MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Towards Measuring Releasability Impact of Microservice Patterns"

verfasst von / submitted by

Gabriel Alexandru Kovacs, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 Problem Statement 3

1.3 Solution 4

1.4 Structure of the Thesis 4

2 Fundamentals **5**

2.1 Microservice Architecture 5

2.2 Microservice vs Monolith 7

3 Related Work **9**

4 Background **11**

5 Implementation **14**

5.1 Microservice patterns 15

5.1.1 Database per service 16

5.1.2 Shared Database 16

5.1.3 API Gateway 17

5.1.4 Publish subscribe 17

5.1.5 Message bus 18

5.1.6 Service Orchestration 19

5.2 Architectural decisions 19

5.3 System services 26

5.3.1 Show Stock Reports (SR) 26

5.3.2 Order Products and Receive Ordered Products (OP&ROP) 31

5.3.3 Show Delivery Reports (DR) 44

5.4 Deployment 53

6 Measurement Generation **58**

7 Data Evaluation **60**

8 Conclusion **73**
8.1 Summary 73
8.2 Threads of Validity 75
8.3 Future Work 76

A Zusammenfasung **85**

Abstract

In the last decade, microservice architecture has seen an increased adoption rate among big tech giants (e.g. amazon, Netflix) as well as smaller companies that want to take advantage of the promised benefits. While much work has been done to describe the architecture and its patterns there is still little to no work on how to successfully manage the transition from a monolith to a microservices architecture. This is especially challenging for small and medium enterprises that don't possess the seemingly endless resources of large businesses. This work describes a roadmap that enables a team to acquire and expand its know-how, aid the team in making the best architectural decision in accordance with their needs, and continuously evaluating the implemented system. The proposed solution is low cost, it yields results after every step and lastly, it creates a working environment that promotes change and architectural evolution.

1 Introduction

1.1 Motivation

Just as [10] noticed, all most all successful companies started with a monolith. Starting with a monolith implementation is the easiest approach that enables a team to ship a product as fast as possible and with a rather reduced risk of failure. In time most of the monoliths become a big ball of mud. This usually happens because there was no ongoing documented evolution of the architecture or if there was one it was not enforced. The project becomes too big to grasp, dependencies between the different modules become convoluted, the fear of changing something and breaking the systems inhibits any refactoring of the existing codebase. Thus in an ever faster passed competitive market companies have to be able to adapt and scale their services continuously if they hope to remain relevant.

1.2 Problem Statement

Thus the challenge is how to make the transition from an unstructured monolith to a microservice architecture. The main points that have to be addressed are data management as well as how to ensure communication between the different services. It can be daunting to start with the decomposition of the monolith if one has

no previous experience or any guideline which can provide a starting point. Hence two research questions can be distilled from the previous statements:

- **RQ1** How should we structure and organize the transition from a monolith to a microservice architecture without to much emerging resistance from the involved stakeholders?
- **RQ2** Which microservice patterns to choose based on the existing requirements of the underlying system that is being built?

1.3 Solution

Our approach to addressing these challenges is to provide a blueprint, guideline which can empower a developer to take the first steps towards refactoring the code to achieve the desired outcome. In [20] the authors provide different levels of modularity in which a software system can be. Just as Kent Beck [4] mentioned before making a change it has to be made easy. Thus to be able to transition from a monolith to microservices the software has to have reached at least a modularity of level 2 (i.e. the modular monolith). Before making the transition one has to decide which microservice patterns should be taken into consideration based on the systems requirements. Thus the proposed solution is to develop a statistical model having as variables all the requirements and the patterns that come into question. The desired outcome is to determine if a specific implementation performs as expected, what are the differences between the distinct solutions, and what could be the drawbacks of a specific solution.

1.4 Structure of the Thesis

The work is structured as follows: Section 2 provides information's regarding the differences between the different architectural styles; Section 3 presents the existing related work; Section 4 provides an introduction to the The Common Component Modeling Example (CoCoMe); Section 5 describes the different microservice patterns,

the implementations, and the reason behind the decisions; Section 6 details how the measurements have been performed; Section 7 specifies how the data has been evaluated and presents the resulting findings, lastly Section 8 contains the summary of the work and the future work.

2 Fundamentals

2.1 Microservice Architecture

In the past, the teams involved the development and the deployment of the application were rather isolated from each other. As a testament for this stands the waterfall model which has been prevalent for many years as the de facto standard for software development. With the advent of the agile movement[1] another movement emerged namely DevOps [2], [28]. DevOps as the name already implies addresses the developer and the operations teams which are encouraged to work together on an application from its inception and through its entire life cycle. This approach leads to the need for continuous development and deployment which in turn lead to a plethora of new tools like: Jenkins[1] , Prometheus[2] , Docker[3] and Kubernetes[4]. Another evolution also took place within the architectural styles which started with the monolith and continued with Service-Oriented Architecture [13], [6] (SOA) followed by the Microservice Architecture [37], [25]. Thus many companies (e.g., Amazon, Netflix, Ebay) started adopting the microservice architectural style and move away from their previous monolithic and SOA oriented architectures [19], [14], [18]. Some of the benefits provided by microservices which are properly implemented are:

1. They are easier to reason about because they are build to do one thing and do it well. This entails that prerequisites like high cohesion and the Single Responsibility Principle [26] are being met.

[1] `https://www.jenkins.io/`, accessed 19.01.2021
[2] `https://prometheus.io/docs/introduction/overview/`, accessed 19.01.2021
[3] `https://www.docker.com/`, accessed 19.01.2021
[4] `https://kubernetes.io/`, accessed 19.01.2021

2. The development and reliability cycle is shortened due to (1).
3. The development teams are not constrained by a single technology stack (e.g., using only relational database or a unique programming language).
4. The resilience and fault tolerance of the entire systems increases. The failure of a service will not lead to cascading errors.
5. Because the services are independent and self-contained units they can be scaled up or down based on the existing needs thus eliminating the need for over-provisioning hence reducing the operations costs.

While microservices come with multiple advantages there are also some challenges that need to be considered before committing to this type of architecture. Several of these drawbacks are:

1. The operations complexity increases considerably because there is not just one big service to deploy but dozens or even hundreds of small ones that have to be managed.
2. It requires the team to change and adapt their way of thinking especially due to the change from a monolith to a distributed system. The novelty factors are: the communication between the services (e.g., synchronous vs. asynchronous), testing and debugging the system, and moving away from a data model driven by ACID towards BASE [5].
3. Extracting new services from an existing monolithic codebase and determining the size and the boundaries of the new microservices is not as straightforward as one would hope. The acquisition of new know-how is required. Some of the new paradigms that could help are: Domain Driven Design [7] which provides a means of identifying the existing Bounded Contexts[5] within the existing application; Responsibility Driven

[5]Entails splitting a business domain into smaller independent subdomains (e.g. shipping, payment)

Design [8] that strives to identify the occurring interactions and responsibilities thus resulting in more cohesive and loosely coupled services; Event Driven Design [38] is mainly focused on the events triggered within an application and their consequences. It usually entails asynchronous communication between the services which facilitates local and temporal decoupling.

Just as previously the Gang of Four compiled a series of Design Patterns [11] which covered recurring problems and their proposed solutions Richardson has compiled a series of Design Patterns [32], [35] geared towards solving the challenges which emerge by adopting a microservice architecture. For example, the Command Query Responsibility Segregation (CQRS) has been devised to accommodate database queries in environments where there is not just one central database but rather multiple standalone databases. The SAGA pattern has been conceived to facilitate transactions that span over multiple services offering the possibility to roll back the previously executed actions if the transaction should fail at any given step. Multiple approaches are available to enable service discovery so that the services can communicate with each other. Microservice architecture has been in use for some years now and more likely than not most of the challenges have been identified and one can find a pattern, best practice, or even software that can aid a team in solving their problem.

2.2 Microservice vs Monolith

The monolithic architecture is usually the go to model when starting with a new application. It is a single indivisible unit that contains the user interface, the business logic as well as the data interface. Usually, a monolith is comprised of only one big codebase that lacks any modularity [12]. It can be considered the opposite of the microservice architecture. This does not mean that this style of architecture does not have its merits. The important thing is to be aware of its limitations and recognize when continuing with a monolith becomes detrimental and a hazard to the system it implements. Thus

a microservice architecture can be seen a the result of a matured monolith.

The advantages of a monolith are prevalent in the beginning stages of a new project. These are:

- Reduced overhead when starting out. There is no need for detailed planning.
- The developer team does not need special know-how. Every developer probably worked at some time on a monolithic code-base.
- New iterations containing new features can be implemented and delivered relatively fast.
- Deployment is easy because there is only one unit that has to be deployed.
- Having all the code base together makes testing easier.

All the above advantages hold as long as the code base is small and transparent. After a certain point, all the benefits become draw-backs. At this point, the microservice architecture starts to shine.

- The bigger the monolith the harder it gets to reason about it and no one person is able to grasp the bigger picture. This usually goes hand in hand with missing or stale documentation thus making it harder for new developers to understand what the system does and how it operates. In contrast, a microservice is much smaller, has a predefined structure and function thus making it easier to comprehend and to make it comprehensive to new team members.
- While shipping new features fast a considerable amount of technical debt [27] is being amassed. The higher the debt the harder it becomes to modify the existing code base due to the higher existing complexity, coupling and dependencies that are scattered true ought the system. At this point, the needed overhead to define the structure of a microservice starts to pay off and makes changing the system much easier and faster.

- While the deployment of a monolith is easier it is also much slower than the deployment of microservices which due to their size can boot up much faster. Furthermore scaling up a monolith implies redeploying the entire system multiple times even if just a subset of the functionality is needed whereas in comparison only the microservices implementing the functionality need to be redeployed. The longer deployment times can also impact development negatively especially in the situation where an entire workflow needs to be tested.

- Withing a monolith there is a big commitment to a specific stack which usually also implies a vendor lock-in. This can become a problem if part of the support for the software becomes discontinued, the software does not evolve fast enough, or the licensing model changes. The flexibility of choosing the right tool for the right job is one of the microservices selling point allowing the development team to change course much faster without having to waste too much time and resources.

The microservice architecture has a steeper learning curve, it requires much more upfront investment than the monolithic architecture but if done properly it yields a much higher reward further along the way and in some cases, it remains the only viable option if scaling and new features are ubiquitous to the application.

3 Related Work

Taibi et al. [39] provide a microservice pattern catalog based on the review of more than 2700 scientific papers. The catalog is split into three main pattern subsections: orchestration and coordination, physical deployment strategies, and lastly data management specifically data storage options. For every pattern, the authors offer a description, advantages, and disadvantages. The authors also discuss the guiding principles which drive the development of microservices as a whole with their advantages and drawbacks.

In their paper, Hassan et al. [15] define an architecture-centric

modeling concept for microservices. The purpose of the modeling approach is to determine the granularity and boundaries of a microservice. They show that their proposed solution for modeling and decomposing microservices can facilitate analysis, evaluation, and mobility/location awareness of the underlying architecture, thus allowing the user to be aware of the QoS trade-offs.

The work of Osses et al. [30] describes a systematic review of microservices patterns and tactics (design decisions) that can be encountered in academia as well as industry. They were able to identify five main quality attributes: scalability, flexibility, testability, performance, and elasticity. Finally, the previously acquired knowledge has been cast in a microservice architecture patterns taxonomy which is meant to aid developers in the pursuit of finding the appropriate solution to their problems based on a given context.

Baresi et al. [3] developed a process that can automatically decompose an existing application into microservices based on an OpenAPI (formerly known as Swagger) description of the service interfaces and a vocabulary based on Schema.org . In their experiments, they were able to achieve an 80% accuracy whit regard to microservice decomposition using rather simple applications (containing 10 respectively 13 microservices).

The work conducted by Zdun et al. [40] strived to define a set of constraints and metric which facilitate the automated conformance evaluation of a model to the existing microservice patterns. The constraints and scoring mechanism which became an integral part of the metric were developed in accordance with the existing best practices associated with microservices and with the intention of reducing possible bias towards a specific solution. The proposed solution was implemented with the help of the Frag Modeling Framework (FMF). The approach was evaluated based on 13 different application implementations and cross-checked against the evaluation of domain experts. Although the result didn't fully overlap they were still promising.

In their work Levcovitz et al. [24] propose an approach for extracting microservices from a monolithic architecture by construct-

ing a dependency graph between the component (e.g., database tables, domain logic, clients) manually thus identifying possible microservices and their coupling.

Richardson's book [32] provides a comprehensive overview regarding the microservice architecture style. The author presents the benefits and drawbacks of microservices, decomposition strategies, describes many of the used patterns as well as covering testing and deployment. The work presented in this chapter will be part of the foundation for developing the proposed solution. The previous work is focused on microservice patterns categorization and on the manual or semiautomatic determination of service boundaries. While providing valuable insights it focuses only on a specific issue of the development process and not on a holistic view of the development cycle. Thus going beyond the academic definition of patterns and compliance of the code could aid the technical lead to confirm that the choices made are the right ones.

4 Background

To be able to determine if there are any satisfactory answers for our research questions an application had to be found that is big enough and that can be split into microservices. These requirements were met by The Common Component Modelling Example (CoCoME) [16]. The CoCoME describes a Trading System that resembles a supermarket in which different goods are sold. Every store has multiple Cash Desks that are grouped together into Cash Desk Lines. Every Cash Desk Line is connected to the Store Server that can be accessed via the Store Client. The Store Clint enables the store manager to view reports, order products, or change the price of the product available within the store inventory. Multiple Stores can be grouped together into an Enterprise. Every Enterprise has an Enterprise Server to which all its Stores are connected thus allowing the Enterprise Manager to generate different kinds of reports via the Enterprise Client.

Figure 1 shows the use case diagram of the existing use cases

that are defined within the CoCoME. The use cases are described as follows:

- UC 1:Process Sale - At the Cash Desk the Cashier scans the products a Customer wants to buy and after the scanning is finished the Customer has the possibility to pay either by card or cash.
- UC 2:Manage Express Checkout - If certain conditions are fulfilled a Cash Desk automatically switches into express mode. As a result, only a maximum of 8 items are allowed per Customer and only cash payment is possible. The Cashier is able to switch back to normal mode at any time.
- UC 3:Order Products - The Store Manager is able to view the current status of his inventory. By choosing a product and the amount to be ordered the Manager can place an Order for which an identifier is generated and shown to the Manager.
- UC 4:Receive Ordered Products - The ordered products arrive at the Store at which point the Stock Manager checks the delivery for completeness and correctness. If the delivery is accepted the Manager introduces the previously generated identifier in the system upon which the inventory of the store is updated.
- UC 5:Show Stock Reports - The Store Manager is able to create and view a report that includes all the available stock items in his Store.
- UC 6:Show Delivery Reports - The Enterprise Manager can generate and view a report that contains the mean delivery times for all the suppliers that supply a specific Enterprise.
- UC 7: Change Price - The Store Manager can view every product in his inventory and change its selling price as he wishes.
- UC 8: Product Exchange - If an item's stock goes below a certain threshold the store's server can inform the enterprise

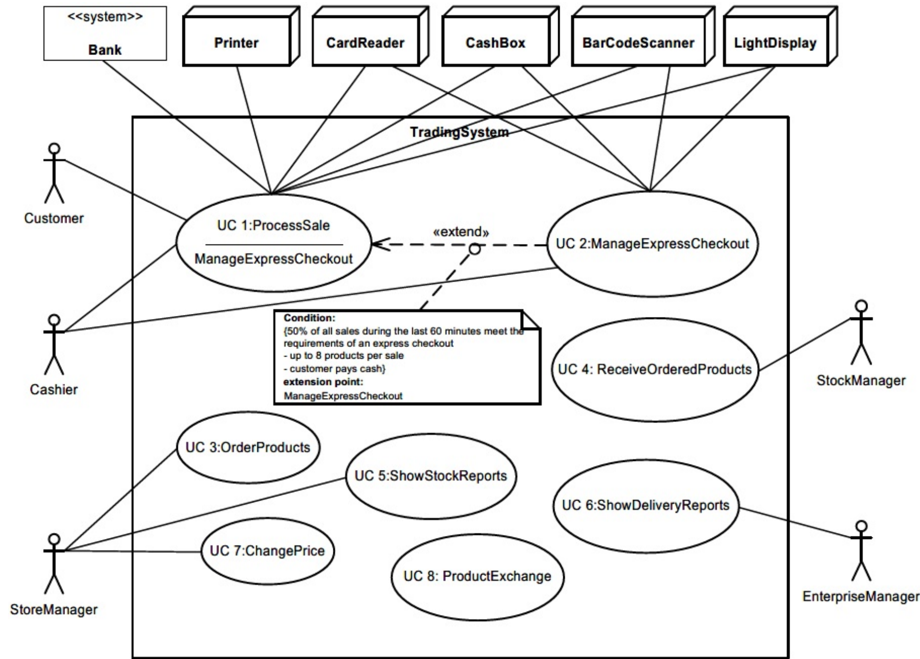

Figure 1: Use Cases CoCoME [16], Fig.4. An overview of all considered use cases of the Trading System.

server about the shortage. Upon receiving the request the enterprise server can query the stock of the item in other shops and if deemed appropriate it initiates a transfer of items from one store to another store.

Another very important and useful information is provided in Figure 2 that contains the data model for the entire Trading System. Thus the structure of the database can be sketched based on the data model. Furthermore, we can get a better understanding of the relationships, interactions, and information exchange between the different use cases.

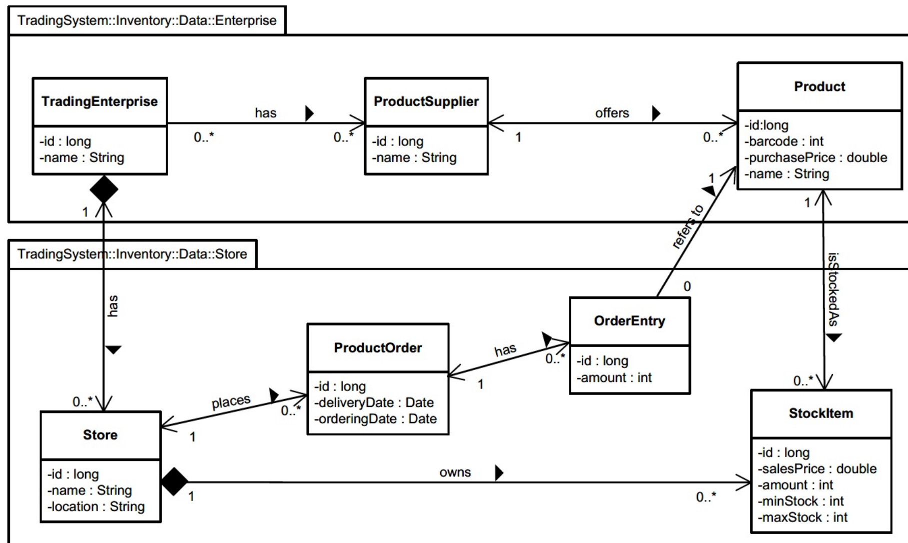

Figure 2: Data Model CoCoME [16], Fig.8. The data model of the TradingSystem.

