and the application. The downside, however, is the extra component that has to be managed with development and deployment, which also increases the difficulty in debugging (**debugging code**). The most problematic issue that could turn into a bottleneck is that updating the API gateway is not lightweight since developers will be forced to wait in line to update the gateway when they need to update the gateway API to expose each microservice endpoint. Similar to the aggregator pattern, there is extra management and maintainability effort required to keep the gateway running as there is an extra service present to handle, however adding a few microservices to act for the API gateway should require low maintenance assuming service discovery and orchestration of the services are automated(**maintainability**) [57]. Last but not least, scaling API gateways are generally not an issue considering providers such as *AWS* can optimize configurations to help your application (**scalability**) [16].

Encapsulation	Scalability	Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability
	x y z				

TABLE 4.6: Ranking metrics based on the API gateway pattern

Advice when to use: If a client has low latency requirements, as well as if the number of dependencies for a microservice is manageable and does not grow over time, this pattern is useful[6]. Another reason for adoption is that the API gateway offers a cleaner interface for clients to interact. Most importantly, where high availability is required, this pattern should be used to handle load balancing as previously mentioned, but it can also be combined with rate limiting and throttling [60].

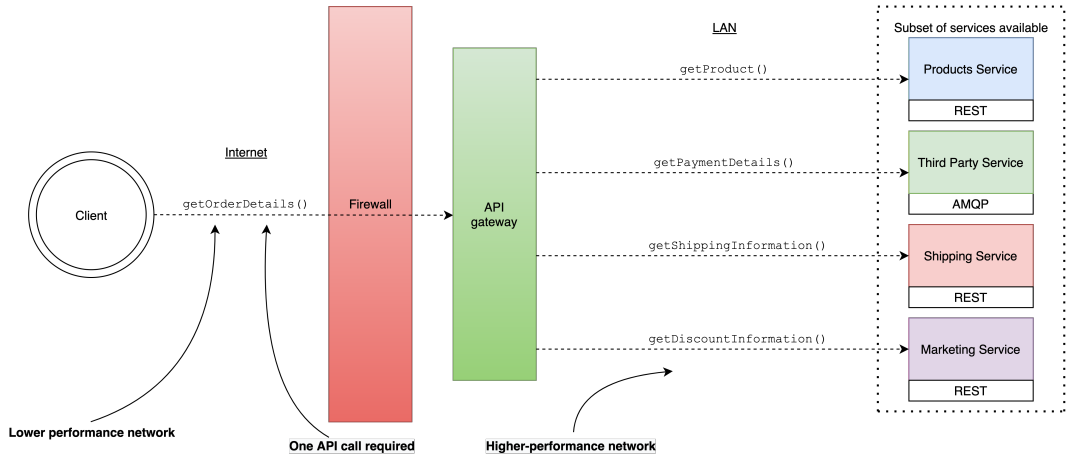

FIGURE 4.7: This figure shows how an API gateway often does API composition, enabling a client such as a mobile device to efficiently retrieve data using a single API request. [47]

4.3.3 Chained Microservice Pattern

The Chained microservice pattern is a sub-pattern on the Aggregator pattern mention at the start of this section. The similarity lies in that it aims to provide a single

[6]See footnote 5

access point for information [39]. While the aggregator works by evaluating and triggering concurrent processes for the microservices responsible for composing the response to the request, the chained pattern works by services calling other services and combined their previous responses to return a concatenated response without an orchestrator.

Regarding figure 4.8, imagine a scenario where **Service A** receives a request but needs to call more endpoints to get all the client's information. Thus it executes a request for **Service B**. Interestingly enough; **Service B** calls **Service C** to receive all the information **Service A** requested. Likewise, **Service C** calls **Service D** to complete its information request by **Service B**. However, **Service D** does not require any more information, so **Service C's** information is passed on to **Service B**, which is then returned to **Service A** and ultimately sent back to the client.

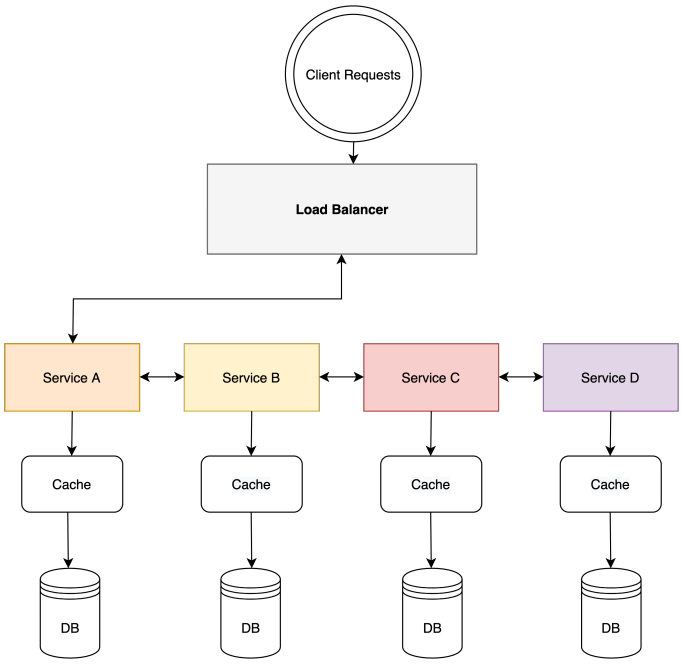

FIGURE 4.8: The architectural structure of the chained microservice pattern

The most apparent benefit of the chained microservice pattern is the practicality of implementation; because of the chain call's synchronous nature, understanding the network calls becomes less mixed up. We prefer not to use asynchronous communication since the response composition control could have a callback system, which grows in complexity and compromises scalability [39]. The obvious issue is that this pattern struggles to handle asynchronous communication increasing complexity and compromising scalability. Debugging code and understanding data ownership are more downsides to this pattern which grow in complexity when using the chain microservice pattern (**debugging code & reduced complexity to orchestrate data**). Last but not least, the longer the communication channel is, the higher the chance of an increase in latency areas can occur (**low latency**). When a service goes down in the middle of communication, it is a good idea to implement the circuit breaker pattern described in section 4.6.2.

As the chained microservice pattern does not require an extra service for data orchestration, maintainability is a positive metric. However, since the communication is synchronous, adding a new service to the chain would require more maintainability effort. (**maintainability**). While this pattern also offers encapsulation of access to microservices (**encapsulation**) [56], it also offers independent scalability in all three axes, all of which are linked to the number of instances of a service that may be accessed using a proxy to reroute requests as stated by V. F. Pacheco (**scalability**) [5].

Encapsulation	Scalability			Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability
	x	y	z				

TABLE 4.7: Ranking metrics based on the chained microservice pattern

Advice when to use: The chained microservice pattern is helpful if the current scope of the application is too large to add in extra microservices. If the current scope of an application has crucial functional requirements to be scalable, then this pattern is possibly suitable for their needs.

4.3.4 Branch Pattern

The Branch pattern is an extended version created from the aggregator and chained design patterns to better serve the application's business layer, and thus, aims to combine the positive aspects of both patterns. The pattern allows simultaneous request/response processing from two or more microservices. To put it simply in figure 4.9 we can see the developer is allowed to configure service calls dynamically. All the calls in this pattern can occur in a contemporary fashion. Hence, service A can call Service B and Service C simultaneously.

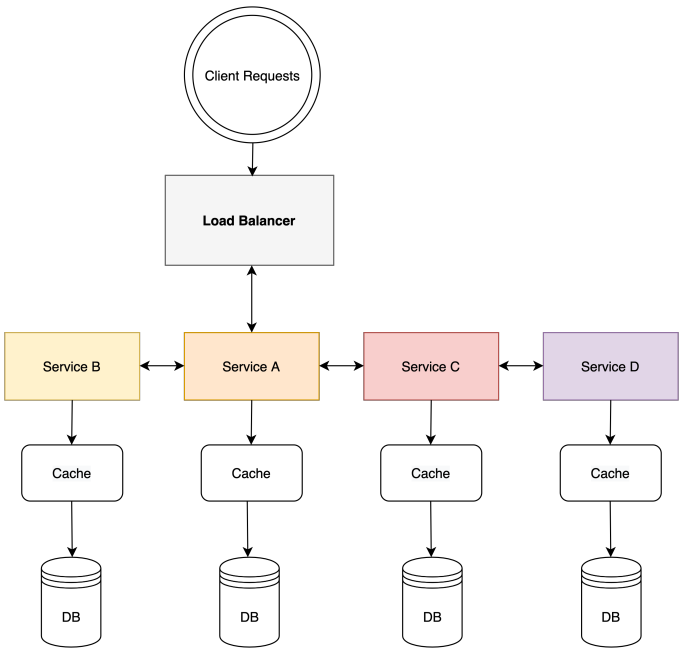

FIGURE 4.9: The architectural structure of the branch pattern

The branch pattern also offers scalability in all three axes, with the feature of encapsulating access to the microservices (similar to the chain microservice pattern) (**scalability & encapsulation**) [39]. However, the main advantages offered in the adoption of this pattern are the flexibility of implementation, compositional ability, and orchestration [39]. Since the pattern combines the aggregator pattern and the chain pattern, we have many similar disadvantages, the first one being an increased chance of latency with long communication chains (**low latency**). Debugging code is also growing in complexity with this pattern, but so does the difficulty in understanding the codebase (**debugging code & Reduced complexity to orchestrate data**). Thus, while the branch pattern can solve many communication problems, it is vital to consider the use case where this will apply. In terms of maintainability, there is a lack of literature on the branch pattern (**maintainability**).