5 Implementation

Based on the use cases described in the CoCoME multiple use cases have been implemented as a monolith [22]. Knowing that the intention is to transition to microservices the application has been implemented having level 2 of modularity in mind. Thus every use case is considered a new feature that is layered vertically having an API, an application- and a persistence-layer where all the use cases share one common database. This approach allowed us to have a first running proof of concept as well as enabled us to get a better understanding of the interactions between the services. Thus it was possible to start with the first consideration with regards to a possible decomposition of the monolith into multiple microservices. Another advantage of starting with a monolith is that the new implementation can be compared against the old consolidated version thus possible errors regarding the behavior of the application can be caught and handled accordingly.

Five of the existing eight use cases described in the CoCoME

have been implemented. These are:

- UC 3 - Order Products
- UC 4 - Receive Ordered Products
- UC 5 - Show Stock Reports
- UC 6 - Show Delivery Reports
- UC 7 - Change Price

The use cases 1 and 2 have been omitted because they foresee multiple smaller interactions with devices like printers, light display, and so on without having a real impact on the existing data model. Use case 8 while interesting it was not implemented because it necessitates multiple stores that are online at the same time and furthermore it was not clear how to implement the heuristic described in the use case.

After multiple deliberations, three microservices have emerged that are able to fulfill the functionally of the five implemented use cases [21]. These three services implement the use cases as follows:

- Microservice SR - implements UC5 and UC7
- Microservice OP&ROP - implements UC3 and UC4
- Microservice DR - implements UC6

These three services represent only the bare bone implementation of the use cases. To facilitate the information exchange between the services themselves and the outer world represented by the clients multiple microservice patterns have been implemented which will be described in the next Section.

5.1 Microservice patterns

This section contains a description of all the patterns that have been implemented. They can be divided into three main groups: data management (see 5.1.1 and 5.1.2), service communication (see 5.1.4 and 5.1.5), and workflow management (see 5.1.3 and 5.1.6).

5.1.1 Database per service

In the database per service approach, every microservice has its own database [34]. This does not mean that every service has to have its own instance of a database but rather that every service owns its own table schema. This approach allows the developer team to make changes to the schema as they see fit without having to coordinate with other teams thus allowing them to work independently in a non-blocking manner. Another advantage is the fact that the entire stack can be deployed on its own and can be scaled up or down as needed. Every service can use the database that is best suited for the use case it implements from various relational or NoSql databases. Some of the disadvantages are: having to duplicate data to ensure the unambiguousness of the data (e.g. duplicating the primary key), joining data over multiple databases for querying becomes harder, defining transactions that span multiple services become much more complex (e.g. Saga pattern) and lastly, the data becomes eventual consistent having to be synchronized over multiple databases.

5.1.2 Shared Database

Just as the name implies this pattern enforces a common database for all services [36]. Thus different services operate on the same tables. In a microservice architecture, this could be considered an antipattern. While counter-intuitive it comes with some advantages like: the service can directly access the database thus bypassing a potential need to use the API of a foreign service; the developer can use SQL at its full potential (e.g. complex joins over multiple tables); making use of ACID transactions which leads to better data consistency. Of course, there are also some drawbacks that have to be taken into consideration when choosing this approach like: a loss of flexibility within the development cycle due to common database schema which has the be agreed upon by all development teams; the type of the database is fixed (e.g. relational or NoSql); long-running transactions could block the other services; the database could get overwhelmed by the concurrent connections when scaling

up the number of services.

5.1.3 API Gateway

Consists of a component that is placed between the client and the provided services [33]. It acts as a single point of entry for the entire system. This means that every incoming request has to traverse the gateway. Using an API Gateway has multiple benefits like: it reduces the potential attack surface because the components that have to be hardened is reduced to only a couple instead of dozens or hundreds of services; can reduce the complexity within the microservice with regards to authentication and authorization logic; decouples the client from the provided services thus the client has to be aware only of the gateway and not all services; gateways can be specialized meaning that they can provide an API for desktops or mobile devices, consequently streamlining the development process for all involved stakeholders. The shortcomings of this pattern are as follows: any failure of the gateway renders the system unusable for any client thus making it a single point of failure; increases complexity by adding a new component that has to be developed and maintained; can lead to increased latency which depending on the use case could have a negative impact on the end-user experience.

5.1.4 Publish subscribe

Publish subscribe is a messaging pattern that facilitates asynchronous communication between multiple services within a private network or over the internet [17], [29].The main components of the pattern are the publishers, subscribers, and a message broker. The publisher is responsible for the creation of messages that are associated with one or more topics (e.g. incoming orders, delivery). The subscribers are the consumers of the messages which they can receive by registering, subscribing to the topics they are interested in. Lastly, the message broker is the middle man that enables the communication between the different actors that create and consume messages. The broker is the one managing all the topics, the sub-

scribers and the publishers, and lastly the message boxes (queues) associated with the different topics and subscribers. Some of the benefits that come with the usage of the pattern are: enables the asynchronous exchange of information meaning that the involved parties must not be available/online at the same time; it decouples the actors and facilitates dynamic scaling. The publisher for example does not have to know the address of 100 services but only the address of the broker, the same also holds for the consumers. The disadvantages of the pattern are: the broker is another component that has to be managed and can become a single point of failure; reasoning about a synchronous workflow is much easier than reasoning about an asynchronous one; debugging such a system becomes challenging.

5.1.5 Message bus

Message bus is another pattern geared towards enabling information exchange between individual services as described in [17]. The pattern is similar to the publish-subscribe pattern by having a message broker as well as creators and consumers of messages. The main difference is that there are not multiple topics but only one common topic. Thus every actor that wishes to participate has to register for one topic/channel. Furthermore, every participant has to comply with a predefined data model and a command set. Every message can be a command or a response (e.g. query result). Whenever a message is published it gets sent to all the subscribers of the channel. The services themselves have to decide if they should process the message or ignore it. This pattern facilitates asynchronous communication and decoupling of services. Another benefit is the ease with which services can join the bus. For example, one could attach a service that is responsible wit logging thus recording all the communication available in the system.

5.1.6 Service Orchestration

Service orchestration is a pattern that has to do with how microservices work together to fulfill a specific use case[31] . The central role is played by the orchestrator who is responsible for the execution of a series of predefined tasks in a specific order to fulfill a specific goal. Every task can correspond to a service offered by different microservices. For example, as noted [9] with the help of the Business Process Model and Notation (BPMN) a workflow can be defined that contains a series of tasks that can be executed in parallel or sequential until an end state has been reached. The orchestrator is the one responsible for executing the workflow until the final state is reached. The workflow could also be seen as a pipeline where the orchestrator is the one calling a service with a specific input processing the output and then calling the next service in the pipeline. Thus the orchestrator becomes a smart pipe. The benefits of orchestration are: it enables an easy understanding of the workflow it implements; it is easy to maintain and modify. The disadvantages are: the orchestrator becomes a single point of failure; due to the sequential nature of the workflow if one process fails the entire workflow fails; the orchestrator must know how to access all the microservices involved in the workflow; increased run times because all traffic has to run from and back to the orchestrator.

5.2 Architectural decisions

The application described in the CoCoME revolves around the management of the supermarket inventory. It is important to be able to manage the state of the inventory at any given time. Thus the data describing the state of the inventory can be considered one of the most valuable assets of the supermarket. Every one of the use cases retrieves and/or modifies the state of the inventory. Given the nature of the application, the decomposition was data-driven, meaning that the first step consists of breaking the database up. The division is not arbitrary but rather geared towards achieving high cohesion between the distinct services as well as low coupling.

Figure 3 shows one of the basic implementations in which every service has its own database and the communication between the service is made via REST calls. In this example, the original database has already been portioned in such a way that the needed information exchange between the services is reduced to a minimum.

Figure 4 shows a variation in which all the services share the same database and the same tables. The structures of the tables is the same as the one used in the monolith implementation. Thus the communication between the services is facilitated via the database. Therefore the REST calls have been removed. To avoid duplication of the database access code, it has been extracted in the *common* jar which is used by all services as a dependency.

Figure 5 represents an expansion of implementation seen in figure 3 by a new microservice that acts as an api-gateway (AG). The AG acts as a proxy between the client and the other services by forwarding the incoming client request to the appropriate services. This extension of the architecture is an organic one. The advantages provided by this pattern, as already described in Section 5.1.3, exceed the drawbacks.

The implementation shown in Figure 6 is similar to the one in Figure 5, the difference being the replacement of the one database per service with one shared database.

The direction taken in Figure 7 is geared towards asynchronous communication. Whenever the client performs a request he also has to provide a callback address to which the result of the request will be POST-ed upon completion by the AG. After receiving the request from the client the AG responds with a 201 Accepted and saves the callback address together with a correlation id in its database (agDB). The communication between the AG and the other services is point-to-point and is being facilitated by a message broker via queues. The communication between the remaining services is enabled via REST calls and every service possesses its own database.

The setup in Figure 8 is analogous to the one in Figure 7 the difference being that the services share a common database. The solution presented in Figure 9 corresponds to the one in 7 with

the exception of the communication among the back-end services, which is made via topics (command and response) over the existing message broker, thus replacing the previous REST calls.

The solution presented in Figure 10 increases the responsibility of the AG. The AG becomes an orchestrator thus making it responsible for the composition of the data needed to be returned as a response to the client's requests. As a consequence, the service does not need to communicate whit each other. The only existent communication channels are between the AG and the services via REST calls.

The last implementation presented in Figure 11 is related to the one in Figure 10. The difference consists in the means of communication between the AG and the other services which are done via queues over a message broker instead of the previously used REST calls.