Encapsulation	Scalability			Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability
	x	y	z				

TABLE 4.8: Ranking metrics based on the branch pattern

Advice when to use: It is first advised to begin with the chained microservice pattern. However, at some point, one realizes that the chain is too long and can be possibly be split up using this pattern, thus creating a sense of concurrency in the microservices calling.

4.3.5 Asynchronous Messaging Pattern

The foundation of the Asynchronous Messaging Pattern was discussed in section 3.2. The Asynchronous Messaging Pattern is primarily based on communication that does not require immediate attention (i.e., no response back).

This pattern is unquestionably the most scalable of all due to the asynchronous character of microservices. Thus not only is it very scalable, but it is also independently scalable in all three axes (**scalability**)[7]. Additionally, it also provides encapsulation of accesses to microservices to benefit from lazy process information. Conversely, understanding the architecture initially can be quite complex (**reduced complexity to orchestrate data**). Debugging asynchronous architectures can also be hard to manage as there is complexity in the monitoring of requisitions (**debugging code**). While adding a messaging queue can increase latency in an application, there are solutions to try and overcome this within the messaging systems configurations themselves (**low latency**). Valdivia et al. concluded that the three main benefits of this pattern are reliability, maintainability, and performance efficiency (**maintainability**) [57]. Last but not least, this pattern also encapsulates access to microservices (**encapsulation**) [39].

Encapsulation	Scalability			Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability
	x	y	z				

TABLE 4.9: Ranking metrics based on the asynchronous messaging pattern

[7] See the link below for more benefits regarding messaging queues https://aws.amazon.com/message-queue/benefits/

Figure 4.10 may look familiar as it utilizes the aggregator pattern where *Service A* acts as an orchestrator, collection information from service *Service B* and *Service C* and communicating directly back to the client. The noticeable difference is that *Service A* also writes data to the message broker, and *Service D* consumes the data in the queue, which may then synchronously communicate with the client.

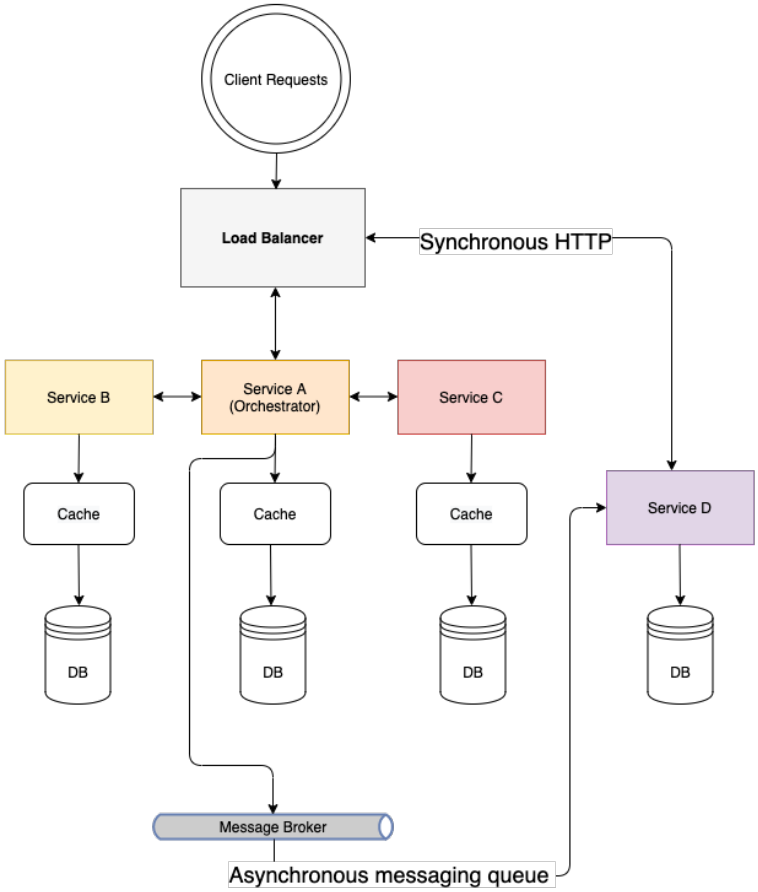

FIGURE 4.10: The architectural structure of the asynchronous messaging pattern

Advice when to use: Asynchronous communication is helpful when an event requires multiple consequences based on what occurred. For example, some more sub-events should occur after the payment is completed, such as a delivery service processing the shipping information. The notification service should alert the customer about the status of the payment and other similar occurrences.

4.4 Database Patterns

Databases are a big part of any application. Hence, this section will cover different methods for handling data with the following patterns: Command Query Responsibility Segregation, Event Sourcing, Database per Service, Shared Database per Service, Saga Pattern. The table below contains the key metrics, which will describe the characteristics of the respective patterns.

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

Each of the following metrics was selected for a specific use case or requirement which are described as follows:

- *Efficient Implementation:* It Is important to be able to query information quickly.
- *Reduced complexity:* It is important that querying the data from different microservices in an understandable method.
- *Maintainability:* Is an important requirement for a good codebase, which is sustainable in the long run.
- *Reduced monitoring:* Is important to ensure the databases work without lots of trouble.
- *Scalability:* Is important if there is an increase in load which starts to affect the read and write performance of the database (s)
- *Loosely coupled:* Is an important aspect to consider because it helps with the current and future development.

4.4.1 Command Query Responsibility Segregation

The CQRS pattern is about separating the create, update, and delete operations from the get operations. The idea is that the query-side model retains updated information by subscribing to the events published by the command side (CUD operations). Due to the difficulty in implementing queries from multiple services when using the database per service pattern, the CQRS pattern helps implement queries.

The most significant benefit CQRS offers is the improved separation of concerns, which means commands and queries are not handled by the same data model (**efficient implementation**). Seo et al. demonstrated the power of efficiency in CQRS queries [52] being approximately eleven times faster than non-CQRS queries in the context of their weather data application. Furthermore, CQRS enables an efficient implementation of queries, as previously mentioned. Last but not least, querying becomes a possibility in an event sourcing-based application. The most considerable price to pay for these advantages is the complex architecture, since developers not only need to write query-side services that update and query the views but also manage and operate different databases (**maintainability & reduced complexity**) [45]. To repeat, having two separate databases could also mean having a relational database for writing and a non-relational database for reading, which can increase the performance of an application and scalability (**scalability**) [45]. Also, with event-driven architecture, another general problem that arises is replication lag, although there are known solutions to overcome the problem (**reduced monitoring**).

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

TABLE 4.10: Ranking metrics based on the CQRS pattern

From figure 4.11 it is visible that the service utilizes event handlers that are subscribed to domain events published by services who own their data, which is how

the leading service holds direct access to the query database can perform efficiently.

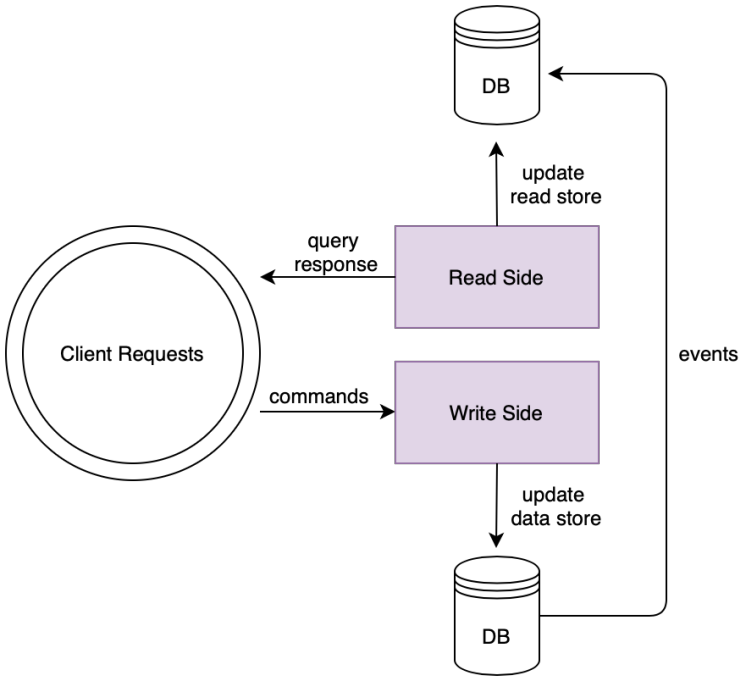

FIGURE 4.11: The architectural structure of the CQRS pattern

Advice when to use: The CQRS pattern should be used alongside event sourcing since the two patterns complement each other. When systems have different loads on the read and write aspects of a database, CQRS helps with separation concerns. When complex domain models are present where reading data needs query into multiple data stores, the pattern offers many benefits, as seen above. Although if the volume of events is insignificant, this pattern should not be used.

4.4.2 Event Sourcing

The idea behind event sourcing is not to store the state but rather to store events in a system that can be replayed when required. Since each event is irremovable and immutable, no updates or deletes are needed enabling good write performance these factors lead to better performance (**efficient implementation**) [12].