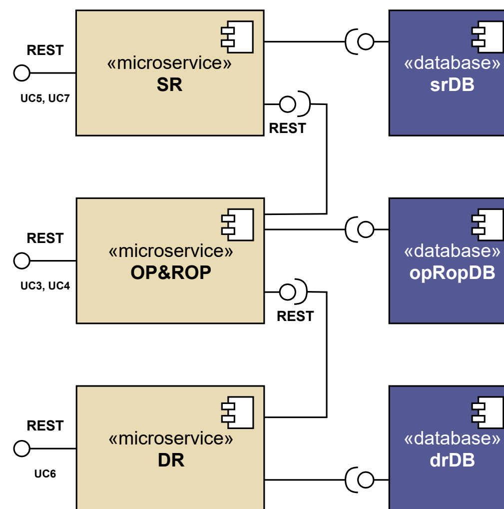

Figure 3: Master with one DB per service

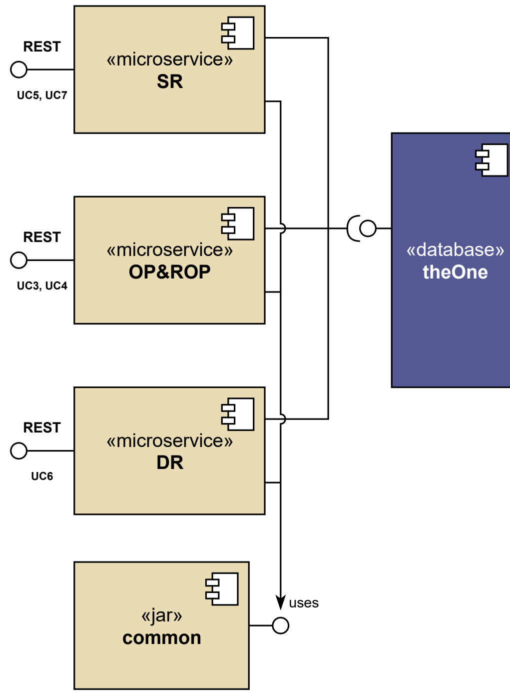

Figure 4: Master with one common DB

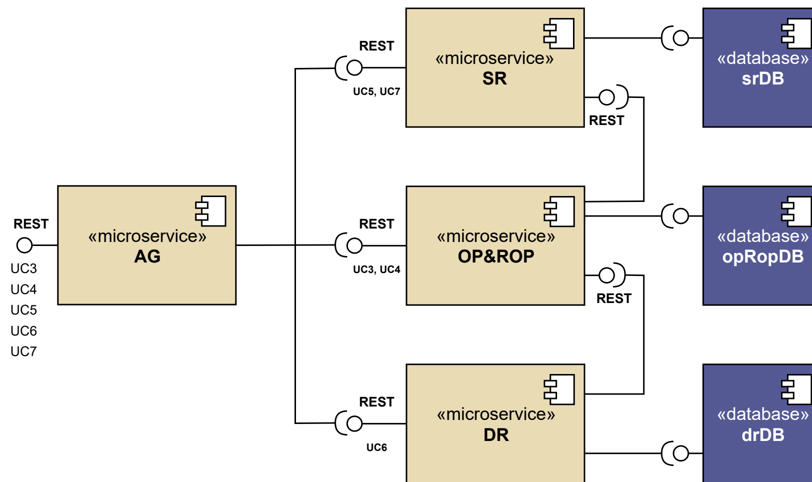

Figure 5: Api-Gateway with one DB per service

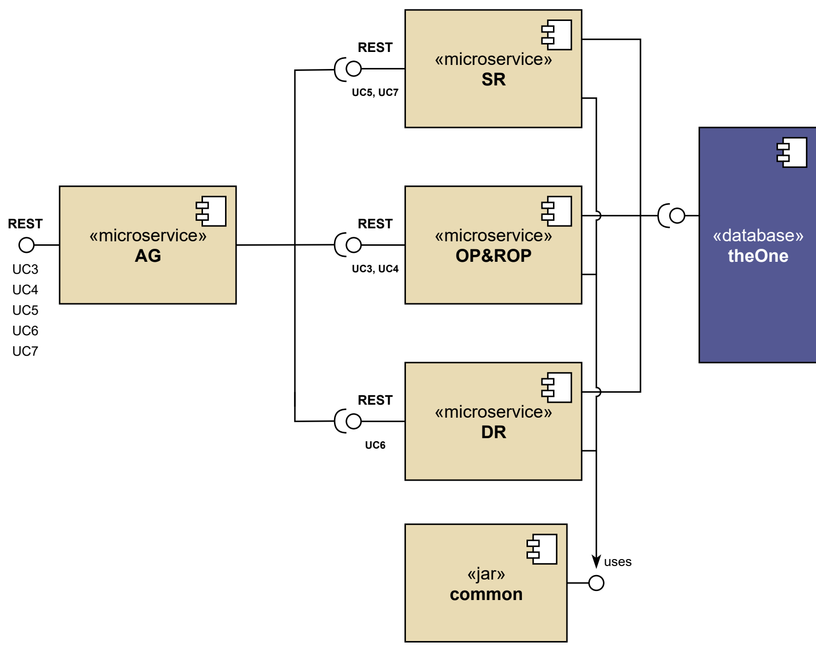

Figure 6: Api-Gateway with one common DB

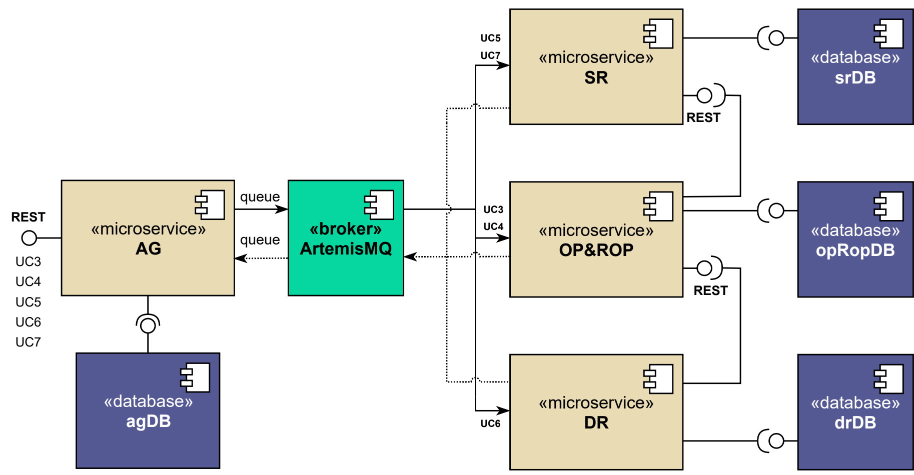

Figure 7: Pub-Sub with one DB per service

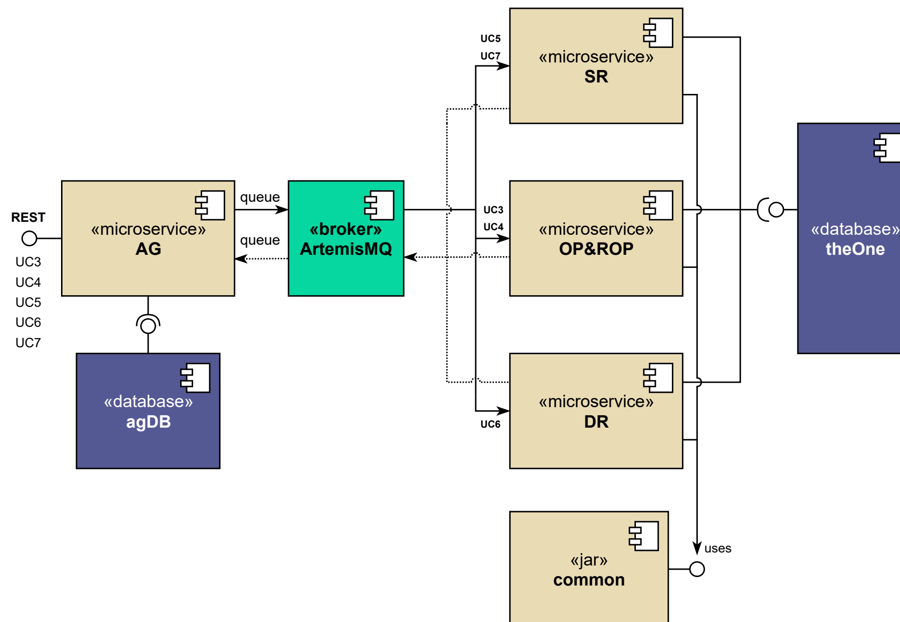

Figure 8: Pub-Sub with one common DB

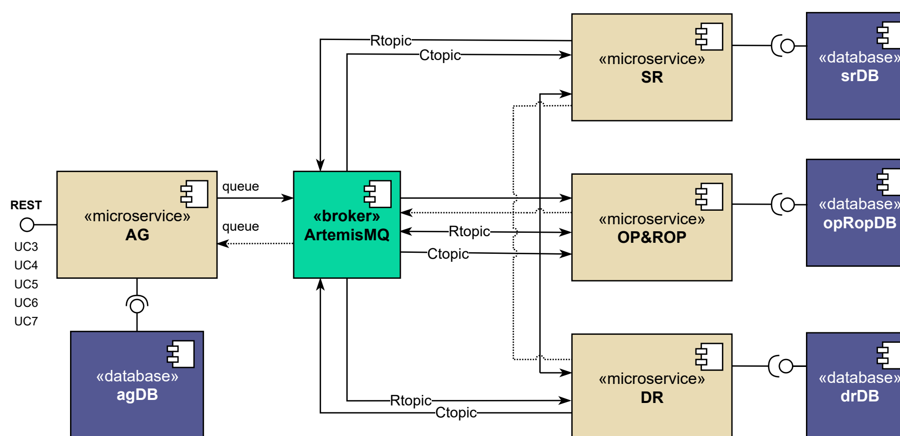

Figure 9: Message bus with one DB per service

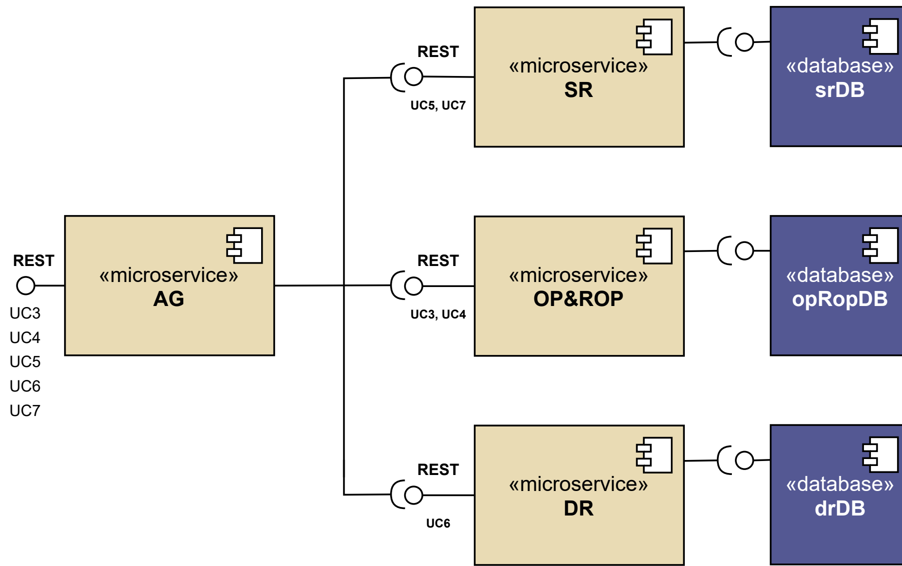

Figure 10: Api-Gateway orchestration with one DB per service

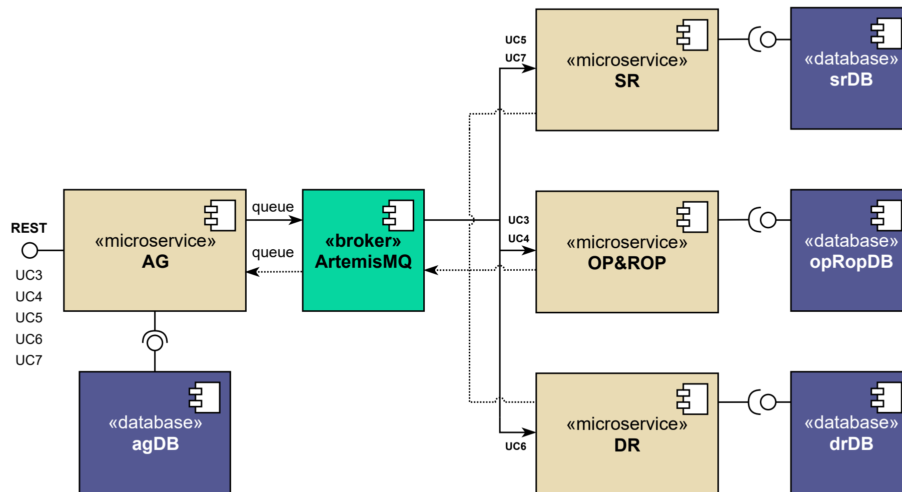

Figure 11: Pub-Sub orchestration with one DB per service

5.3 System services

5.3.1 Show Stock Reports (SR)

The SR microservice is responsible for the realization of UC 5 and UC 7. The service is self-contained meaning that it does not need other services to fulfill any incoming requests associated with the two use cases. The service contains all the needed business logic and also has direct access to the database that accommodates the data.

The data is comprised of a table that contains the store id, the product id, the product's price, the available amount in the store as well as the minimum- and the maximum stock for a given product. Thus the main table is the StockItem table from Figure 2 with additional references to the ids contained in the Store and Product table.

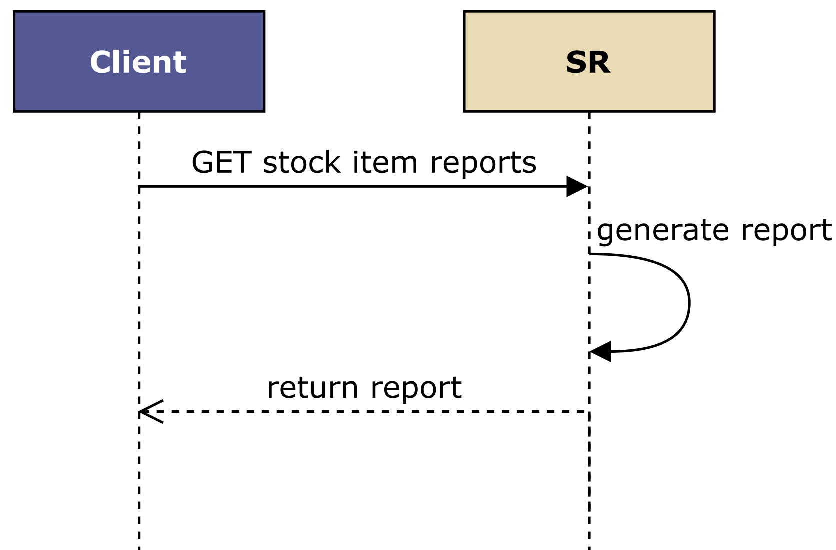

Figure 12: UC 5 sequence diagram for version v01 and v02

The sequence diagram shown in Figure 12 describes the workflow for acquiring the stock item report in v01 (master-1-to-1-db) and v02 (master-one-db). The workflow for v01 and v02 are identical because the SR service is not dependent on the data managed by another

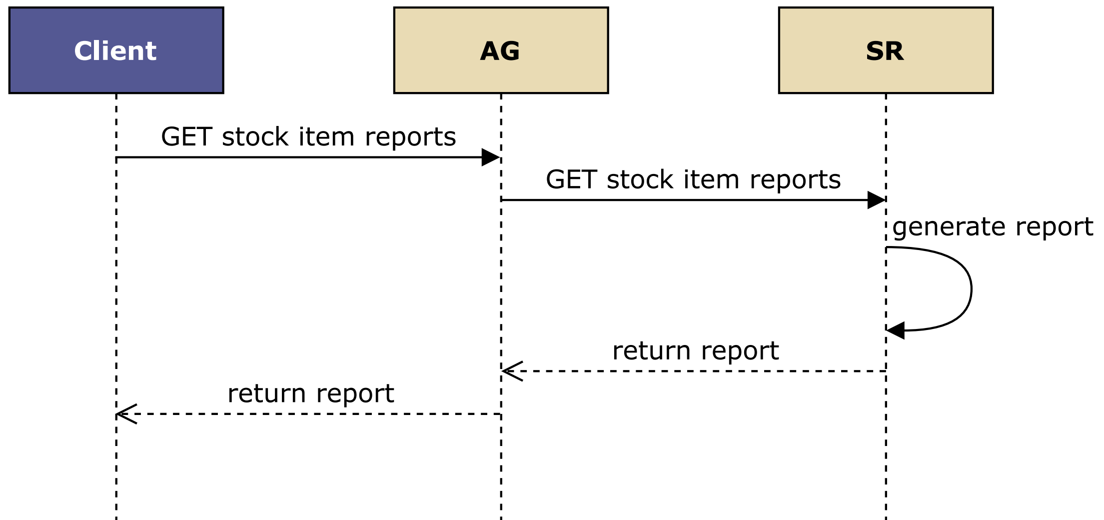

Figure 13: UC 5 sequence diagram for version v03, v04 and v09

service. Thus when a report is requested the service just pulls the data from the database and generates the report. For acquiring the report the Client sends a GET request to the SR service. Upon receiving the request the SR service queries the needed data from the database, generates the report, and returns the report to the Client. All the provided interfaces are RESTful.

The sequence diagram shown in Figure 13 describes the workflow for acquiring the stock item report in v03 (api-gateway-1-to-1-db), v04 (api-gateway-one-db), and v09 (orchestrate-api-gateway-1-to-1-db). The workflow execution of the three versions is the same. The AG (Api-Gateway) is the new component that has been introduced between the Client and the SR service. Thus the operation execution looks like the following: the Client sends a GET request to the AG to retrieve the report. After receiving the solicitation the AG sends a GET request to the SR service for the report. As soon as the SR service registers the incoming request it retrieves the data from the database, generates the report, and sends it back to the AG. Lastly, the AG hands over the report to the Client. All the provided interfaces are RESTful.

The sequence diagram shown in Figure 14 describes the work-

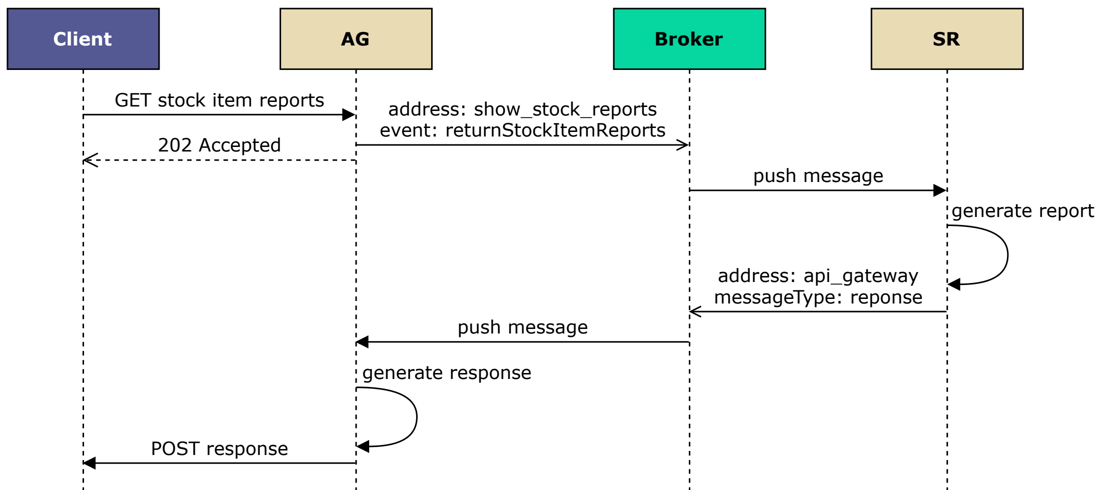

Figure 14: UC 5 sequence diagram for version v05, v06, v07, v10

flow for acquiring the stock item report in v05 (pub-sub-1-to-1-db), v06 (pub-sub-one-db), v07 (message-bus-1-to-1-db) and v10 (orchestrate-pub-sub-1-to-1-db). The order of the tasks to be executed to retrieve the report are the same in all four versions. Unlike the previous five versions that were synchronous, the version depicted in Figure 14 is asynchronous. A new component has been added namely the Broker that handles the point-to-point communication between the AG and the SR service. This is facilitated with the help of queues which can be identified with the help of an address. Every message that is disseminated with the help of the Broker also contains a label that enables the services to determine the type of the message and for what it is intended. The execution of the workflow starts with the Client sending a GET request to the AG which also includes a callback address to which the AG should send the report wants its finishes. As soon as the AG receives the request it publishes the request onto the Broker's queue with the address *show_stock_reports* and with the event named *returnStockItemReports*. If the message has been successfully published the AG sends a request accepted response to the Client. Once the Broker is aware of the incoming message from the AG it forwards it to the SR ser-

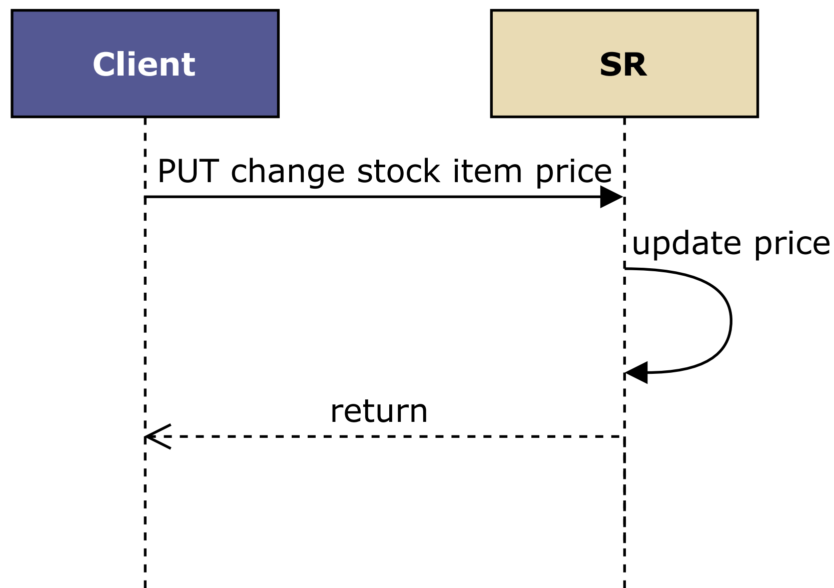

Figure 15: UC 7 sequence diagram for version v01 and v02

vice. The SR service then retrieves the data from the database, generates the report, and publishes it on the Broker's queue with the address *api_gateway*. Upon receiving the message from the SR the Broker tries to deliver the report to the AG. Finally, after the AG receives the report it sends it to the Client with a POST request to the previously provided callback address.

The sequence diagram shown in Figure 15 describes the workflow for changing the stock item price in v01 (master-1-to-1-db) and v02 (master-one-db). As already mentioned the SR service is the one responsible for managing the products of a Store thus it is also the service responsible for setting the price of the products. The versions V01 and v02 have the same workflow for changing the price. The client sends a PUT request to the SR service requesting the update of the price for a given product in a specific Store. Upon receiving the request the SR service updates the price in accordance with the incoming data and returns to the Client the updated stock item. All the provided interfaces are RESTful.

The sequence diagram shown in Figure 16 describes the workflow

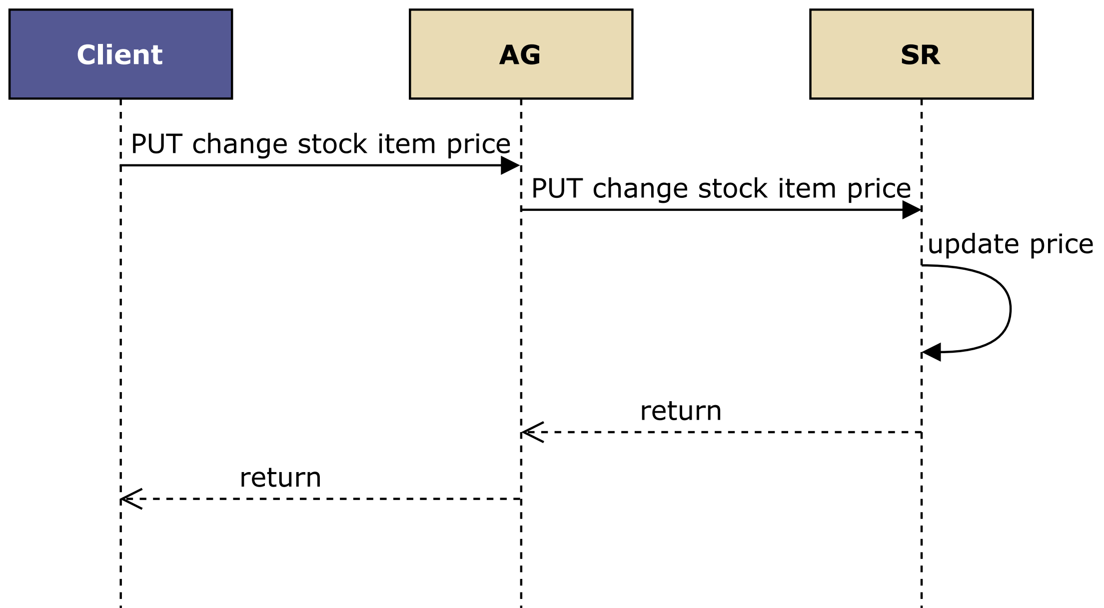

Figure 16: UC 7 sequence diagram for version v03, v04 and v09

for changing the stock item price in v03 (api-gateway-1-to-1-db), v04 (api-gateway-one-db), and v09 (orchestrate-api-gateway-1-to-1-db). The three versions follow the same task execution. The main difference to the versions in Figure 15 is the AG (Api-Gateway) service which sits between the Client and the SR service. The execution starts with the Client sending a PUT request to the AG demanding a price update. After receiving the solicitation from the Client the AG also sends a PUT request downstream to the SR service. Upon receiving the request the SR service updates the stock item price and returns the new item to the AG which in turn forwards the item to the Client. All the provided interfaces are RESTful.