Event sourcing is reliable because it publishes events whenever the state of aggregate changes. Another great addition to using event sourcing in a platform is that it preserves the history of aggregates, which is valuable for auditing and regulatory purposes. Since event sourcing stores the history of an application, it helps developers implement new features with existing data, which otherwise would not be possible since traditional applications do not preserve this information. On the other hand, there is added complexity since it has a different programming model with a learning curve that may take developers time to adjust to (**reduced complexity**). Given that events are stored for an indefinite period, deleting data for GDPR reasons can be challenging (**maintainability**). As previously mentioned, this pattern goes along with CQRS for the sole reason that queries are complex and most likely inefficient. Thus there is additional development required for implementing queries

with CQRS. Event-sourced systems strive for the minimum amount of synchronous interaction, which results in loosely coupled scalable systems (**loosely coupled & scalability**). As Diakov et al. stated, one of the most glaring benefits of event sourcing is its scalability [17]. According to Valdivia et al., the two main benefits found in their literature research were security and performance efficiency (**efficient implementation**) [57].

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

TABLE 4.11: Ranking metrics based on the event sourcing pattern

From figure 4.12 we can visualize a shift from previously stored as a state into a system that stores events now.

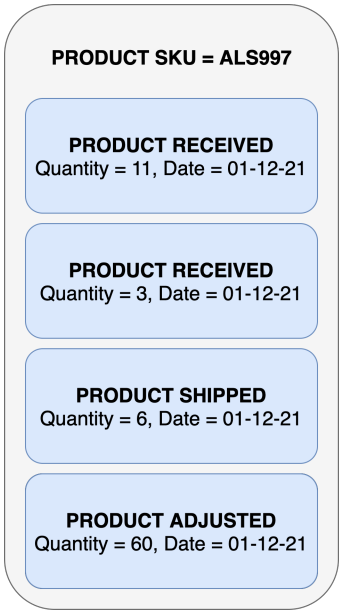

FIGURE 4.12: An example of how event sourcing stores events in a system

Advice when to use: Event sourcing is most commonly used for classic message-driven or event-driven systems (e-commerce and reservation systems). This pattern should also be considered when building a large-scale system that needs a secure audit log. However, adopting it only for having a secure log is not a good idea, considering the maintenance and complexity required to keep an event sourcing system running.

4.4.3 Database per Service

This pattern may sound simple in practice since it essentially keeps each microservice's persistent data private to that service and accessible only via its API. Thus, a service transaction only involves its database. Figure 4.13 shows an example of how multiple services keep track of their private databases.

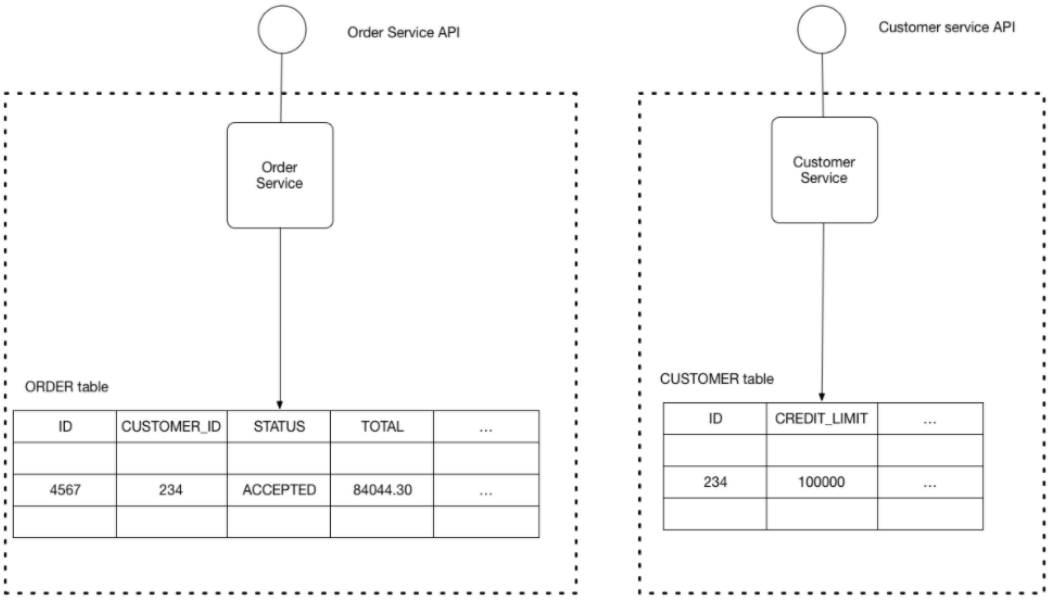

FIGURE 4.13: This figure shows how how the database per service pattern would function in an application [48]

The database per service is offers enhanced loose coupling since each database only serves one service making, them independent (**loosely coupled**) [55]. Some databases may require higher security requirements. For example, Researchable has to comply with GDPR. Hence having separate databases for sensitive data can increase security safety. However, it may also lead to a more efficient implementation since different types of databases can have different use cases (**efficient implementation**).

Last but not least, this pattern leads to a more granular control of scaling (**scalability**) [56]. The downside is that implementing complex transactions and queries that span multiple microservices has becomes very challenging (**reduced complexity**). Managing multiple forms of databases, such as relational and non-relational databases, become a challenge for companies, so maintainability is not a good feature offered by this pattern (**maintainability**). The extra databases need to be maintained and monitored, which is additional overhead to consider (**reduced monitoring**).

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

TABLE 4.12: Ranking metrics based on the database per service pattern

Advice when to use: The database per service pattern should be taken advantage of where teams require complete ownership of their microservices for scaling and velocity development. It is also recommended to use this pattern in large-scale enterprise applications. Nevertheless, in small-scale applications where one team manages all the microservices, this pattern should be analyzed since the disadvantages possibly outweigh the advantages for this specific case.

4.4.4 Shared Database per Service

As the name suggests, this pattern leads to the same database being shared by multiple microservices. This pattern often leads to microservices become a distributed monolith which can become a nightmare for developers. While this pattern means sharing a database, it does not mean single tables should ever be shared among multiple microservices (which should be avoided).

From the diagram, it is noticeable that implementing this pattern leads to minor refactoring of an existing code base, with the benefit of maintaining and managing a single database (**reduced monitoring**). As previously seen, this pattern is complex because of interdependencies amongst existing microservices. However, that helps reduce the need to redesign the existing data layer (**reduced complexity**) [56]. This pattern also helps with maintainability due to the decrease in configurations and overwatch required (**maintainability**) [56]. Most importantly, this pattern helps enforce data consistency by using transactions that are ACID. However, sharing databases means tightly coupling the available services (**loosely coupled**) [34]. If the entire application relied on a single shared database, it would turn into an Anti-pattern. Although, it is acceptable to use this pattern in parts of a more prominent application to gain the benefits listed above. In terms of scalability, it can highly depend on the configurations and other technical details in a codebase (**scalability**) [37].

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

TABLE 4.13: Ranking metrics based on the shared database pattern

In figure 4.14 one can see how multiple services share the same database but not the tables

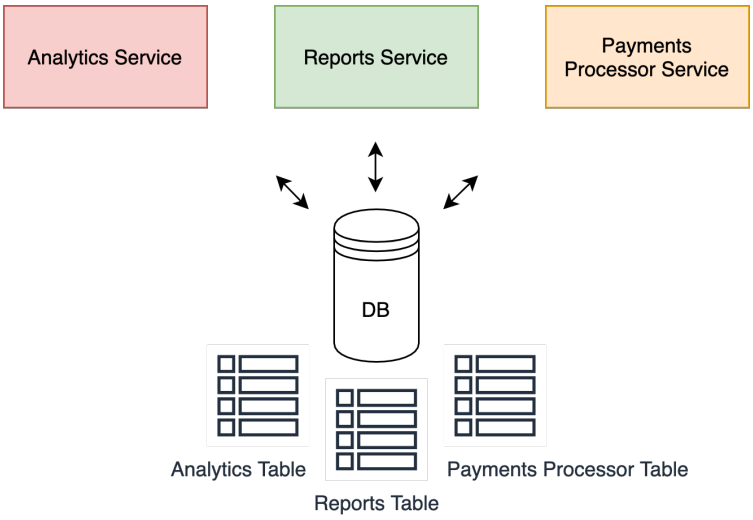

FIGURE 4.14: The figure depicts how a single database with multiple tables can be utilized my multiple services when using the shared database pattern

Advice when to use: A large majority of the advice on when to adopted is listed

above. Nonetheless, it is essential to reiterate that if an organization prefers not to refactor an existing code base by many changes due to business requirements or other limitations, this pattern can be encouraged.

4.4.5 Saga Pattern

A well-known issue in the microservice world is dealing with distributed transactions (when a transaction spans multiple services). The saga pattern can be applied after one has used the database-per-service pattern. Essentially a saga is a chained series of local transactions, where a local transaction can trigger another (local) transaction through an event. The fail-safe mechanism is to execute a series of compensating transactions that undo the changes made by the initial local transactions.

There are two approaches to developing sagas:

1. **Orchestration based:** Local transactions take advantage of a messaging queue to publish domain events that trigger local transactions in other services.
2. **Choreography based:** An orchestrator informs the participants what local transactions to execute.