The sequence diagram shown in Figure **??** describes the workflow for changing the stock item price in v05 (pub-sub-1-to-1-db), v06 (pub-sub-one-db), v07 (message-bus-1-to-1-db) and v10 (orchestrate-pub-sub-1-to-1-db). The in the workflow involved services are the same as in the versions seen in Figure 14. The main difference consists in the performed operations. The price change starts with the Client sending a PUT request to the AG which in turn forwards a

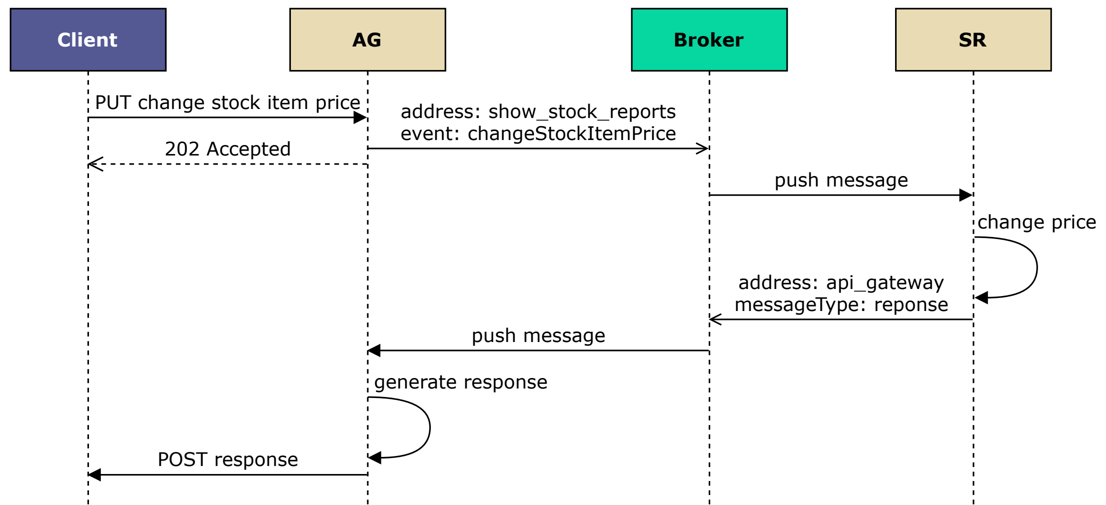

Figure 17: UC 7 sequence diagram for version v05, v06, v07, v10

message onto the Broker's queue with the address *show_ stock_ reports* and the event name *changeStockItemPrice*. After successfully sending the message to the Broker the AG sends a request accepted message to the Client. The Broker delivers the message from the AG to the SR service which updates the item's price and publishes the newly update item onto the Broker's queue with the address corresponding to the AG. The Broker then forwards the message to the AG which in turn delivers the item to the Client via a POST request to the callback address provided by the Client in the initial request.

5.3.2 Order Products and Receive Ordered Products (OP&ROP)

The OP&ROP microservice implements UC 3 and UC 4. This service is not fully self-sufficient needing to interact with the SR service to complete the receiving incoming order use case (i.e UC 4). The data which the service manages directly consists of two tables. One that persists the data regarding the carried out order that contains the product id, the ordered amount, and the order id. Thus UC 3 can be performed without needing another service. The second table contains the data needed to track the order status. This is

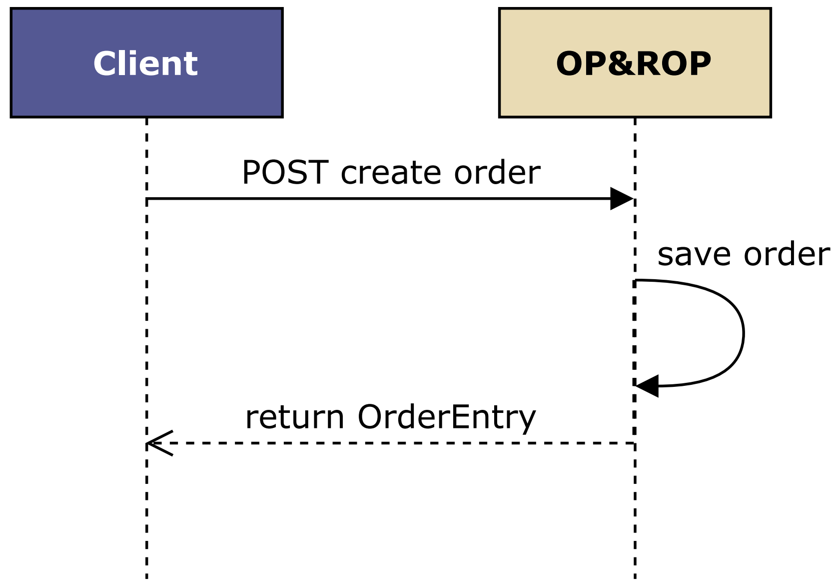

Figure 18: UC 3 sequence diagram for version v01 and v02

accomplished by storing the order id, the ordering- and the delivery date as well as the store id. Thus the tables that are reproduced from the CoCoME data model 2 are ProductOrder which also contains a reference of the Sore id as well as OrderEntry which holds a reference to the Product id.

The sequence diagram shown in Figure 18 describes the workflow for ordering products in v01 (master-1-to-1-db) and v02 (master-one-db). The ordering process is for both versions the same. The workflow starts with the Client sending a POST request to the OP&ROP service containing the order information (i.e. product id and order amount). Upon receiving the request the OP&ROP service creates a new entry in the database inserting a new product and order entry in the database. After successfully persisting the order in the database the OP&ROP service returns the order entry to the Client. All provided interfaces are RESTful.

The sequence diagram shown in Figure 19 describes the workflow for ordering products in v03 (api-gateway-1-to-1-db), v04 (api-gateway-one-db) and v09 (orchestrate-api-gateway-1-to-1-db). These

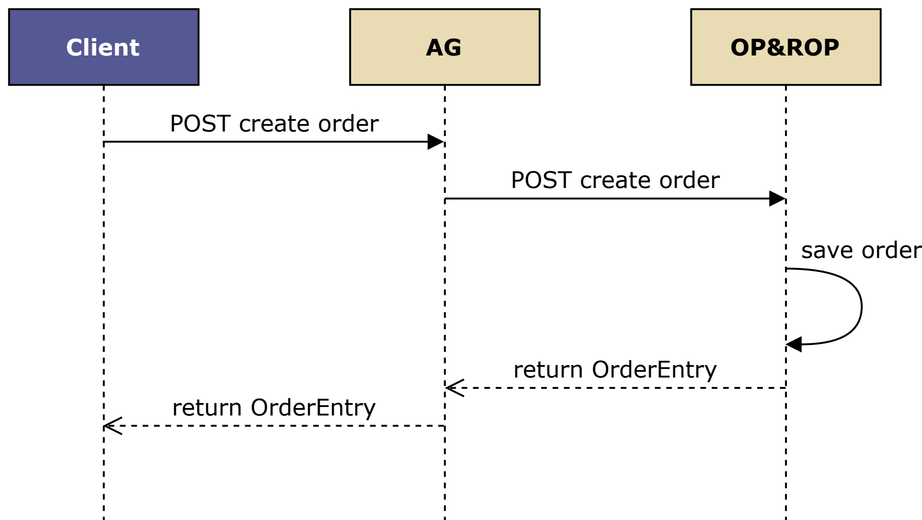

Figure 19: UC 3 sequence diagram for version v03, v04 and v09

three versions are an evolution of the versions provided by Figure 18 by adding the AG (Api-Gateway) service between the Client and the OP&ROP service. The ordering process starts with the Client sending a POST request to the AG soliciting the creating of a new product order based on the provided data. Upon receiving the data the AG forwards the request to the OP&ROP service which saves the order related data and returns the information prevalent to the new order to the AG. The AG service then sends the incoming order data back to the Client. All provided interfaces are RESTful.

The sequence diagram shown in Figure 20 describes the workflow for ordering products in v05 (pub-sub-1-to-1-db), v06 (pub-sub-one-db), v07 (message-bus-1-to-1-db) and v10 (orchestrate-pub-sub-1-to-1-db). These four versions are a progression of the versions shown in Figure 19 by adding the Broker which has to role to decouple the AG service from the OP&ROP service. The previous versions had a blocking workflow being synchronous in nature while the current one is rather asynchronous. The product order starts with the Client sending a POST request to the AG which immediately tries to push the incoming data onto the Broker's queue with the address *order_receive_products* having *orderProduct* as the event

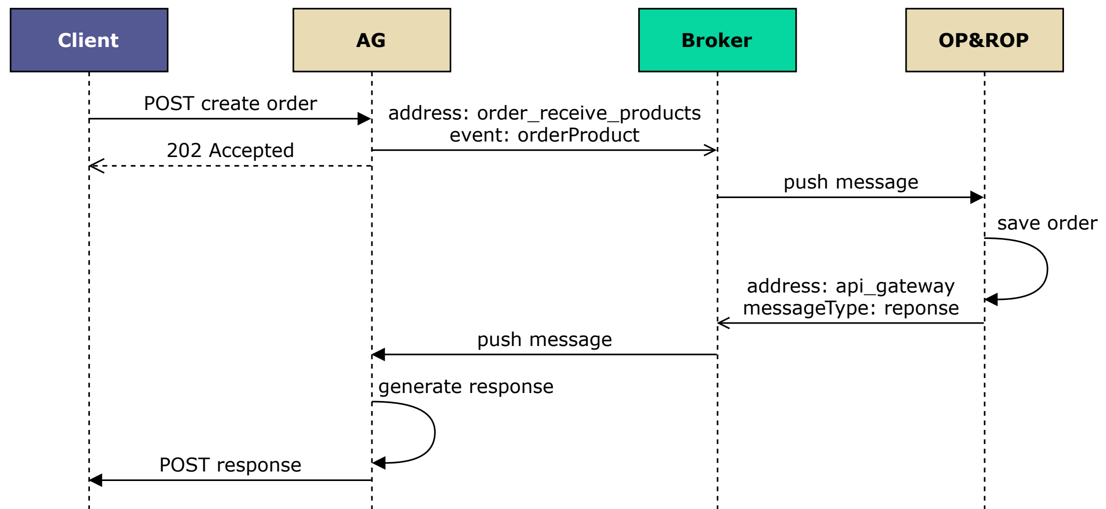

Figure 20: UC 3 sequence diagram for version v05, v06, v07, v10

name. Right after the messages have successfully been sent to the Broker the AG sends a request accepted message to the Client. The Broker pushes the message it received from the AG to the OP&ROP service. The OP&ROP service creates the new order saves it to the database and sends it onto the Broker's queue that is addressed to the AG. The Broker then forwards the order data to the AG which in turn delivers it to the Client by sending a POST request to the callback address provided by the Client in the initial request.

The sequence diagram shown in Figure 21 describes the workflow for receiving the ordered products in v01 (master-1-to-1-db). The process starts with the Client sending a PUT request to the OP&ROP service containing the delivery date of the order. After receiving the request the OP&ROP service sends a GET request to the SR service to inquire about the current stock of the previously ordered product. The SR service returns the current stock amount to the OP&ROP service. After receiving the current stock status the OP&ROP service calculates the new amount and sends a PUT request to the SR service to update the amount of the product. After receiving the solicitation the SR service updates the inventory and returns the updated stock item to the OP&ROP service which

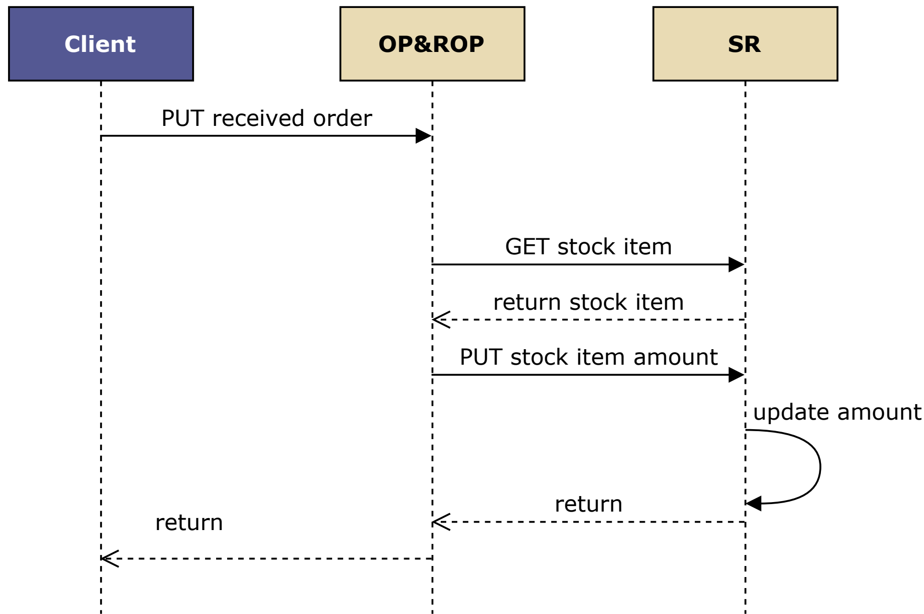

Figure 21: UC 4 sequence diagram for version v01

in turn returns the stock item to the Client and updates the order delivery time. All the interfaces provided by the involved services are RESTful.

The sequence diagram shown in Figure 22 describes the workflow for receiving the ordered products in v02 (master-one-db). This is the simplest version of this use case because the OP&ROP and the SR service share the same database thus the OP&ROP can directly access the stock item table. The process starts with the Client sending a PUT request to the OP&ROP service with the delivery date. Upon receiving the request the OP&ROP service calculates the new stock and updates the product amount and the product order delivery time. Lastly, the OP&ROP service returns the new stock item to the Client. All the interfaces are RESTful.

The sequence diagram shown in Figure 23 describes the workflow for receiving the ordered products in v03 (api-gateway-1-to-1-db). This is an extension of version v01 which is realized by introducing the AG (Api-Gateway) between the Client and all the other ser-

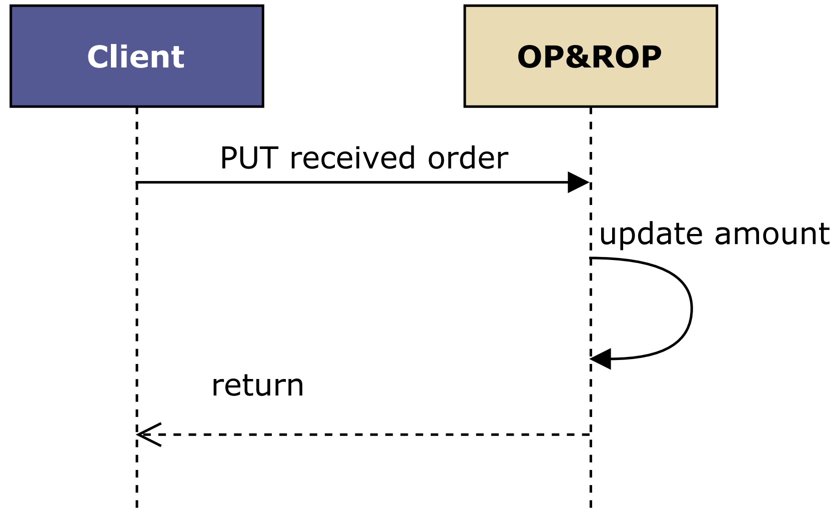

Figure 22: UC 4 sequence diagram for version v02

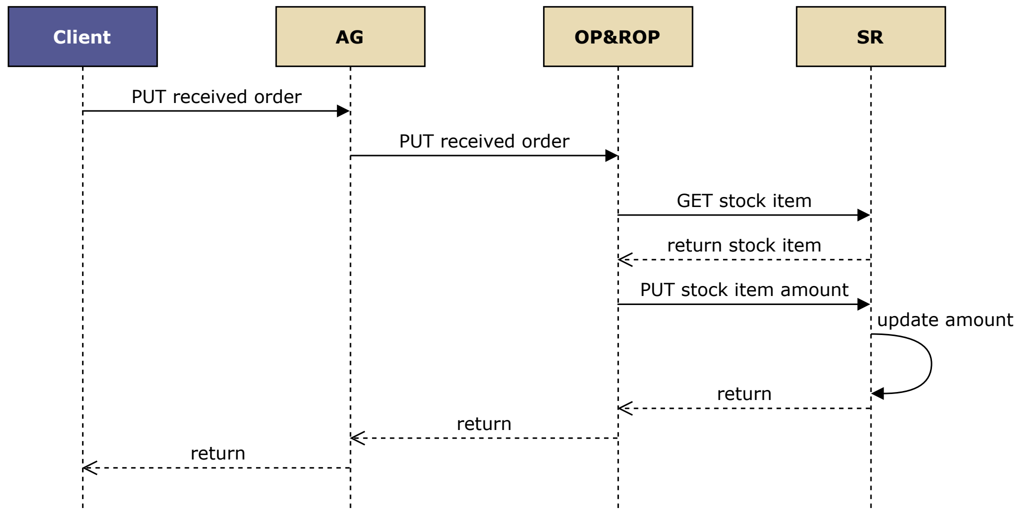

Figure 23: UC 4 sequence diagram for version v03

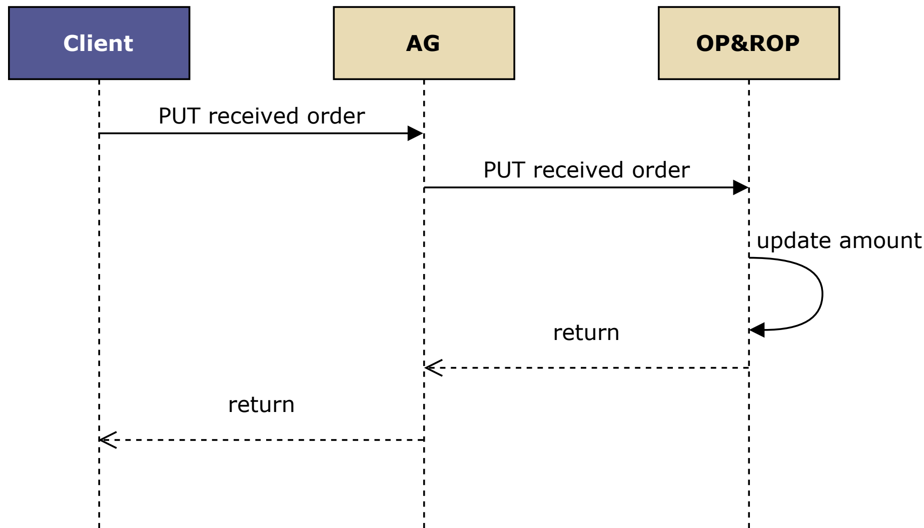

Figure 24: UC 4 sequence diagram for version v04

vices. The process starts with the Client sending a PUT request to the AG with the delivery date of the product order. The AG then forwards the request from the Client to the OP&ROP service. The OP&ROP determines the incoming item and requests from the SR service via a GET request the amount available in the inventory. After receiving the request the SR service queries the database for the item and returns the data to the OP&ROP service. Upon receiving the information the OP&ROP service recalculates the stock item amount and sends a PUT request to the SR service to update the stock item amount. The SR service then performs the update and returns the stock item to the OP&ROP service which in turn updates the order delivery time and forwards the item to the AG. Lastly, the AG return the response to the Client. All the interfaces are RESTful.

The sequence diagram shown in Figure 24 describes the workflow for receiving the ordered products in v04 (api-gateway-one-db). This version is an evolution of version v02 which was achieved by adding the AG service between the Client and the OP&ROP service. The Client starts the workflow by sending a PUT request with the de-

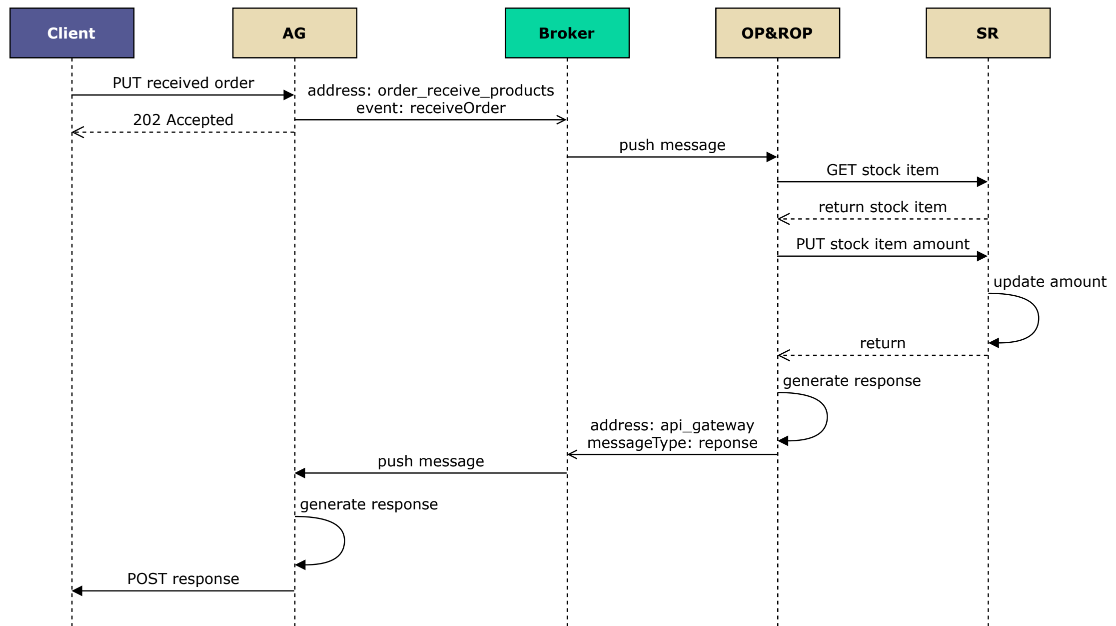

Figure 25: UC 4 sequence diagram for version v05

livery date to the AG service. The AG service then forwards the request to the OP&ROP service via a PUT request. Upon receiving the request the OP&ROP updates the inventory of the newly received item and the order delivery time and returns the item to the AG. Lastly, the AG forwards the stock item to the Client. All the interfaces are RESTful.

The sequence diagram shown in Figure 25 describes the workflow for receiving the ordered products in v05 (pub-sub-1-to-1-db). This is the first version of UC 4 where the information exchange between the service is not solely via REST and synchronous but also contains asynchronous communication which is facilitated by the Broker. The process starts with the Client sending a PUT request to the AG with the delivery date. Upon receiving the request the AG publishes the data to the Broker's queue with the address *order_receive_products* and the event name *receiveOrder*. After successfully pushing the message onto the queue the AG sends a request accepted message to the Client. In the meantime, the Broker

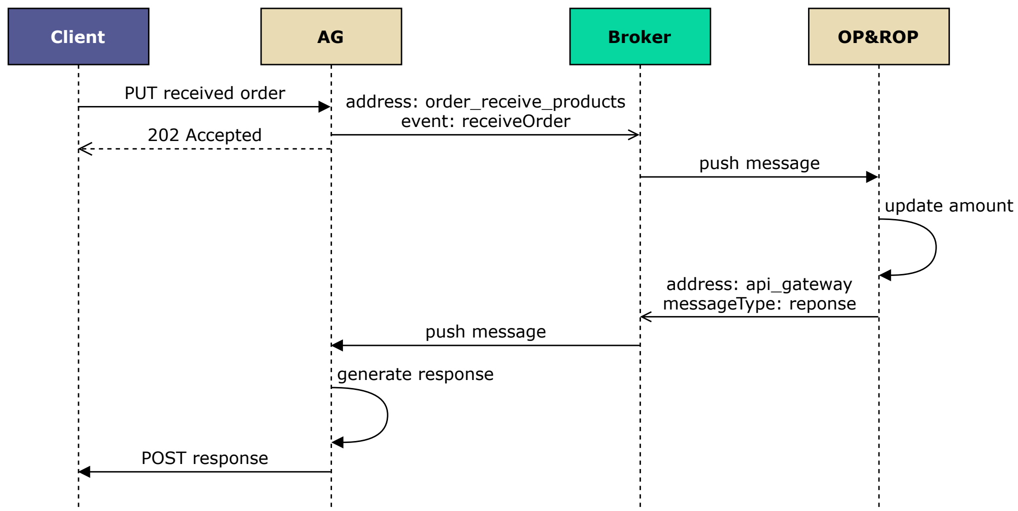

Figure 26: UC 4 sequence diagram for version v06

delivers the message to the OP&ROP service. Upon receiving the message the OP&ROP service determines the item that has been delivered and requests the current inventory item from the SR service via a GET request. The SR service then returns the stock item to the OP&ROP which calculates the new stock value and sends a update request to the SR service via PUT to update the inventory. Upon receiving the request the SR service updates the inventory and returns the item to the OP&ROP service. After receiving the response from the SR the OP&ROP service updates the delivery date and pushes the item to the Broker's queue with the address *api_gateway*. The Broker then delivers the response to the AG which in turn delivers the item to the Client via a POST request to the callback address that was provided together with the initial request.

The sequence diagram shown in Figure 26 describes the workflow for receiving the ordered products in v06 (pub-sub-one-db). The process starts with the Client sending a PUT request with the delivery date to the AG service. The AG then tries to deliver the incoming message to the Broker by pushing it onto the queue to which the OP&ROP service is subscribed. Upon successful delivery,

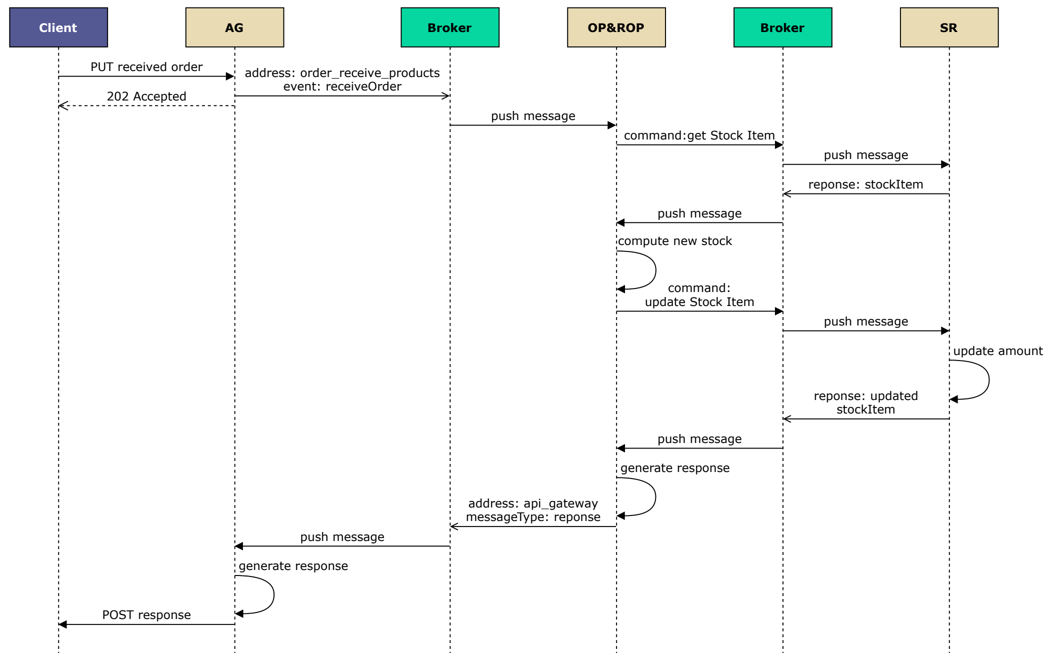

Figure 27: UC 4 sequence diagram for version v07

the AG provides a request accepted response to the Client. Meanwhile, the Broker sends the message from the AG to the OP&ROP service. After receiving the message the OP&ROP service updates the inventory as well as the order delivery date and returns the item to the Broker's queue to which the AG is subscribed. After receiving the data the Broker delivers the message to the AG. The AG then forwards the updated item to the Client by performing a POST request to the callback address previously provided by the Client.

The sequence diagram shown in Figure 27 describes the workflow for receiving the ordered products in v07 (message-bus-1-to-1-db). This version is a variation of the version provided in Figure 25 which has been adjusted by making the communication between the OP&ROP and the SR service asynchronous by introducing a Broker. The Broker makes use of a topic instead of a queue thus making the commutation one-to-many instead of one-to-one. This means that all services that are subscribed to the topic receive all messages that are pushed onto the topic. To facilitate differentiation

between the information the massages have been divided into commands and responses which have been labeled to facilitate an even finer-grained distinction. The Client sends the PUT request with the delivery date to the AG which pushes the data on the Broker's queue to which the OP&ROP service is subscribed. As soon as the AG manages to successfully publish the data it returns a request accepted message to the Client. Next, the Broker pushes the message to the OP&ROP service. Upon receiving the message the OP&ROP identifies the incoming item and pushes the command *get Stock Item* onto the topic managed by the Broker. The Broker then pushes the message to all the subscribed services. Upon receiving the message the SR service realizes that the message is meet for it and acts upon the request. It responds with the stock item, it labels the response with *stockItem* and pushes it onto the common topic. The Broker then pushes the response to the OP&ROP service which calculates the new stock and transmits a new command (i.e. *update Stock Item*) onto the topic which contains the item to be updated. The Broker then again disseminates the command to all the services. The command is picked up by the SR service which updates the inventory. After finishing the update the SR service pushes onto the topic a response labeled *updated StockItem* that contains the updated item. The Broker then pushes the response to all the services. Upon receiving the message the OP&ROP service updates the delivery date and pushes the stock item onto the queue to which the AG is subscribed. The Broker then delivers the message to the AG which in turn returns the item to the Client by executing a POST request to the Client's callback address. The entire workflow can be considered asynchronous.

The sequence diagram shown in Figure 28 describes the workflow for receiving the ordered products in v09 (orchestrate-api-gateway-1-to-1-db). In this version, the AG has the role of an orchestrator meaning that all the communication has to go through the AG. The Client starts the process by sending a PUT request to the AG containing the delivery date. Upon receiving the request the AG forwards the PUT request to the OP&ROP service. After receiving

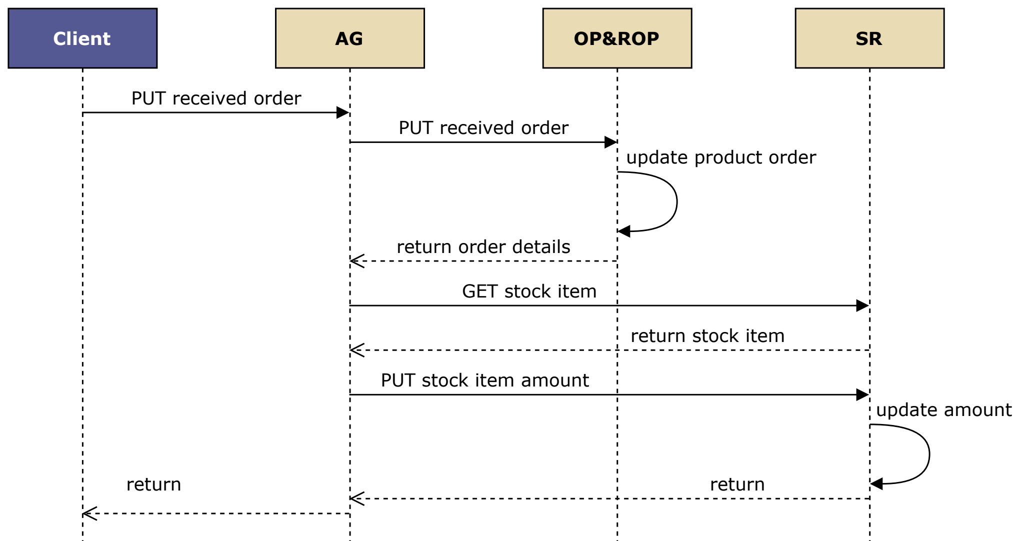

Figure 28: UC 4 sequence diagram for version v09

the request the OP&ROP service updates the delivery date and returns the order details to the AG. The AG then requests the stock item from the SR service via a GET request. The SR service then returns the stock item to the AG which calculates the new item amount and requests the SR service to update the inventory by sending a PUT request with the new stock item. After receiving the request the SR service updates the stock item and responds to the AG with the newly updated stock item. Lastly, the AG returns the stock item to the Client. The entire workflow is synchronous and makes use of RESTful interfaces.

The sequence diagram shown in Figure 29 describes the workflow for receiving the ordered products in v10 (orchestrate-pub-sub-1-to-1-db). This version is similar to the one in version v09, the main difference being that the communication between the AG and the other two microservices is not synchronous but asynchronous. The workflow starts with the Client sending a PUT request to the AG service with the delivery date. Upon receiving the information the AG pushes a message onto the Broker's queue with the address *order_receive_products* and the event name *updateProduct-*

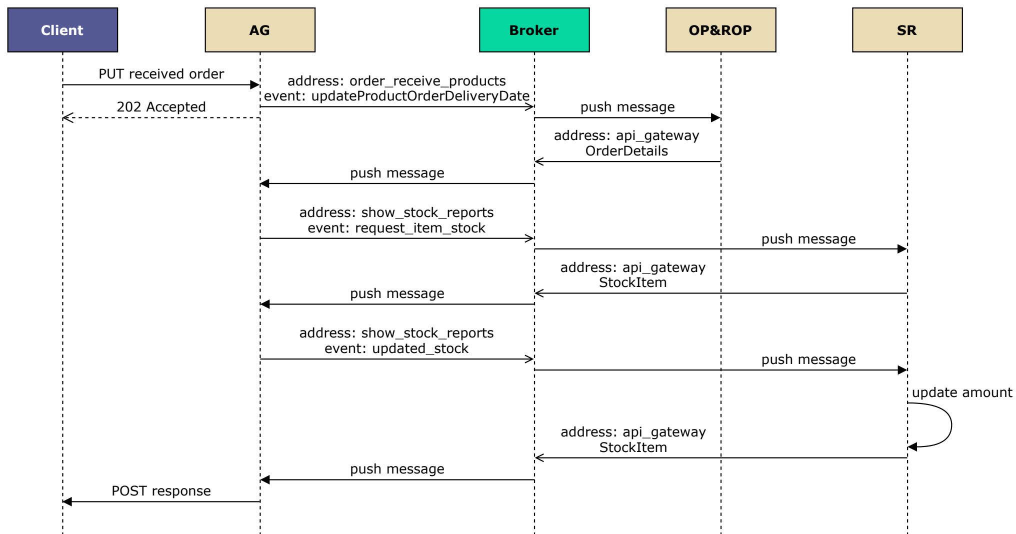

Figure 29: UC 4 sequence diagram for version v10

OrderDeliveryDate. After successfully delivering the message to the Broker the AG sends a request accepted response to the Client. The Broker delivers the message to the OP&ROP service that updates the delivery date and pushes the order details onto the queue to which the AG is subscribed. The Broker then delivers the queued message to the AG. Based on the received order details the AG requests the stock item from the SR by sending a message to the queue with the address *show_stock_reports* and the event name *request_item_stock*. The Broker then pushes the message to the SR service which in turn responds to the AG by adding the stock item details to the *api_gateway* queue. Once the AG receives the stock item information from the Broker it updates the stock item and sends again via the Broker a message to the SR service this time with the event description *updated_stock*. After receiving the stock item information from the Broker the SR service updates the inventory and returns the newly updated item to the AG via the Broker. Finally, after the AG receives the stock item from the Broker it delivers it to the Client by performing a POST request to the callback address belonging to the Client.

5.3.3 Show Delivery Reports (DR)

The DR microservice realizes UC 6. Its database holds four different tables which contain information about the enterprise, the suppliers, and the products they supply to which enterprise. Hence the database contains the following tables from the CoCoME data model (see Figure 2): TradingEnterprise, ProductSupplier, and Product. To be able to properly link the suppliers to the trading enterprise a fourth table has been created that contains the ids from the TradingEnterprise and the ProductSupplier table.

To be able to generate the delivery reports the service needs the data which is being maintained by the OP&ROP service namely the time needed for delivery. Thus depending on the implementation version, the SR service retrieves the data in different ways, which will be discussed next.

The sequence diagram shown in Figure 30 describes the workflow for acquiring the delivery report in v01 (master-1-to-1-db). In this version, the DR, as well as the OP&ROP service, provide a RESTful interface for the other services that wish to access their data and/or functionality. Thus the Client requests the delivery report for a given enterprise by sending a GET request to the DR service which in turn solicits the delivery times for all the products that are being delivered within a specific enterprise. Upon receiving the response from the OP&ROP service the DR service generates the delivery report and sends it back to the Client.

The sequence diagram shown in Figure 31 describes the workflow for acquiring the delivery report in v02 (master-one-db). Similar to the v01 in this version the DR service implements a RESTful interface. Because the DR and OP&ROP share the same database the DR service does not need to request the data from the OP&ROP service but it can retrieve the needed information directly from the database. Then upon receiving the request from the Client the DR service queries the needed data from the database, generates the report, and sends it back to the Client.

The sequence diagram shown in Figure 32 describes the workflow for acquiring the delivery report in v03 (api-gateway-1-to-1-db). The

operations are almost identical to the ones from version v01. The single difference being that between the Client and the DR service the AG (Api-Gateway) has been interposed. In this case, the AG acts as a proxy that accepts requests from the Clients and forwards them to the corresponding services, and upon receiving a response from the services the AG returns the response to the appropriate Client. The AG implements a RESTful interface.