Orchestration based

Each implementation approach has its own merits and demerits. However, first, the implementations work as follows. Figure 4.15 demonstrates how the orchestration-saga approach would function in practice for an e-commerce application. The order flow would act as follows (The services communicate with each other through a messaging broker):

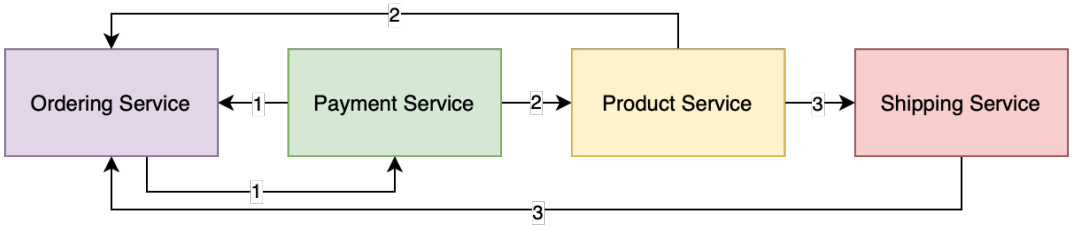

FIGURE 4.15: The order of communication which occurs in the saga orchestration is shown in this this figure

1. The inventory service provides information to the ordering service about the availability.
2. The Ordering Service asks the Payment Service to complete the transaction for the goods ordered, with a returned response stored in the Ordering Service.
3. The Payment Service, if successful, calls the product service to get the right products into the pipeline for delivery. Which in turn returns the information of that Ordering Service
4. Finally, if the Product Service was successfully called, the Shipping Service will be alerted, and then the Ordering Service will receive an update with the required information.

The orchestration-saga approach offers a simplistic easy to understand approach (**reduced monitoring**). Since each service can implement an API that the orchestrator invokes, there is less coupling because the orchestrator does not need to be aware of the events released by the saga participants (**loosely coupled**) However, this approach can quickly get out of hand if one keeps adding extra steps between a transaction process (**reduced complexity**). On average, the recommended number of steps is two to four. Having another service that communicates with the rest of the services also introduces a single point of failure (when there is one coordinator) (**reduced monitoring**). The upside is that cyclic dependencies cannot occur; the price is that handling state and coordinating transactions can be complex (**maintainability**).

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

TABLE 4.14: Ranking metrics based on the saga orchestration pattern

Choreography based

From figure 4.16 the steps taken for services to communicate to each other based on a messaging queue.

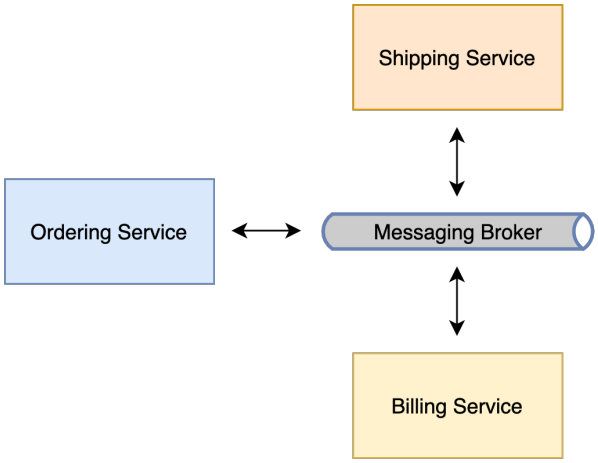

FIGURE 4.16: When using the saga choreography pattern the services communicate through a messaging queue to communicate to each other

1. The Order Service pushes an *ORDER_PLACED* event to the queue
2. The Billing Service consumes an *ORDER_PLACED* event from the queue
3. The Billing Service pushes an *ORDER_BILLED* event to the queue
4. The Shipping Service consumes an *ORDER_BILLED* event from the queue and creates a shipping level
5. The Shipping Service pushes a *SHIPPING_LABEL* event to the queue

6. The Order Service consumes the *SHIPPING_LABEL* event and updates the status to *READY_TO_SHIP*

Each service will produce and consume events of other services and decide what actions to take based on the message type (**efficient implementation**). Thus there is no central coordination. Unlike the orchestration-saga, the choreography-saga-based approach offers no SPOF, which aligns closely with the theme of microservices. The lack of an orchestrator (**loosely coupled**) [47] in the application implies no additional coordination logic is needed, which improves maintainability (**maintainability**) [49]. The downside, however, is that the workflow can get complex as the architecture grows (**reduced complexity**) [47]. Not having an orchestrator also increases the risk of a cyclic dependency arising between services. Since services publish events when they perform CUD operations on business objects, monitoring does not have to become necessarily complex and can depend on the implementation (**reduced monitoring**).

Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled

TABLE 4.15: Ranking metrics based on the saga choreography pattern

Advice when to use: As previously stated, cyclic dependencies are an issue with this pattern, so it should generally be avoided in systems where cyclic dependency exists among services. If an organization needs to ensure data consistency in a distributed system without tight coupling, the pattern should be adopted. Another scenario is when one needs to compensate if one of the operations in the sequence fails. Lastly, the pattern should be avoided when transactions are tightly coupled since it brings added complexity for no benefit whatsoever. Ideally, when there are less than (and including) three services where data can be lost on rare occasions, the Choreographer is the right approach. Otherwise, the Orchestrator should be used when dealing with four or more services.

4.5 Observability Patterns

This subsection will lay out the patterns that make sure applications are operating reliably. No matter how careful developers are, issues during production are inevitable. The four methods which individually can help a microservice application in terms of reliability are as follows:

- Distributed Tracing
- Health Check API
- Log Aggregation
- Metrics

(*NOTE:* the four patterns are not alternatives of each other, but rather can be complementary).

This section does not contain any tables because each pattern attempts to provide a different aspect to improving a microservice application. Thus, each table would have different metrics for each pattern, unlike the previous patterns. The main focus

for deciding on these patterns was maintainability. Valdivia et al. concluded in her research that the health check, log aggregation, and metrics all have the benefit of improving maintainability in microservice applications [57], although her research lacked information regarding distributed tracing.

4.5.1 Distributed Tracing

Distributed tracing is a commonly used pattern of in-service monitoring. Distributed tracing follows and observes requests in a distributed environment, giving the ability to pinpoint failures and performance issues to fix them quickly. Hence, to reiterate, the problem is understanding the behavior of an application and troubleshooting problems. The pattern offers three main benefits [42]:

1. By providing visibility into changes decreases the overhead necessary for rollbacks and deploys.
2. It supports polyglot development. Due to the agnostic nature of distributed tracing, a single trace can propagate through different clients.
3. It can help with productivity because developers will spend less time troubleshooting and debugging with distributed tracking than without it.

The most prominent disadvantage is that aggregating and storing traces can require significant infrastructure, which is expensive in human capital and monetarily.

4.5.2 Health Check API

Health monitoring is critical to multiple aspects of operating microservices. Essentially, it can allow near real-time information about the state of containers and microservices. The main issue to solve is how should a live service detect that it is unable to handle requests. The solution is implemented as an API endpoint that returns the health status of the microservice. More specifically, they can be configured for three prominent use cases of real-time monitoring scenarios[8]:

1. The API can test the dependencies of an application (external services and databases) to check availability levels and for expected responses
2. The use of memory and other physical server resources can be pinged for a healthy status
3. Health checks can be used by container orchestrators to check an app's status.

While it is easy to see the benefits the pattern offers, it is essential to remember that the health check service instance might fail between health checks, and thus, requests might still be routed to a failed service.

4.5.3 Log Aggregation

Log Aggregation is used when a microservice architecture is in place, and one needs to debug problems that span multiple components. The issue it attempts to solve is how to effectively view and search different log files resulting from individual distributed runtimes in a microservice environment [13]. The solution is to use a

[8]See Microsoft documentation regarding more information on health checks `https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks`

centralized logging service that aggregates logs from individual service instances, thus, allowing users can search and analyze the logs or configure alerts based on certain output logs. According to Rosa et al. [38], log aggregation increases coupling at a service level, although similar to the other observability patterns, it helps developers spend less time troubleshooting and debugging. Unlike the other patterns, logging is the most accessible type of data to generate [54].

4.5.4 Metrics

Metrics are a numerical representation of data, giving developers insights into the past and current state of a system. It offers the ability to perform statistical analysis and predictions on a system's future behavior since it deals with numeric data. The problem which metrics attempt to solve is how to understand the behavior of an application. Using some tools to gather statistics about individual operations becomes possible. When comparing metrics to logs, a significant advantage is that the cost of metrics does not increase concerning user traffic or similar system activity that could result in a significant increase in data [54]. Metrics are also better for triggering alerts because running queries against an in-memory, time-series database is far more efficient, not to mention more reliable. Thus, it contrasts running queries against a distributed system such as Elasticsearch and then aggregating the results before deciding whether or not an alert should be triggered [54].

4.6 Miscellaneous Patterns

While there are many patterns for microservices as described in the following resources (which inspired patterns in this paper):

- Design Patterns for Microservices by Mike Wasson [59]
- Microservices Migration Patterns, by Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, Theo Lynn [10]
- Implementation Patterns for Microservices Architectures, by Kyle Brown and Bobby Woolf [13]
- Microservice Architecture - A Pattern Language for Microservices, by Chris Richardson [48]
- Microservice Patterns and Best Practices, by Vinicius Feitosa Pacheco [39]

We can not describe all of them; hence, the following subsections will describe four patterns used for different use cases in the context of microservices.

4.6.1 Service Discovery Pattern

The two primary forms of discovery patterns are server and client-side discovery patterns. Service discovery patterns are helpful since they help us find the locations of services required to be invoked. While a traditional distributed system meant services ran at a fixed location in terms of hosts and ports, it was easy to call another service using HTTP/REST or some RPC mechanism. Unfortunately, with the rise of virtualized or containerized environments, this becomes a new challenge since the number of service instances and their locations change dynamically [48]. The

solution introduces a service that can be invoked by other services to retrieve critical information about other components. Thus, microservices communicate to the registry to publish their locations, whereas clients address the registry to discover registered services [33].

Server-side Service Discovery Pattern

Taking a look at how server-side discovery would function, from diagram 4.17 before a request is sent to the load balancer, the client makes a request through a load balancer. The load balancer invokes the service registry (a database of services, their instances, and their locations), which forwards the request to an available service instance [31].

Advantages

1. **Less Code:** A benefit in contrast to client-side discovery, the client code is simpler since it does not have to deal with discovery. Instead, a client requests the router

Disadvantages

1. **More Network Calls:** Requires more network calls than using client-side discovery, which is a considerable drawback.
2. **Extra Complexity:** Another drawback if it is not integrated into the cloud environment, the router is another component that must be installed and configured that needs replicated for availability.

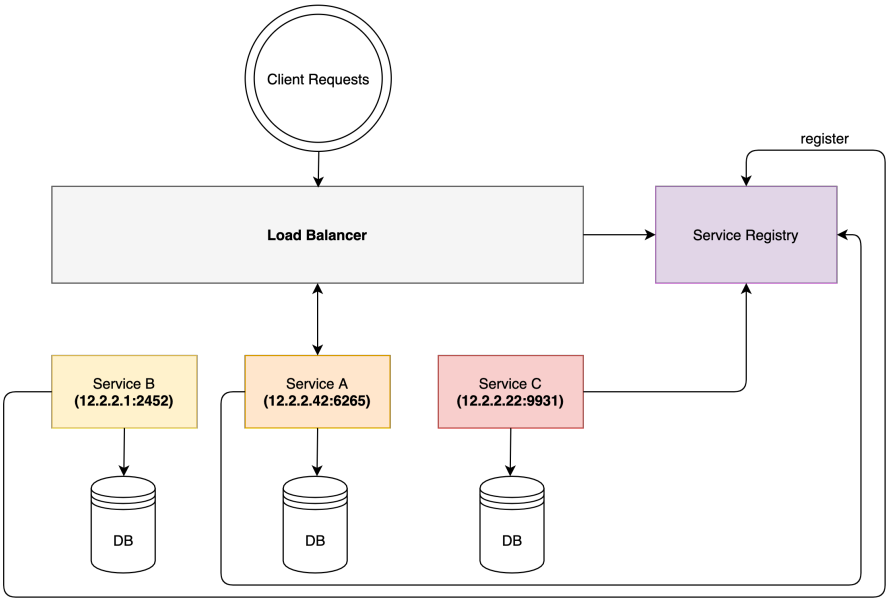

FIGURE 4.17: An overview of the server-side service discovery pattern architecture

Client-side Service Discovery Pattern

The Client-side Service Discovery Pattern functions as follows when a request is created. First, the client obtains the location of a service instance by invoking the service registry and then can communicate directly with one of the service instances, as seen in figure 4.18.

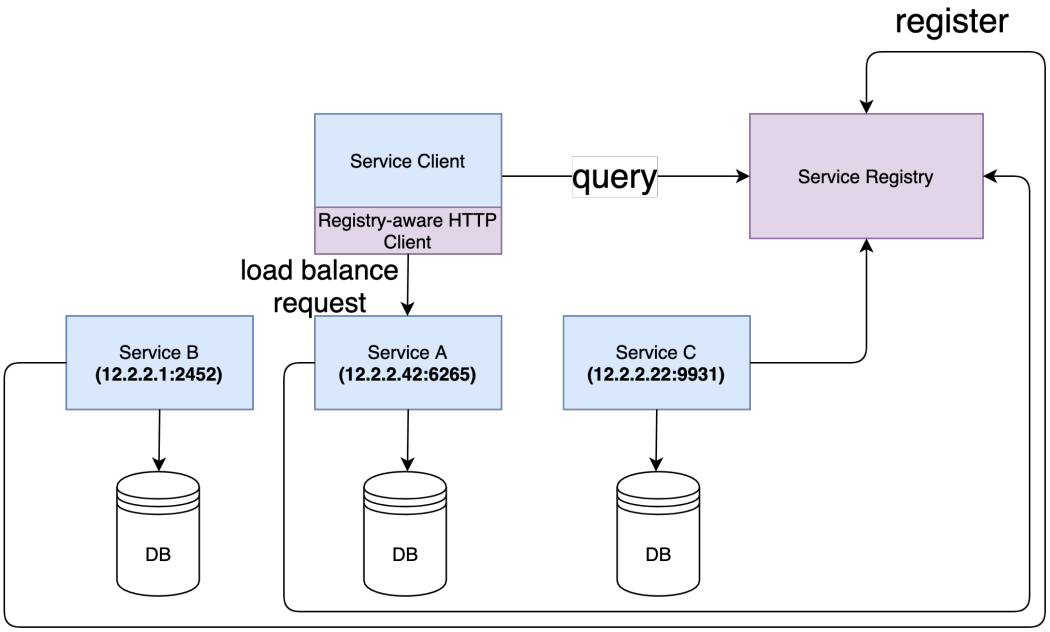

FIGURE 4.18: An overview of the client-side service discovery pattern architecture

Advantages

1. **Lower Latency:** Since the client communicates directly with service registry there are fewer moving parts and network hops compared to server-side Discovery.

Disadvantages

1. **Extra Development:** A significant disadvantage of this pattern is the extra client-side service discovery logic required for each programming language/framework used by the application.

2. **Coupling:** The client becomes tightly coupled with the service registry

4.6.2 Circuit Breaker Pattern

The Circuit Breaker Pattern is a reliability pattern. The term circuit breaker comes from the electrical switch in houses that automatically protects an electrical circuit from damage done by an overload. The pattern becomes relevant because, in an application where one service synchronously invokes another, there is a possibility that another service is unavailable, so requests will continuously be pining that service.

In the meantime, if the services are still pining unavailable services, it could lead to network resource exhaustion, leading to a lower performance and user experience. As a result, the pattern aims to prevent a single component's failure from cascading beyond its limits and bringing the entire system down with it [33].

The solution is that when several consecutive failures cross a threshold, the circuit breaker gets triggered; thus, for the duration of the timeout, all the attempts invoking the service will fail immediately. After which, the circuit breaker will allow a subset of test requests. Conditional on a successful response, the circuit breaker resumes regular operation; otherwise, a timeout period occurs again. The pattern is beneficial since it has the advantage of handling the failure of services it invokes. However, it does bring the challenging disadvantage of selecting timeout values without introducing extra latency or creating false positives [48].

4.6.3 Canary Pattern

The Canary Pattern is a deployment pattern that introduces new updates in microservices in a safe and reliable method. The concept is to release a new deployment to a subset of users. The increments can be set in the target environment and generally are updated in small increments such as 5%, 25%, 50%, 75%, and 100%. This deployment pattern has a lower risk of failure (in contrast to the other deployment patterns [48]). If an application has an app-breaking bug introduced into production after a new release, only a subset of users will be affected due to its incremental nature.

This pattern shines in allowing organizations to test in production with real users and compare the performance of both versions live. It creates an easier way to trigger a rollback and faster since a subset of users exposed to the updated version must be re-routed.

On the other hand, scripting a canary release can be very complex. There also is an increase in required monitoring, and instrumentation for testing in production may involve additional research. From figure 4.19, we can see how requests would be distributed through the load balancer onto the two live versions based on a user ratio threshold.

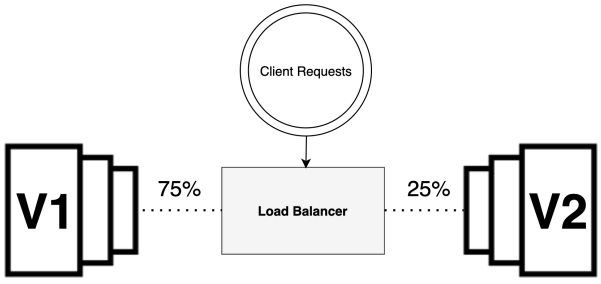

FIGURE 4.19: A simple example of how using the Canary deployment would distribute load

Chapter 5

Case Study: Researchable

This section will attempt to apply some real-life scenario patterns after learning about the different patterns microservices can leverage to create safer, scalable, and robust microservices.

Background

Researchable is a startup in *Groningen, Netherlands* that focuses on developing software solutions for improving scientific research and helping researchers to spend their time on more valuable tasks. While they have many client projects that mainly involve a monolithic architecture, in one of their projects, they have implemented a microservice architecture that creates a case study for this paper.

Context

The project using a microservice-oriented approach is called *Sports Data Valley*[1] (SDV), the largest national data platform for data analysis and sport and physical exercise research. The project was created as a sizeable scalable data platform that can safely manage, analyze and access data through a central infrastructure.

Area of improvement

The section up for improvement in the whole application is the data ingestion service. Essentially, as there are so many data sources to connect and retrieve data from, a service must be present to handle this data for either storing or some extra cleaning of the data (post-processing).