The sequence diagram shown in Figure 33 describes the workflow for acquiring the delivery report in v04 (api-gateway-one-db). This version represents a variation of v02 where the AG (Api-Gateway) has been introduced between the Client and the DR service. Thus after receiving the request from the Client the AG delivers the request to the DR service. Based on the data in the request, the DR service queries the database, produces the report and sends the answer back to the AG that transfers the incoming data to the Client.

The sequence diagram shown in Figure 34 describes the workflow for acquiring the delivery report in v05 (pub-sub-1-to-1-db). Until now within all versions the services communicated with each other in a synchronous way. This is the first version where part of the information exchange within the workflow is done asynchronously. When the Client sends the GET requests for the delivery reports it also provides a callback address to the AG. This callback address is nothing more than a REST endpoint provided by the Client to which the AG can POST the delivery report once the AG gets it from the DR service. Thus as soon as the AG receives the request from the Client it forwards the request to the next hop within the workflow and lets the Client know that its request has been accepted and is being processed. The AG sends the incoming request to the Broker which manages a queue that can be addressed by sending a message to the *show_ delivery_ reports* address. Furthermore, the message contains an event description namely *returnDeliveryReports* which enables the receiver to differentiate between the tasks it has to perform. After the Broker queues the massage from the AG it tries to deliver the message to the DR service. As soon as the message has been successfully delivered it gets removed from the queue and

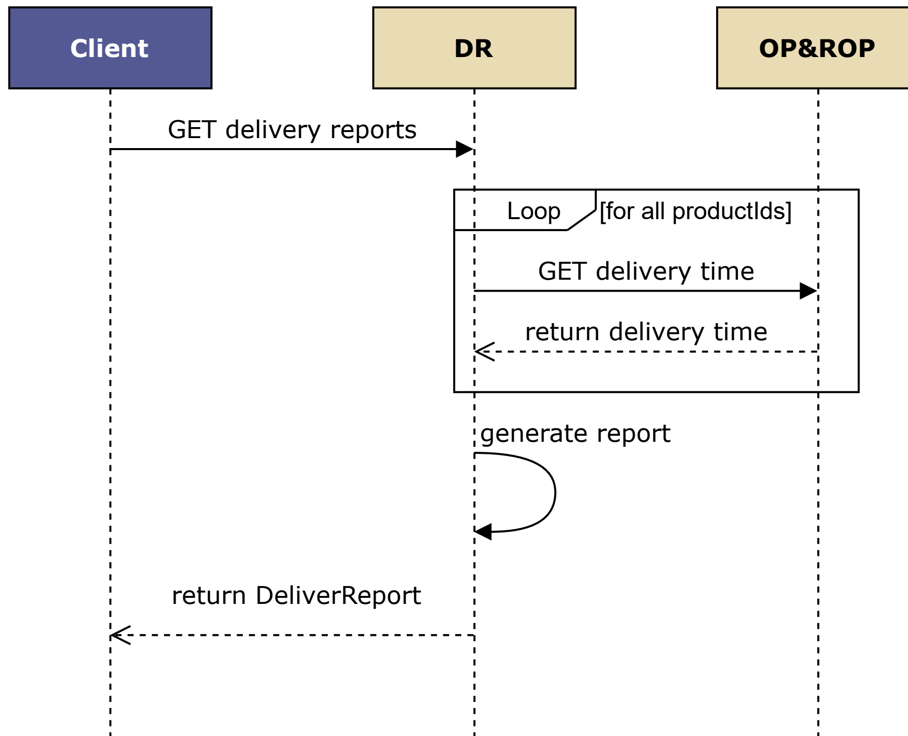

Figure 30: UC 6 sequence diagram for version v01

the DR starts requesting the data needed for the report from the OP&ROP via multiple GET requests. After the DR service has generated the report it is forwarded to the Brokers queue with the address *api_gateway*. After queuing the message the Broker tries to deliver the message to the AG. Upon receiving the message the AG sends the report to the Client by POSTing the payload to the callback address.

The sequence diagram shown in Figure 35 describes the workflow for acquiring the delivery report in v06 (pub-sub-one-db). This is a variation of version v05 where the services share the same database thus the DR service can directly retrieve the needed data from the database and does not need to request it from the OP&ROP service. Thus the use case is executed as follows: The Client requests the report from the AG via a GET request. The AG tries to publish

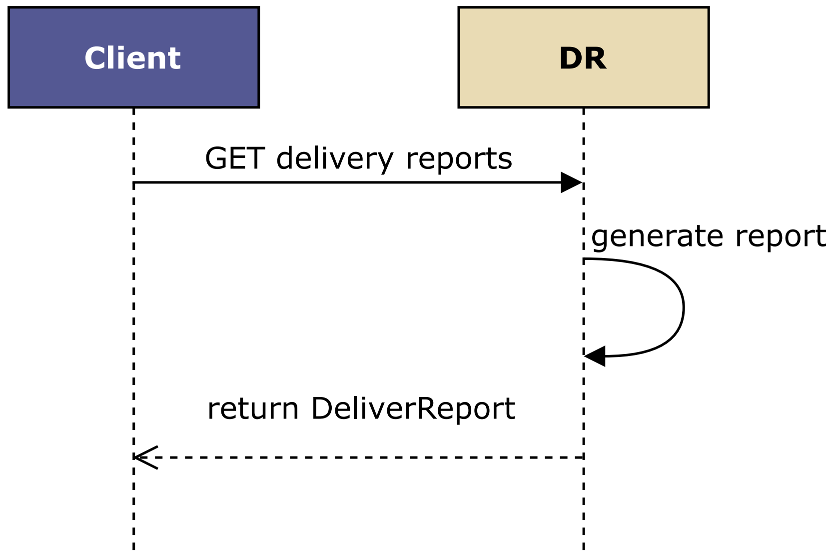

Figure 31: UC 6 sequence diagram for version v02

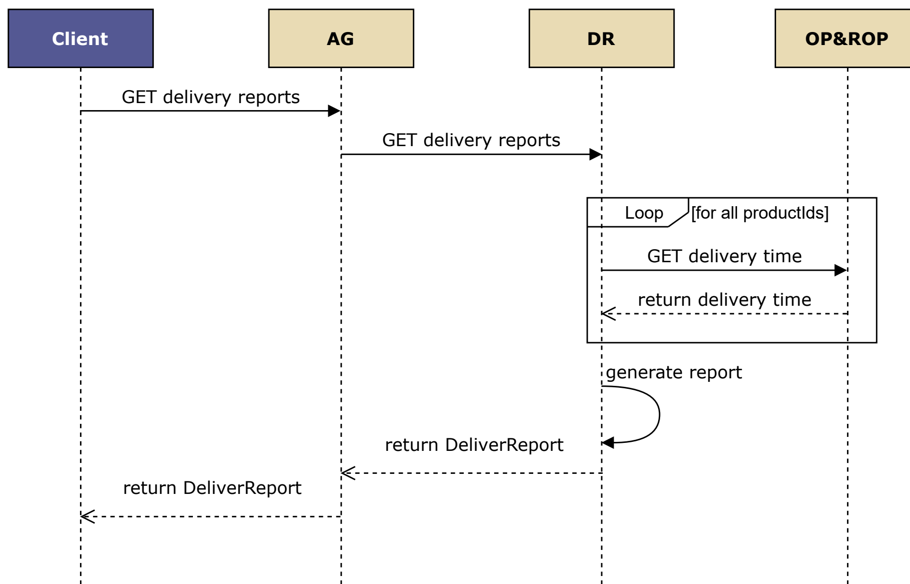

Figure 32: UC 6 sequence diagram for version v03

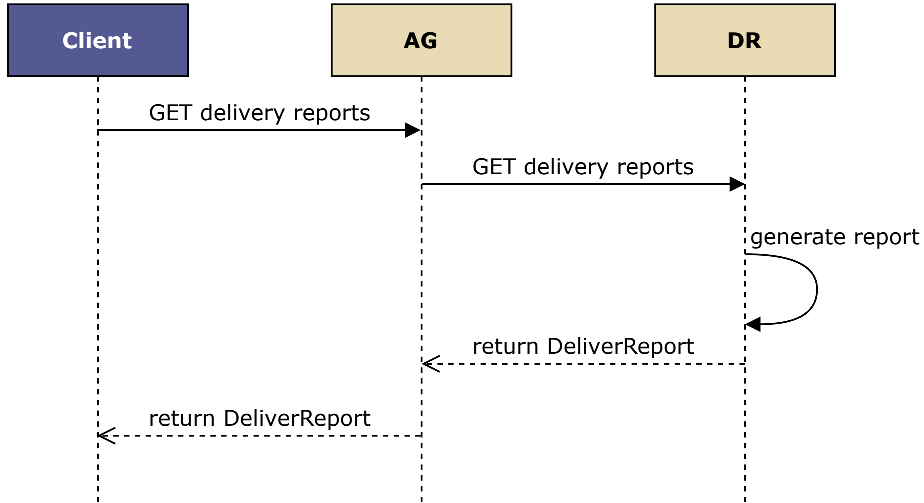

Figure 33: UC 6 sequence diagram for version v04

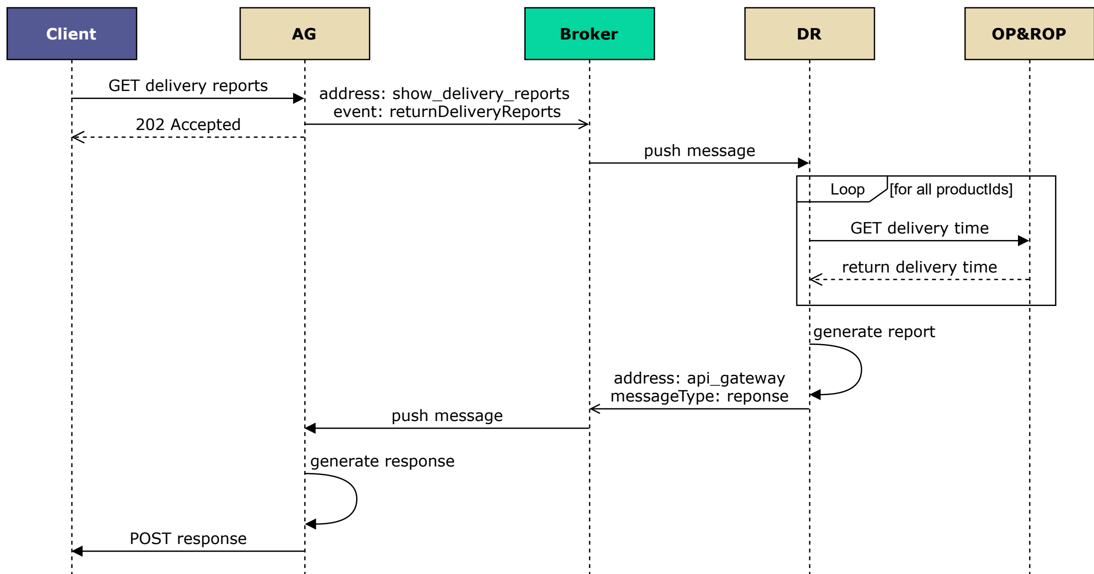

Figure 34: UC 6 sequence diagram for version v05

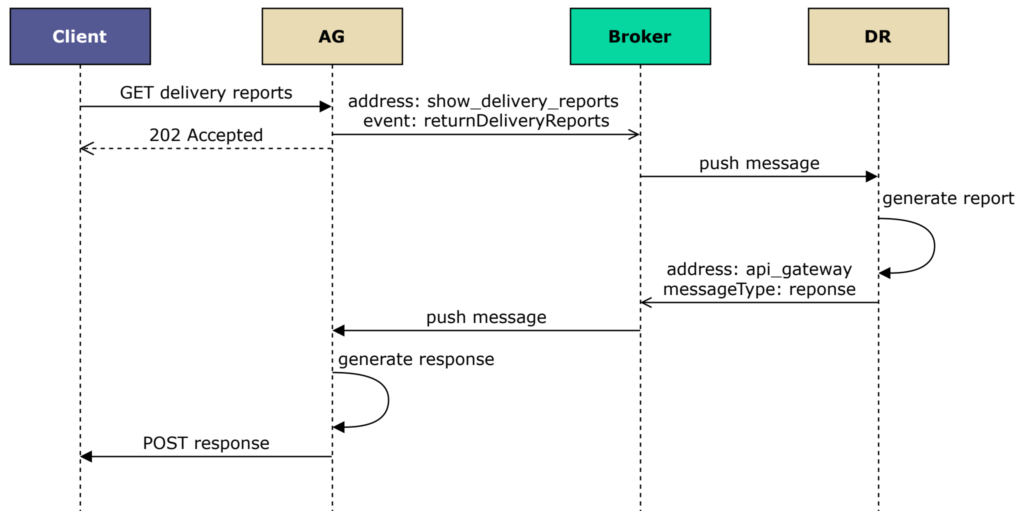

Figure 35: UC 6 sequence diagram for version v06

the request onto the queue and if successful send a request accepted message back to the Client. Once the Broker receives a new message it tries to deliver it to the DR service. After receiving the request the DR service gets the needed data from the database, generates the report, and lastly pushes the report onto the queue managed by the Broker. The Broker forwards the message to the AG which in turn transmits the report to the Client via a POST request to the previously provided callback address.

The sequence diagram shown in Figure 36 describes the workflow for acquiring the delivery report in v07 (message-bus-1-to-1-db). This is a version of UC 6 where all the communication between the involved services is done asynchronously. The execution starts with the Client sending a GET request to the AG which immediately after pushing the message successfully onto the Broker's queue replays to the Client with an acceptance message. The Broker pushes the message further onto the DR service. The DR service sends a command (i.e. *getDuration for ProductSupplierAndProducts*) onto the topic managed by the Broker. A topic in this case behaves like a message bus, meaning that all the services that are subscribed to the topic receive every message published onto the respective topic.

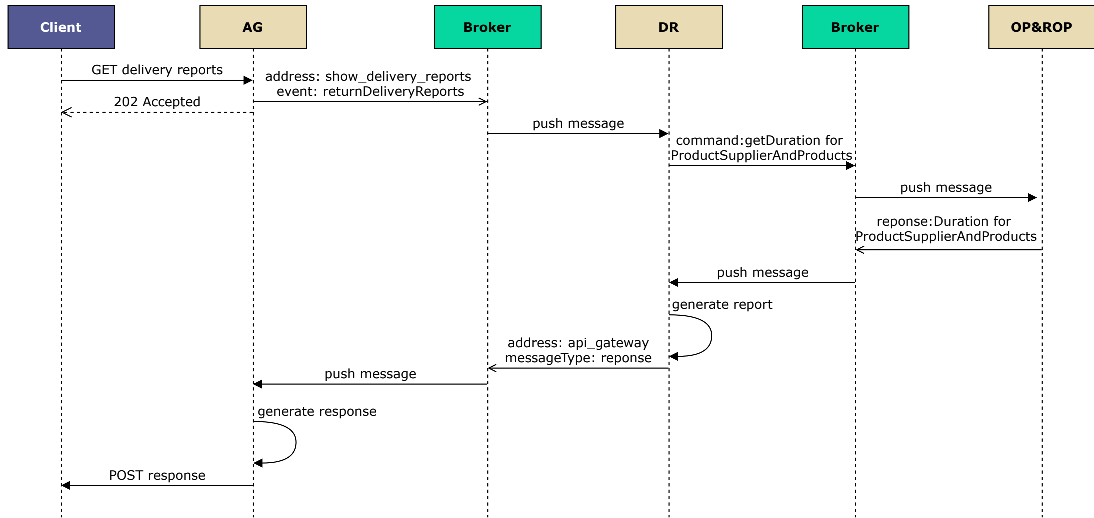

Figure 36: UC 6 sequence diagram for version v07

This implies that the services themselves have to determine if a given message is relevant for them and how to act if it is meant for them. Thus the command description aids the services in taking the right actions. The message queued in the topic is then pushed to all our three services. The OP&ROP service recognizes that the command is meant for it and retrieves the data and pushes it on the topic as a response and marks it with a label that can be recognized by the DR service. Again the Broker pushes the response to all the subscribed services. Next, the DR service retrieves the data from the response message, generates the report, and pushes the report onto the queue which manages the responses meant for the AG. The Broker forwards the incoming report to the AG which in turn POSTs the report to the Client's callback address.

The sequence diagram shown in Figure 37 describes the workflow for acquiring the delivery report in v09 (orchestrate-api-gateway-1-to-1-db). In this version the AG does not act just like a proxy but also functions as an orchestrator, meaning that all the communication between the services is done via the AG. Furthermore, all the communication between the involved parties is synchronous and is realized via RESTful interfaces. The Client sends a GET request

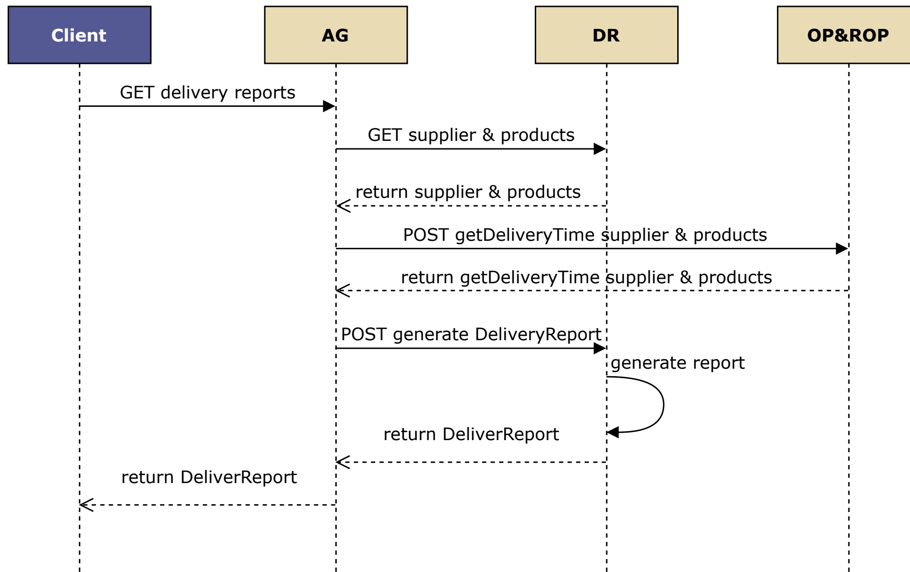

Figure 37: UC 6 sequence diagram for version v09

to the AG soliciting the delivery reports. The AG sends a GET request to the DR service requesting the appropriate suppliers and the products they deliver. The DR responds to the AG demand with the corresponding suppliers and their products. Next the AG sends a POST request to the OP&ROP service with the previously acquired suppliers and products with the intention of receiving the delivery times for the corresponding orders. Upon receiving the request from the AG the OP&ROP services queries the database and returns the answer to the AG. Having the suppliers, the products and the delivery times the AG sends all the data to the DR service via a POST request and asks for the generation of the delivery report. The DR service generates the delivery report based on the received data and sends it back to the AG. The AG responds to the initial GET request with the generated delivery report.