Main Challenge

The main challenge is many different data sources: Garmin, Fitbit, Strava, and similar sources. Each platform has a different interface/ API to pull data for customers. For example, some sources may have the option to export all the data for a specific user, whereas others may require pinging an endpoint several times. While this may sound simple in practice, developers are often faced with more profound issues such as rate-limiting thresholds for APIs. Thus, the final question we shall attempt to answer is:

Should SDV keep its current architecture where there is a single data ingestion service where all the data pulling happens in one service, or should there be a dedicated service for each respective source to pull data?

[1] See the link below more description regarding SDV https://researchable.nl/projects/sportdatavalley

Analysis

Let us start by analyzing the positive aspects of using a single service.

1. **Less microservices:** As this paper has mentioned, having fewer microservices present is not bad. Having fewer microservices leads to

   - Fewer services to deploy
   - Less service discovery issues
   - Fewer resources needed

   All of which are issues that developers and organizations would prefer not to happen.

2. **Less development time:** In this case, since the single data ingestion service is already running, there is less development time spent on splitting the current service into their respective services for each source. For startups such as Researchable, this is quite important since they have clients for other projects.

Now for the advantages of having individual services for each source to retrieve data for SDV.

1. **Easier to pinpoint failure in a service:** Because we have a service running for each respective source, it becomes easier to pinpoint failure. For example, suppose Strava's API is facing downtime, and all the other services acting as connectors are pulling data smoothly with no errors. In that case, it is easy to see which service is not running properly due to its single responsibility nature.

2. **No single point of failure:** Assuming microservices have no downtime would be a naive statement. If Researchable kept having one service to pull data, then a severe bug or memory leak in that services would entail that none of the data sources can pull information as they are all coupled in one service. However, if Researchable kept n different services and, for example, the *Strava* service (which pulls data from a *Strava* API) had a memory leak, the other services such as the *Fitbit* service and *Garmin* service would remain functioning as intended.

3. **Easier to add new connector services in the future:** This advantage is quite opinionated. Nevertheless, suppose each data source has it is on service which is responsible for pulling data. In that case, there is a clear separation between services, which means if there is a new requirement to connect a new source like *Runtastic*. All that is needed is a new service for *Runtastic*. Whereas, if all the code was coupled in a since service adding new methods in possible long files could create more confusion if not a mini monolith.

A new aspect, one might argue, is that each team should only be responsible for a single service. However, there is more detail to this statement. The real intent is that multiple teams should **never** own a microservice. Otherwise, the goal of ensuring rapid and total control over the service would not be present if the service spans multiple teams. Thus, as long as one team owning multiple services does not break the rule above, there should not be any worries. That is, if the team has the capacity and ownership to handle the microservices end-to-end from development, integration, testing, versioning, and deployment, then it should be fine.

Utilizing Patterns

As we have two different ways of approaching the data ingestion problem, let us start by reiterating the useful patterns for both cases and then dive deeper into the patterns useful for individual cases. The *decomposition* patterns are not helpful in this case since the application is also split up using a method suitable for Researchable. Regarding the *observability* patterns, we have four options that possibly go well together, as we have seen. Since we are requesting information on external APIs, the *health check API* is likely implemented from our external data sources. Researchable has already implemented log aggregation, which means they should consider using *distributed tracing* and *metrics*. Distributed tracing is generally relevant when data gathering is done across services and not just when pulling data. However, they can consider adopting it in a different section of the application. Metrics are beneficial in both approaches. Thus, it is recommended that Researchable consider adopting this observability pattern.

When considering the *miscellaneous patterns*, we visited three patterns. The *service discovery pattern* is an essential part of a microservice architecture. It ensures locations of microservices are not fixed when using containers (since they often are spun based on load and other factors). This pattern is not specific to the data ingestion problem but rather the entirety of the application. Likewise, the *circuit breaker pattern* is useful when microservices communicate between 3 or more services. While this is not the case for our current case study, it can and should be considered a different case of the SDV application. Last but not least, the *canary pattern* is applicable for this case study. When Researchable wants to test our new features, they can use this pattern to safely test features and gradually roll them into production, minimizing a significant failure for all customers.

Moving forward to the *integration patterns*, Researchable should consider adopting the *asynchronous messaging pattern*. This pattern helps with creating scalable services, which are also maintainable. For example, after the data is retrieved from a particular data source, we can produce messages onto the messaging queue. The final data processing service can consume the data written in the queue. The key aspect to note is how this pattern ensures the final destination is not coupled to the sources. A specific integration pattern that Researchable should consider if they decide to have a service per source is the *API gateway pattern*. This pattern ideally stands in front of the services encapsulating their behaviors. It can also handle authentication, security, and routing, all of which are useful if there are services for each source. We have already seen the benefits and drawbacks of introducing an API gateway in the previous section. Thus, after careful analysis, Researchable should consider adopting this pattern should they change their data ingestion service structure.

Finally, yet importantly, it is time to consider the *database patterns*. For either case (single service versus multiple services), the *Database per Service* is useful and a good idea. While it does reduce more monitoring in multiple databases, there are many benefits such as scalability, efficiency, and loosely coupling, as previously seen. Maintainability is a crucial aspect of Researchable's decisions on patterns. One possibility if the data ingestion service is split into multiple services is the *shared database pattern* since the maintainability is low since there is only one database to monitor. However, to reiterate, this does couple the microservices through the database. For example, suppose one needs to change any table, relation, as similarly because one

service needs it. In that case, modification and redeployment are required for the other microservices to adapt to the changes. Thus, depending on what needs to be stored in the databases. For example, suppose the only data that needs to be persisted and shared between the services is the token. In that case, *Researchable* could consider using a distributed cache like Redis since it is faster than a traditional relational database.

Conclusion

The honest answer here is that it depends. *Researchable* could maintain their single service with heavily modularized sources, each implementing the same interface as an easier option. However, if they have the resources and the three benefits listed previously sound appealing and the new patterns to implement, they should work on a service per source solution in SDV. A good reason for adoption will be if *Researchable* needs to scale resources of different aspects independently. My opinion of this case is that *Researchable* should move over to have multiple services where each service is only responsible for one source. The most significant advantage is that we are making sure not to couple the different sources. The APIs of the different providers can change, or some of them disappear, or more sources appear, and the rest of the sources will not be affected. No source code changes and no new releases would be needed to fix the issue.

Nevertheless, I would suggest designing a subset of the system, as seen in figure 5.1. The aspect to note here is that the rest of SDV communicates to the API gateway, handling more functions. After the data is pulled from the suitable sources, it is being produced to a messaging queue, and from there, a data processing service consumes the data in the queue. In this (opinionated) recommended approach, each service maintains there own databases to ensure the microservices are loosely coupled.

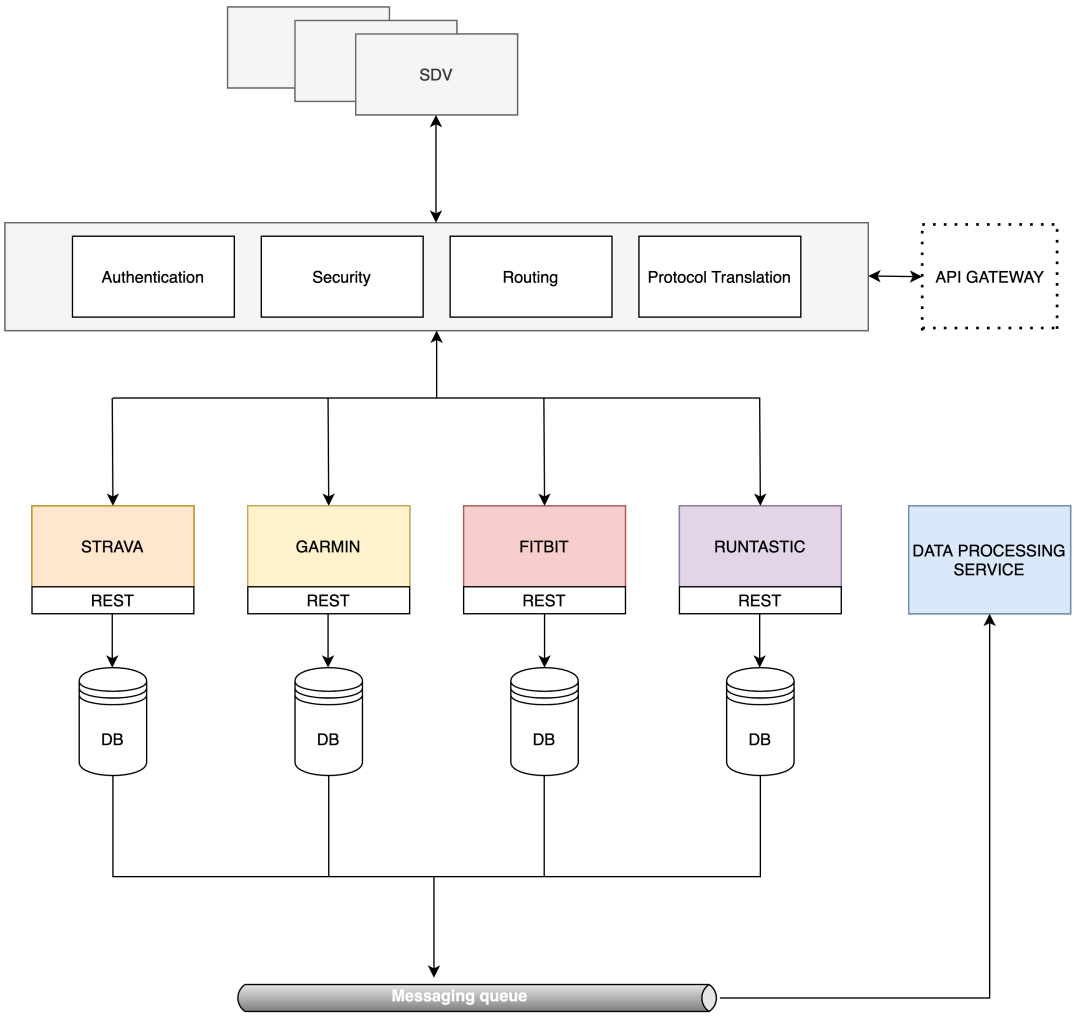

FIGURE 5.1: An architectural overview of SDV with the recommended patterns from this paper (*NOTE:* the fitness sources are just an example, and there could likely be more services integrated)

Chapter 6

Conclusion

This paper started by analyzed different architectures, namely, monolithic, SOA, microservice, and serverless architectures. We stated the advantages and disadvantages for each of the architectures and understood how the components interact with each other. Before diving into the architectural patterns, this paper listed supporting technologies such as load balancers, messaging queues, containers, orchestration, and scaling cube. These technologies laid out the foundation before getting into the technical specifications of the architectural patterns in chapter 3.

The section containing the architectural patterns has by far the most content present. For the first three sub-patterns, they each had a mark for each metric to categorize them.

	Stability	Coupling	Easability of split	Reduction in the number of microservices	Improved Availability	Data consistency
Decompose by Business Capability						
Decompose by Subdomain						
Decompose by Transactions						
Decompose by service per team pattern						

TABLE 6.1: Collection of decomposition patterns with the relevant metrics categorized

Some of the main observations from table 6.1 regarding the decomposition patterns are as follows:

- All the *easability of split* results were red, indicating there is no easy or clear-cut method of moving over to a microservice architecture.
- *Stability* is indicated as green for decomposition by business capability and subdomain.
- The current scope of literature lacks information on the *availability* for all but decomposition by transactions.
- Decomposition by transactions is the most robust pattern when focusing on *reducing the number of microservices*, *increasing availability*, and *data consistency*.
- The least amount of *coupling* occurs in decomposition by subdomain and service per team pattern.

- There is a lack of information on the decompose by service team pattern. Thus, the metric results are likely to fluctuate until more research arises on the pattern.

In this paper, we also categorized the integration pattern (how services interact with each other) in table 6.2. The conclusions were as follows:

- All the patterns are good at *encapsulating* the internals of the services.
- All the patterns are good at scalability in all three axes, with a slight exception of the aggregator pattern scaling in the *y* axis.
- While the API gateway and the chained microservice lack data regarding *reducing complexity to orchestrate data*, the rest of the patterns face a challenge in this metric.
- The two main patterns which promote good *maintainability* are the API gateway and asynchronous messaging patterns.
- *Debugging code* in all but the aggregator pattern is a challenging process.
- The API gateway is the only pattern that certainly has *low latency*.

	Encapsulation	Scalability			Reduced complexity to orchestrate data	Low latency	Debugging code	Maintainability
Aggregator Pattern		x	y	z				
API Gateway Pattern		x	y	z				
Chained Microservice Pattern		x	y	z				
Branch Pattern		x	y	z				
Asynchronous Messaging Pattern		x	y	z				

TABLE 6.2: Collection of integration patterns with the relevant metrics categorized

Finally, the last table 6.3 listed below was concerning the database patterns. Similarly, they were categorized on specific metrics described previously in chapter 3. The conclusion derived from table 6.3 are as follows:

- All the patterns are *efficient* apart from the shared database per service, which depends on the implementation.
- The shared database per service pattern has the *least complexity* issues and monitoring requirements than the rest of the patterns.
- The three most *scalable* patterns are CQRS, event sourcing, and the database per service pattern.
- The three most *maintainable* patterns are CQRS, choreography-based saga, and the shared database per service.
- The four most *loosely coupled* patterns were event sourcing, database per service, and the choreography and orchestration-based saga patterns.

The paper then dove into observability patterns which help us ensure applications are operating reliably. It was comprised of distributed tracing, health check API, log aggregation, and metrics. Each of these patterns attempted to maximize the ability to monitor our microservices.

	Efficient implementation	Reduced complexity	Maintainability	Reduced monitoring	Scalability	Loosely coupled
CQRS						
Event sourcing						
Database per Service						
Shared Database per Service						
Saga (Orchestration)						
Saga (Choreography)						

TABLE 6.3: Collection of database patterns with the relevant metrics categorized

Last but not least, we finished the chapter with some miscellaneous patterns covering a reliability pattern, which was the circuit breaker pattern. In our deployment pattern (canary pattern), we understood how to reliably deploy applications to users to minimize the chance of a large failure. This subsection also explained how microservices should automatically store their location on a service registry to maximize maintainability through the service discovery patterns. Finally, the circuit breaker helps us design microservices in which faults may occur in communication chains and how to deal with similar situations safely.

The following chapter described and attempted to solve a case study for Researchable. In the case study, we analyzed the different perspectives. We then recommended the approach and patterns they should utilize to maximize their goal of maintainability, which was also the research question of this paper. The three main patterns consisted of:

1. API gateway pattern
2. Asynchronous messaging pattern
3. Database per service pattern

It was also mentioned that Researchable should consider adopting at least one observability pattern. To reiterate, the three patterns which focused on improving maintainability was:

1. Health check API
2. Log aggregation
3. Metrics

Finally, when considered the miscellaneous patterns, Researchable should consider their current deployment method and compare it to the canary deployment for the reasons mentioned earlier. The circuit breaker pattern could also be helpful if Researchable has a chain communication in another part of their application; however, there was no such scenario for the scope of the case study.

In conclusion, after deriving the results from the literature to create the tables and results, it was a simple task to apply it to Researchable's case, having known their goals by adopting. Similarly, for a new business, if the goals of implementing a pattern are scalability, it becomes a more manageable task having the results in this paper. Using the metrics and results, the company has to decide its priorities, thus, creating ease in deciding the patterns for consideration.

Chapter 7

Future Work

A question to be answered at this stage is:

What are the following steps to be taken after this research?

Thankfully this is a straightforward answer. One key separator between this paper and the current literature is that this paper ranked patterns based on their characteristics for what they are and are not good at. While this is helpful when companies have the same metrics in mind, there lies a gap when a metric is not listed in one of the tables. Thus, motivation is created for metrics not listed in the tables for this paper. As the more metrics become listed out per pattern, the easier a comparison can be made for an argument of adoption.

Additionally, with an ever-increasing rate of *deployment patterns*, namely six so far from Chris Richardson [48], one's research could categorize metrics for those patterns, similar to this paper. For future analysis, implementing each pattern based on a fixed set of case studies would create a stronger argument for ranking the metrics how they currently stand instead of basing the outcomes on current literature, case studies, and implementations.

List of Figures

List of Tables

Bibliography

[1] M. L. Abbott and M. T. Fisher. *Scalability Rules: 50 Principles for Scaling Web Sites*. 1st. Addison-Wesley Professional, 2011, pp. 25–33. ISBN: 0321753887.

[2] M. L. Abbott and M. T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. 2nd. Addison-Wesley Professional, 2015, p. 351. ISBN: 0134032802.

[3] P. Aditya et al. "Will Serverless Computing Revolutionize NFV?" In: *Proceedings of the IEEE* 107.4 (2019), pp. 667–678. DOI: 10.1109/JPROC.2019.2898101.

[4] M. Ahmadvand and A. Ibrahim. "Requirements Reconciliation for Scalable and Secure Microservice (De)composition". In: Sept. 2016, pp. 68–73. DOI: 10.1109/REW.2016.026.

[5] A. Akbulut and H. G. Perros. "Performance Analysis of Microservice Design Patterns". In: *IEEE Internet Computing* 23.6 (2019), pp. 19–27. DOI: 10.1109/MIC.2019.2951094.

[6] N. Alshuqayran, N. Ali, and R. Evans. "A Systematic Mapping Study in Microservice Architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[7] N. Alshuqayran, N. Ali, and R. Evans. "A Systematic Mapping Study in Microservice Architecture". English. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications*. 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications ; Conference date: 04-11-2016. IEEE, Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[8] N. Ashikhmin, G. Radchenko, and A. Tchernykh. "RAML-Based Mock Service Generator for Microservice Applications Testing". In: Nov. 2017. ISBN: 978-3-319-71254-3. DOI: 10.1007/978-3-319-71255-0_37.

[9] T. Asik and Y. E. Selcuk. "Policy enforcement upon software based on microservice architecture". In: *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*. 2017, pp. 283–287. DOI: 10.1109/SERA.2017.7965739.

[10] A. Balalaie et al. "Microservices migration patterns." English. In: *Software : Practice and Experience* 48.11 (Nov. 2018), pp. 2019–2042. ISSN: 0038-0644. DOI: 10.1002/SPE.2608.

[11] I. Baldini et al. "Serverless computing: Current trends and open problems". In: *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[12] D. Betts et al. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. 2013.

[13] K. Brown and B. Woolf. "Implementation Patterns for Microservices Architectures". In: Monticello, Illinois: The Hillside Group, 2016, pp. 1–35.

[14] T. Černý, M. Donahoo, and J. Pechanec. "Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems". In: Sept. 2017, pp. 228–235. DOI: 10.1145/3129676.3129682.

[15] A. Chun et al. *Accelerating Modernization with Agile Integration*. IBM Redbooks, 2020, p. 18. ISBN: 9780738458366. URL: https://books.google.nl/books?id=DurNDwAAQBAJ.

[16] S. Coleman et al. "Architecture of a Scalable, Secure and Resilient Translation Platform for Multilingual News Media". English. In: *Proceedings of the 1st International Workshop on Language Technology Platforms*. Marseille, France: European Language Resources Association, May 2020, pp. 16–21. ISBN: 979-10-95546-64-1. URL: https://aclanthology.org/2020.iwltp-1.3.

[17] S. Diakov, T. Zubrei, and A. Samoidiuk. "Application of event sourcing and CQRS patterns in distributed systems". In: (2019). DOI: https://doi.org/10.20535/1560-8956.1.2019.178224.

[18] N. Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *CoRR* abs/1606.04036 (2016). arXiv: 1606.04036. URL: http://arxiv.org/abs/1606.04036.

[19] A. Eivy and J. Weinman. "Be Wary of the Economics of "Serverless" Cloud Computing". In: *IEEE Cloud Computing* 4.2 (2017), pp. 6–12. DOI: 10.1109/MCC.2017.32.

[20] E. van Eyk et al. "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 1–4. ISBN: 9781450354349. DOI: 10.1145/3154847.3154848.

[21] M. Fowler and J. Lewis. *Microservices a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 02/14/2021).

[22] "Google Cloud Infrastructure Components Incident 20013". In: (2020). URL: https://status.cloud.google.com/incident/zall/20013 (visited on 02/28/2021).

[23] M. Hamzehloui, S. Sahibuddin, and A. Ashabi. "A Study on the Most Prominent Areas of Research in Microservices". In: *International Journal of Machine Learning and Computing* 9 (Apr. 2019), pp. 242–247. DOI: 10.18178/ijmlc.2019.9.2.793.

[24] S. Haselböck and R. Weinreich. "Decision Guidance Models for Microservice Monitoring". In: IEEE. Apr. 2017, pp. 54–61. DOI: 10.1109/ICSAW.2017.31.

[25] G. Hohpe. "Developing software in a service-oriented world". In: *Datenbanksysteme in Business, Technologie und Web, 11. Fachtagung des GIFachbereichs "Datenbanken und Informationssysteme" (DBIS)*. Ed. by G. Vossen et al. Bonn: Gesellschaft für Informatik e.V., 2005, pp. 476–484.

[26] B. Jambunathan and K. Yoganathan. "Microservice design for container based multi-cloud deployment". In: *Journal of Advanced Research in Dynamical and Control Systems* 2017 (June 2017).

[27] M. Kalske, N. Mäkitalo, and T. Mikkonen. "Challenges When Moving from Monolith to Microservice Architecture". In: Feb. 2018, pp. 32–47. ISBN: 978-3-319-74432-2. DOI: 10.1007/978-3-319-74433-9_3.

[28] E. Kisller. *THE BASICS: The Role of Containers in Your Microservice Architecture*. Apr. 2021. URL: https://jfrog.com/knowledge-base/the-basics-the-role-of-containers-in-your-microservice-architecture/ (visited on 04/26/2021).

[29] G. Lewis et al. "Common Misconceptions about Service-Oriented Architecture". In: *2007 Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07)* (2007), pp. 123–130.

[30] Z. Mahmood. "The promise and limitations of service oriented architecture". In: *International journal of Computers* 1.3 (2007), pp. 74–78.

[31] A. Messina et al. "A Simplified Database Pattern for the Microservice Architecture". In: June 2016. DOI: 10.13140/RG.2.1.3529.3681.

[32] V. Michell. "A focussed approach to business capability". In: *First International Symposium on Business Modelling and Software Design–BMSD*. 2011, pp. 105–113.

[33] F. Montesi and J. Weber. *Circuit Breakers, Discovery, and API Gateways in Microservices*. 2016. arXiv: 1609.05830 [cs.SE].

[34] K. Munonye and P. Martinek. "Evaluation of Data Storage Patterns in Microservices Archicture". In: *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*. IEEE. 2020, pp. 373–380.

[35] D. Namiot and M. sneps-sneppe. "On Micro-services Architecture". In: *International Journal of Open Information Technologies* 2 (Sept. 2014), pp. 24–27.

[36] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. 1st. O'Reilly Media, Feb. 2015, p. 280. ISBN: 978-1491950357.

[37] E. Ntentos et al. "Supporting architectural decision making on data management in microservice architectures". In: *European Conference on Software Architecture*. Springer. 2019, pp. 20–36.

[38] T. de Oliveira Rosa et al. "A Method for Architectural Trade-off Analysis Based on Patterns: Evaluating Microservices Structural Attributes". In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. EuroPLoP '20. Virtual Event, Germany: Association for Computing Machinery, 2020. ISBN: 9781450377690. DOI: 10.1145/3424771.3424809. URL: https://doi-org.proxy-ub.rug.nl/10.1145/3424771.3424809.

[39] V. F. Pacheco. *Microservice Patterns and Best Practices*. Packt Publishing, 2018. ISBN: 1788474031.

[40] C. Pahl and P. Jamshidi. "Microservices: A Systematic Mapping Study". In: Jan. 2016, pp. 137–146. DOI: 10.5220/0005785501370146.

[41] C. Pahl et al. "Cloud Container Technologies: A State-of-the-Art Review". In: *IEEE Transactions on Cloud Computing* PP (May 2017), pp. 1–1. DOI: 10.1109/TCC.2017.2702586.

[42] A. Parker et al. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated, 2020. ISBN: 9781492056638. URL: https://books.google.nl/books?id=fgfIyAEACAAJ.

[43] F. Ponce, G. Márquez, and H. Astudillo. "Migrating from monolithic architecture to microservices: A Rapid Review". In: 2019, pp. 1–7. DOI: 10.1109/SCCC49216.2019.8966423.

[44] V. Power. *Microservices vs. Service Oriented Architecture (SOA)*. July 2020. URL: https://www.sumologic.com/blog/microservices-vs-service-oriented-architecture-soa/ (visited on 03/01/2021).

[45] P. Rajkovic, D. Jankovic, and A. Milenkovic. "Using CQRS Pattern for Improving Performances in Medical Information Systems". In: *BCI*. 2013.

[46] M. Richards. *Microservices Vs. Service-oriented Architecture*. O'Reilly Media, 2015. ISBN: 9781491975657. URL: https://assets.openshift.com/hubfs/pdfs/Microservices_vs_SOA_OpenShift.pdf?hsLang=en-us.

[47] C. Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018, pp. 16–17, 51–55, 121, 261, 399. ISBN: 9781617294549. URL: https://books.google.nl/books?id=UeK1swEACAAJ.

[48] C. Richardson. *A pattern language for microservices*. 2019. URL: https://microservices.io/patterns/ (visited on 02/13/2021).

[49] *Saga distributed transactions - Azure Design Patterns*. URL: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga (visited on 07/17/2021).

[50] A. Sampaio Junior et al. "Supporting Microservice Evolution". In: Sept. 2017. DOI: 10.1109/ICSME.2017.63.

[51] A. Selmadji. "From monolithic architectural style to microservice one : structure-based and task-based approaches". PhD thesis. Oct. 2019.

[52] B. Seo et al. "Implementation of query model of CQRS pattern using weather data". In: *Journal of the Korea Institute of Information and Communication Engineering* 23.6 (2019), pp. 645–651.

[53] A. Singleton. "The Economics of Microservices". In: *IEEE Cloud Computing* 3.05 (Sept. 2016), pp. 16–20. ISSN: 2372-2568. DOI: 10.1109/MCC.2016.109.

[54] C. Sridharan. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media, 2018. ISBN: 9781492033424. URL: https://books.google.nl/books?id=07EswAEACAAJ.

[55] T. D. Stojanovic et al. "Identifying microservices using structured system analysis". In: *2020 24th International Conference on Information Technology (IT)*. 2020, pp. 1–4. DOI: 10.1109/IT48810.2020.9070652.

[56] D. Taibi, V. Lenarduzzi, and C. Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study". In: Mar. 2018. DOI: 10.5220/0006798302210232.

[57] J. A. Valdivia et al. "Patterns Related to Microservice Architecture: a Multivocal Literature Review". In: *Programming and Computer Software* 46.8 (2020), pp. 594–608.

[58] M. Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: Oct. 2015. DOI: 10.1109/ColumbianCC.2015.7333476.

[59] M. Wasson. *Design patterns for microservices*. July 2017. URL: https://azure.microsoft.com/en-in/blog/design-patterns-for-microservices/ (visited on 05/19/2021).

[60] J. Zhao, S. Jing, and L. Jiang. "Management of API Gateway Based on Microservice Architecture". In: *Journal of Physics: Conference Series* 1087 (Sept. 2018). DOI: 10.1088/1742-6596/1087/3/032032.