The sequence diagram shown in Figure 38 describes the workflow for acquiring the delivery report in v10 (orchestrate-pub-sub-1-to-1-db). This version is similar to version V09 in that the AG again functions as an orchestrator but this time the entire service

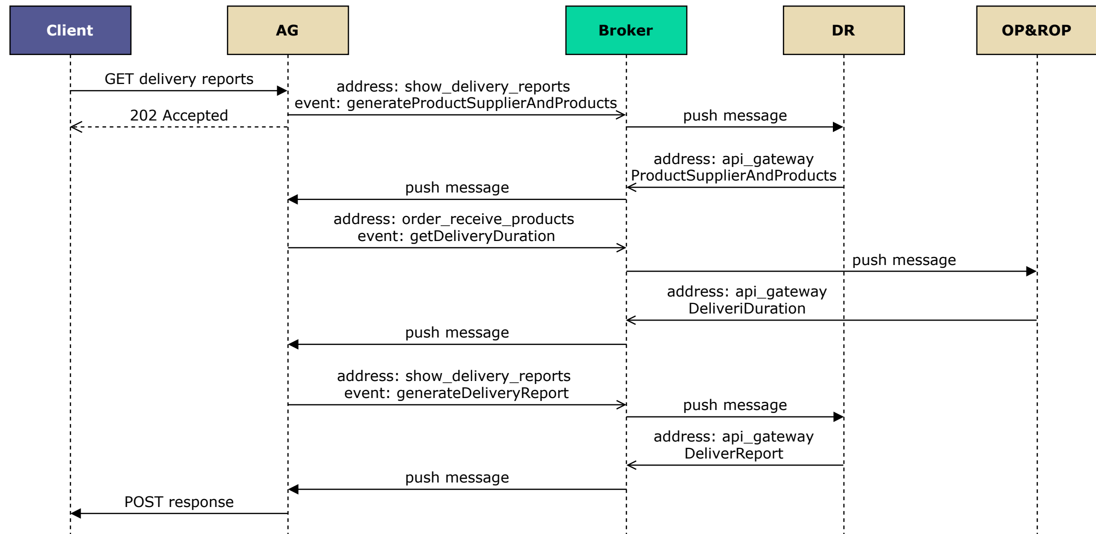

Figure 38: UC 6 sequence diagram for version v10

exchange is asynchronous and makes use of a Broker and queues. The workflow starts with the Client sending a GET request to the AG asking for the delivery reports. Upon receiving the request the AG sends a message to the Broker corresponding to the queue with the address *show_delivery_reports* and the event name *generateProductSupplierAndProducts* which is meant for the DR service. After successfully sending the message to the Broker the AG sends a request accepted message to the Client. Upon receiving the message the Broker forwards it to the DR service. After receiving the message the DR service acquires the corresponding suppliers and their products and pushes the payload onto the Broker's queue on which the AG is listening. After receiving the payload the Broker delivers it to the AG. The AG sends a message to the OP&ROP via the Broker's queue having the address *order_receive_products* and the event named *getDeliveryDuration*. The Broker then forwards the message to the OP&ROP service. Based on the information within the received message the OP&ROP gets the delivery duration and pushes it onto the queue with the AG's address. The Broker then forwards the message to the AG which again sends a message to the

DR requesting the report generation via the Broker. The Broker sends the message to the DR service which generates a delivery report and pushes it back on the Broker which itself forwards it to the AG. Lastly, the AG sends the report to the Client by sending a POST request to the callback address of the Client.

5.4 Deployment

The microservices are being deployed with the help of Jenkins more exactly every implementation has its own pipeline which has to be triggered manually. Every pipeline consists of multiple stages:

- Build - where all the jars are built with maven
- Build Image - for every service a docker image is built
- Push Image - every image is pushed to the Docker hub
- Download and run Images - every docker image is downloaded on a different VM and ran

While all the microservices run within docker the database (PostgreSQL[6]), the broker (Apache ActiveMQ Artemis[7]) and Jenkins do not make use of docker but run directly within the VM. To better emphasize the differences between the chosen architectures the following deployment has been deemed as fair:

- Every microservices runs on his own VM alongside his dedicated database
- In the scenarios with the shared database pattern the OP&ROP VM has been chosen as host because this service makes use of the DB the most
- The broker runs on the same VM as the Api-Gateway
- Apache JMeter[8] and the Client (WebHook) run on their dedicated VM

[6] https://www.postgresql.org/
[7] https://activemq.apache.org/components/artemis/
[8] https://jmeter.apache.org/

The Figures from 39 to 47 offer a detailed view of deployment per scenario.

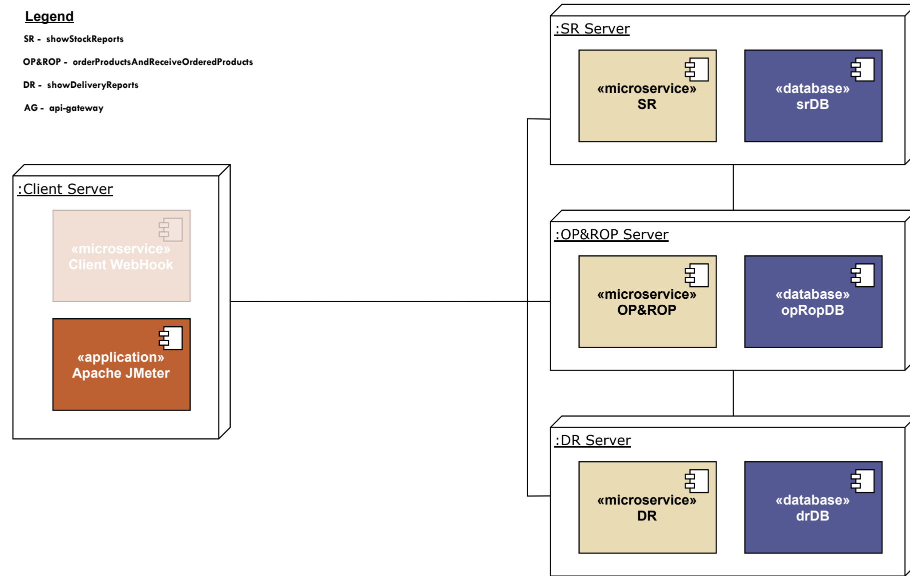

Figure 39: Master with one DB per service

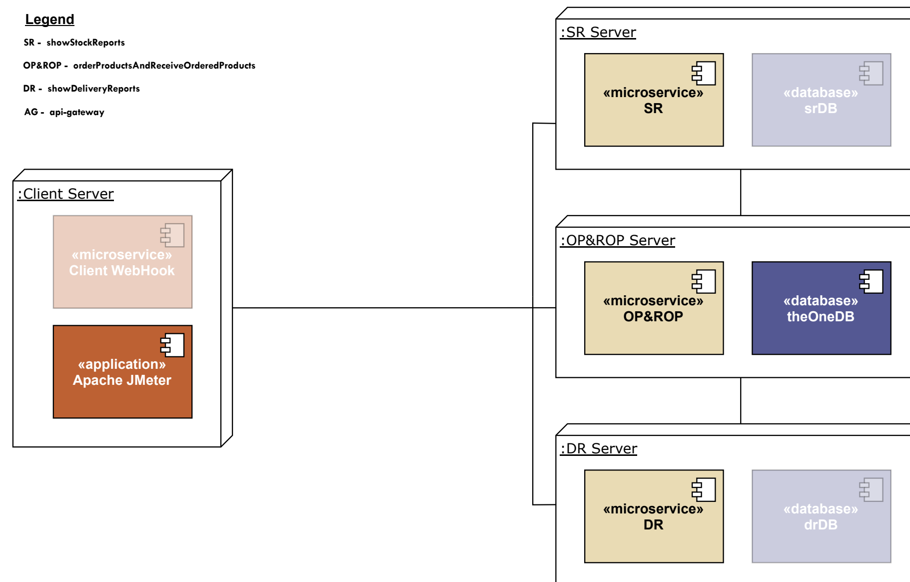

Figure 40: Master with one common DB

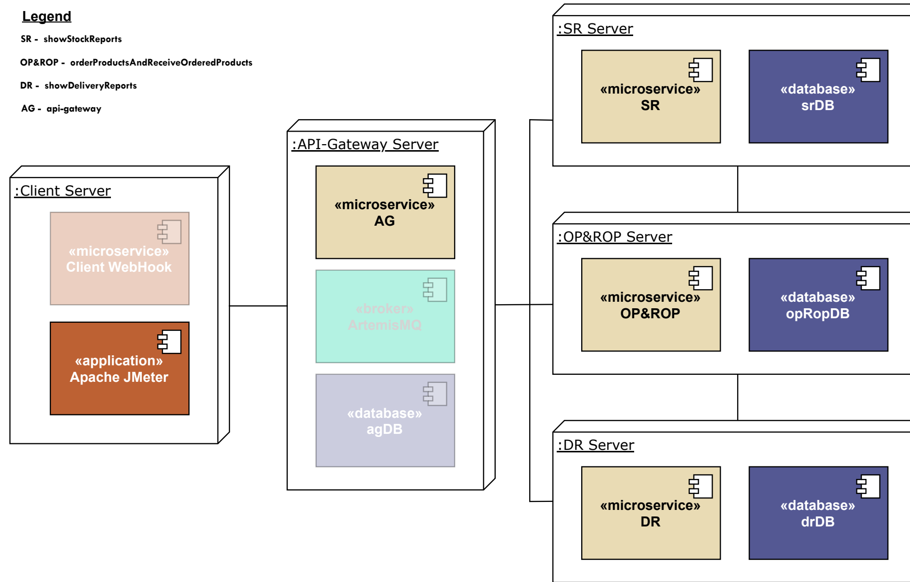

Figure 41: Api-Gateway with one DB per service

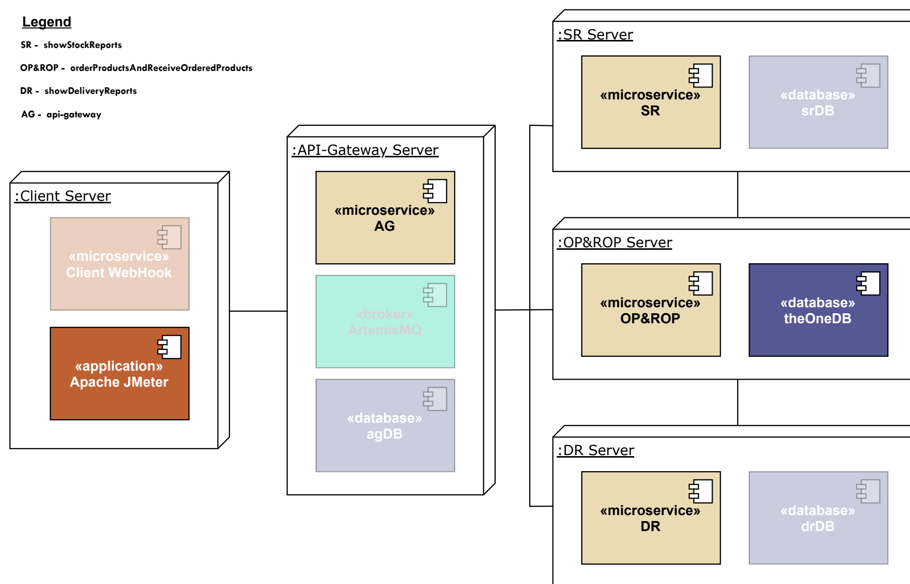

Figure 42: Api-Gateway with one common DB

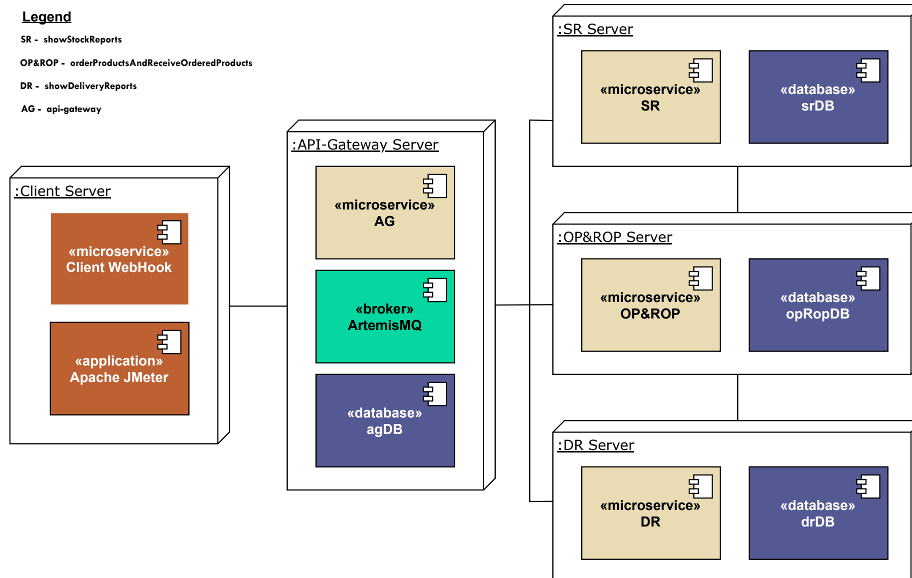

Figure 43: Pub-Sub with one DB per service

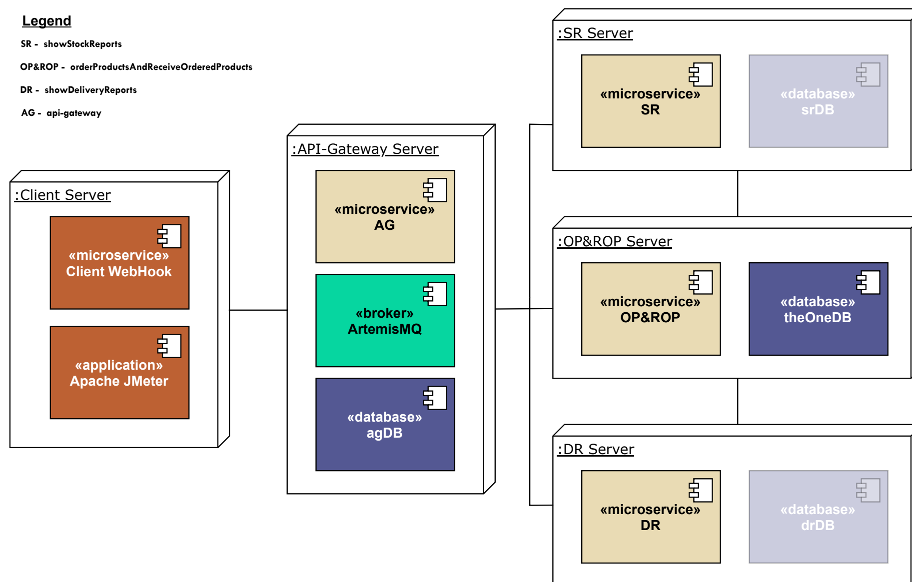

Figure 44: Pub-Sub with one common DB

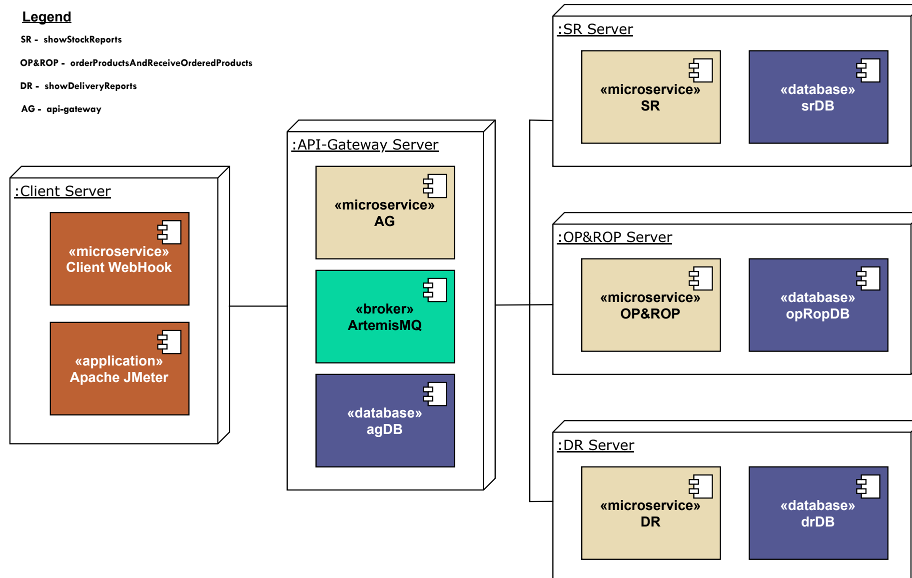

Figure 45: Message bus with one DB per service

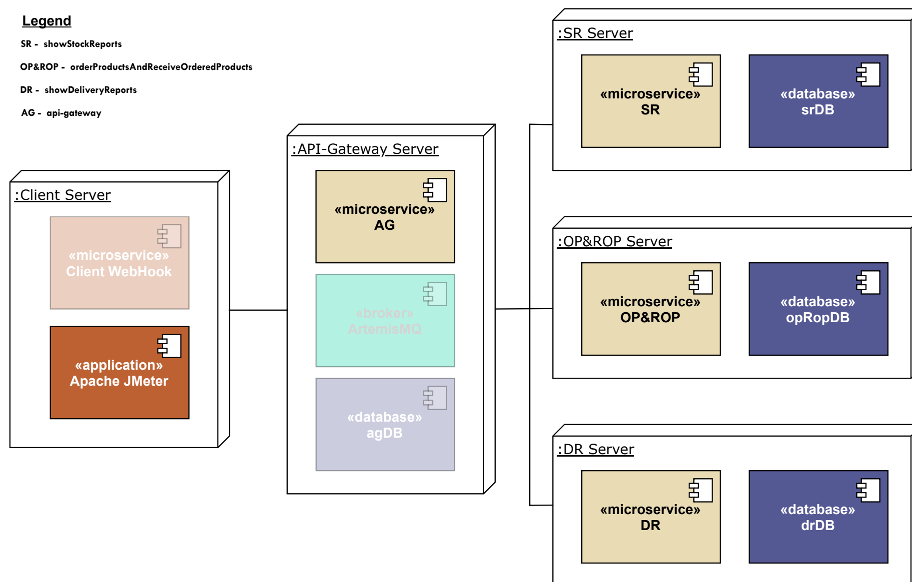

Figure 46: Api-Gateway orchestration with one DB per service

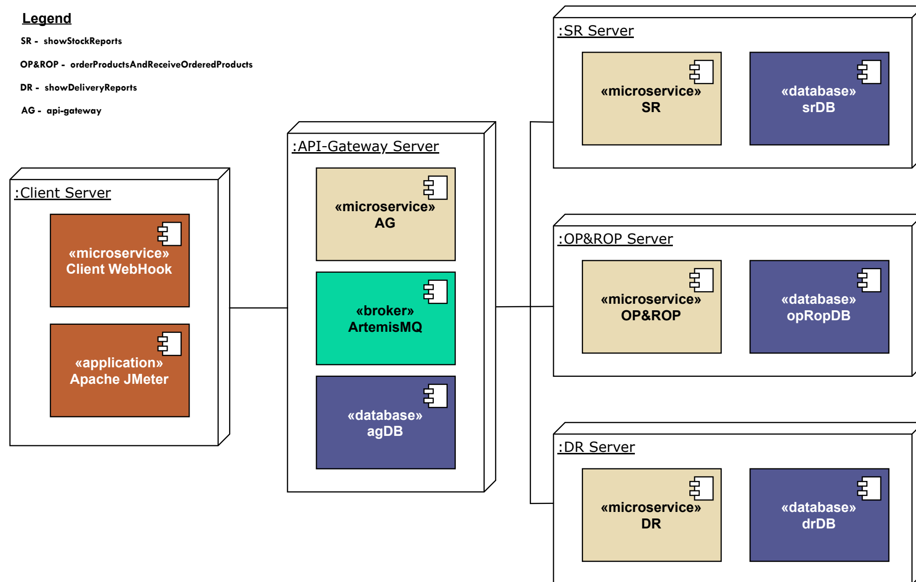

Figure 47: Pub-Sub orchestration with one DB per service

6 Measurement Generation

As already mentioned in Section 5.4, JMeter is one of the tools that has been deployed. Its role is to assess the performance of each of the nine presented implementations. For every version, a test plan has been generated that facilitates the measurement of their performance. Every plan consists of thread groups that correspond to the existing five use cases. Within every thread group, three parameters can be adjusted to simulate different load scenarios. These are:

- Number of threads (NoT) - represents the number of users who will access the UC functionality
- Ramp-up period (RuP) - the time (seconds) until all users (threads) have been started. A NoT of 10 and a RuP of 10 would mean that every second a new user is being generated
- Loop Count - determines how often a user will trigger a UC

Figure 48 exhibits the structure of one of the used test plans. On the test plan level, it can be decided if the threads should all start running at the same time or in a sequential manner.

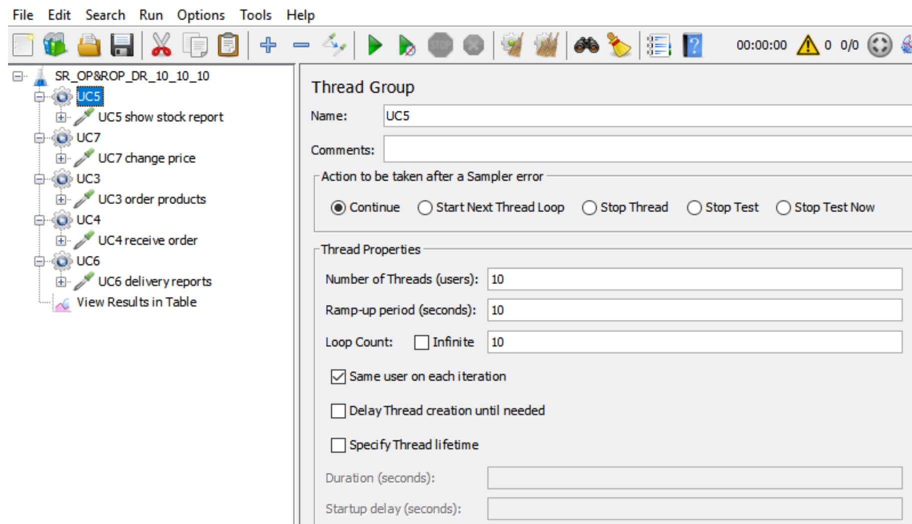

Figure 48: JMeter test plan

A thorough search did not yield any best practices regarding the setup of a measurement for the evaluation of a service. In all found documented cases the number of users and the number of requests are known. Hence based on this predefined data it is possible to determine if a given service can provide the desired runtime or not. Furthermore, in most cases, the tested system is a production-ready system and not a proof of concept.

The intention was to test the services themselves and not the hardware they are running on. For this reason, htop [9] was used to determine the load that every service undergoes when the measurements are executed. Figure 49 shows the setup used when running the measurements with the help of JMeter against the existing services.

The window in the foreground represents the VM on which JMeter is running. The windows in the background represent the VM on

[9] https://htop.dev/

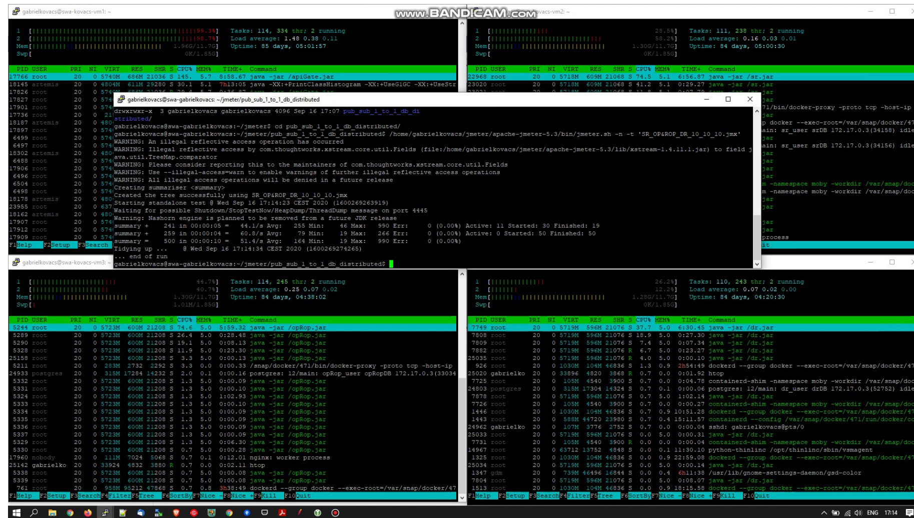

Figure 49: JMeter measurement execution

which the four existing microservices are running. Starting from the left upper corner and going clockwise these are AG, SR, OP&ROP, and DR.

Multiple iterations have been tested until a relatively similar distribution, along with all implementations, could be observed. The final chosen measurement setup consisted of one user (thread) executing 300 requests which are 300 milliseconds apart from each other. This setup was applied for each use case (thread group). All the thread groups started executing their requests at the same time. Thus per every individual run, 1500 data points are being generated. For every implementation, the individual run has been executed 14 times yielding a number of 21.000 data points per implementation. The resulting graphical representations of the measurements can be seen in Section 7 while the JMeter source code and the resulting data can be found within this project [23] hosted publicly.

7 Data Evaluation

This Section aims to determine if and what insights can be gained from the previously gathered data. The main goal is to work out a

model which enables a software architect to determine how a specific combination of microservice patterns impact the runtime of the entire system. But at the same time, the already gathered data can also be analyzed thus allowing the evaluation of the system from a different perspective. As already mentioned in Section 6 the available data is reduced to the existing patterns (categorical data) and the runtime (continuous data) in milliseconds.

Version Nr	Name
v01	master-1-to-1-db
v02	master-one-db
v03	api-gateway-1-to-1-db
v04	api-gateway-one-db
v05	pub-sub-1-to-1-db
v06	pub-sub-one-db
v07	message-bus-1-to-1-db
v09	orchestrate-api-gateway-1-to-1-db
v10	orchestrate-pub-sub-1-to-1-db

Table 1: Implementations versions

Version Nr	Min	Q1	Median	Q3	Max
v01	2	4	6	14	27
v02	2	5	6	8	12
v03	4	8	11	19	34
v04	4	8	10	12	18
v05	12	21	26	35	56
v06	15	22	26	32	47
v07	13	23	29	40	65
v09	4	8	11	17	30
v10	14	23	35	51	93

Table 2: Boxplot Values

Before building the statistical model we take a look at how every implementation performs based on the previously executed measure-

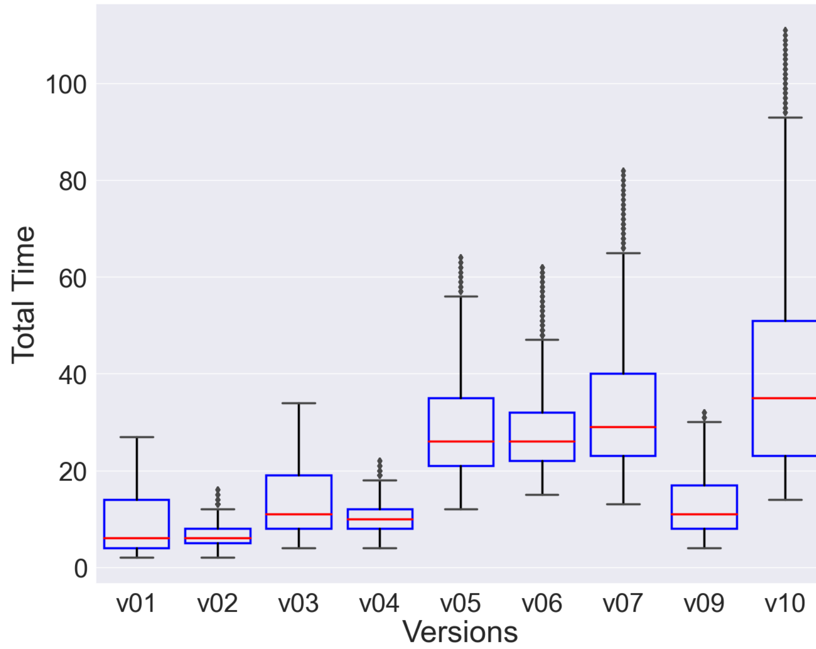

Figure 50: Box Plot

ments. Table 1 contains the version number for every implementation and their respective names. Table 2 contains the specific values contained within the boxplot defined in Figure 50. As expected versions v01 and v02 manage to provide the fastest response to the client. The numbers are comparable until the third quartile (Q3) after that the shared database pattern starts to outperform the synchronous RESTful communication between the services. Versions v03 and v04 expanded the previous two versions by introducing the API-Gateway pattern. The runtime until Q3 is almost doubled in comparison to v01 and v02 while starting with Q3 the discrepancy becomes smaller and tappers of towards the upper limits. Version v05 and v06 are a variation of v03 and v04 where the communication between the API-Gateway and the other services is being done asynchronously via the publish-subscribe pattern instead of synchronous via REST. The minimum value of v05 is slightly lower (circa 20%) than the value of v06 while the values from Q1 to Q3 are almost

identical. At the maximum value, the situation gets reversed and v06 outperforms v05 by about 16%. The communication within v07 is fully asynchronous. The values are very similar to the ones from v06. Starting with the median the values in v07 start increasing slowly until they reach an increase of approximately 28% at the maximum in comparison to v06. Version v09 produces similar results as v03 and v04. Starting from the minimum value until including the median the vales are identical to v03 and starting with Q3 v09 slightly outperforms v03. At Q3 v09 has an approximately 30% increase of runtime in comparison to v04 reaching circa 40% towards the maximum value. Version v10 which is a variation of versions v05 and v06 produces similar results to the two. The minimum and Q1 are comparable to v06 but starting with the median an approximately 25% increase can be observed in v09 that continues increasing until at the maximum the discrepancy reaches circa 50%. It can also be observed that versions v06, v07, and v10 display a higher number of outliers while v02, v04, v05, and v09 show a lower number of outliers, and lastly v01 and v03 pose no outliers at all thus indicating the most consistent behavior.

The following distribution plots ranging from Figure 51 to 59 provide an insight into the effectuated measurements for the distinct implementations.

The distribution showed in Figure 51 represent the total time in milliseconds (ms) needed for all requests to finish, that have been send against the v01 implementation. The plot is right-skewed and appears to have one peek (unimodal) which spreads from roughly 3 ms to about 6 ms. There are multiple gaps between the bins which show us that some of the numbers with regards to runtime have never been hit.

The distribution showed in Figure 52 represent the total time in milliseconds needed for all requests to finish, that have been sent against the v02 implementation. The distribution is right-skewed and has a peak (unimodal) at about 5 ms. After the peak, the number of requests whit a longer runtime tends to steadily decline. The bins themselves are very narrow indicating that not the entire

values between the min and the maximum value have been hit.

The distribution showed in Figure 53 represent the total time in milliseconds needed for all requests to finish, that have been sent against the v03 implementation. This distribution is also right-skewed having a peak (unimodal) at about 8 ms. Again multiple gaps between the gaps can be observed. The plot is very similar to the one of version v01. The main difference is that all the values have been shifted to the left thus as expected the introduction of the Api-Gateway increases the overall runtimes.

The distribution showed in Figure 54 represent the total time in milliseconds needed for all requests to finish that have been sent against the v04 implementation. This distribution is right-skewed having a peak (unimodal) at about 7.5 ms. After the peak, the number of requests with a longer runtime is steadily going down. The bins have a comb-like appearance. A similarity to version v02 can be observed.

The distribution showed in Figure 55 represent the total time in milliseconds needed for all requests to finish that have been sent against the v05 implementation. This distribution is right-skewed having what it seems like two peaks (bimodal) one at around 20 ms and another one at around 27 ms. After the second peak, the number of requests with higher runtimes starts to tapper of not before having something of a plateau between 32 and 38.

The distribution showed in Figure 56 represent the total time in milliseconds needed for all requests to finish that have been sent against the v06 implementation. This distribution is right-skewed having a peak (unimodal) that resembles a plateau that spans from about the 22 until the 27 mark. After the end of the peak, the height of the bins starts to monotonously get smaller. Multiple gaps between the bins can be observed.

The distribution showed in Figure 57 represent the total time in milliseconds needed for all requests to finish that have been sent against the v07 implementation. This distribution is right-skewed having multiple peaks (multimodal). The first peak is at about 18 ms followed by the highest peak at about 24 ms, the second highest

is at around 28 ms. The next three peaks are very similar in size and can be observed at roughly 35 ms, 39 ms, and 44 ms. Beginning with the peak near 50 ms they start to get smaller in relationship with the bins surrounding them. While the requests with longer runtimes get fewer the tapering is not as smooth having multiple ups and downs until the maximum value is reached.

The distribution showed in Figure 58 represent the total time in milliseconds needed for all requests to finish that have been sent against the v09 implementation. This distribution is right-skewed having what seems like two peaks (bimodal) one at around 7 ms and another one at around 17 ms. The distribution around the peaks seems to be fairly symmetric. This plot also contains multiple gaps between its bins.

The distribution showed in Figure 59 represent the total time in milliseconds needed for all requests to finish that have been sent against the v10 implementation. The distribution has two peaks (bimodal) one at about 20 ms and the second at about 45 ms. The fall after the first peak is very steep while the one after the second peak is much smoother.

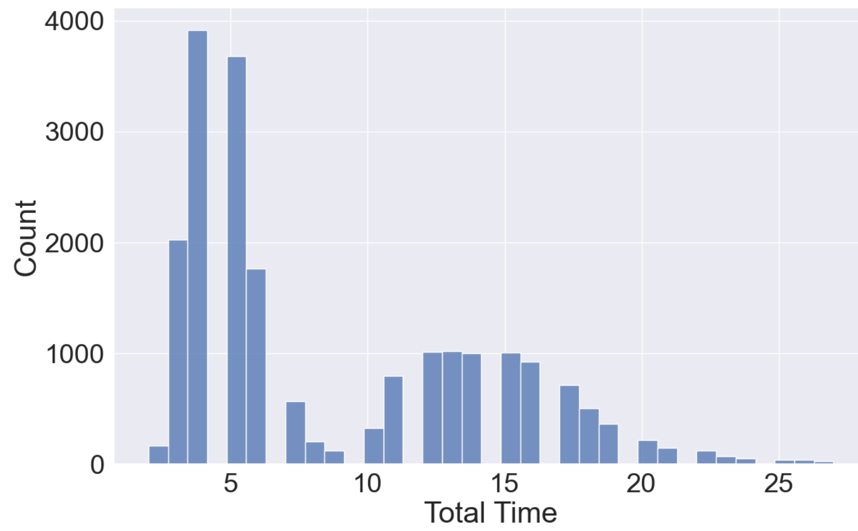

Figure 51: Master 1-to-1 db (v01)

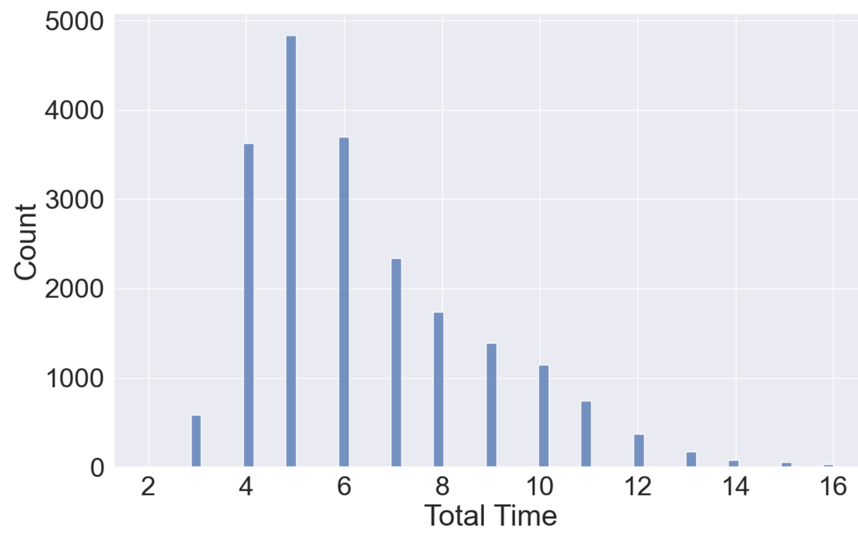

Figure 52: Master one db (v02)

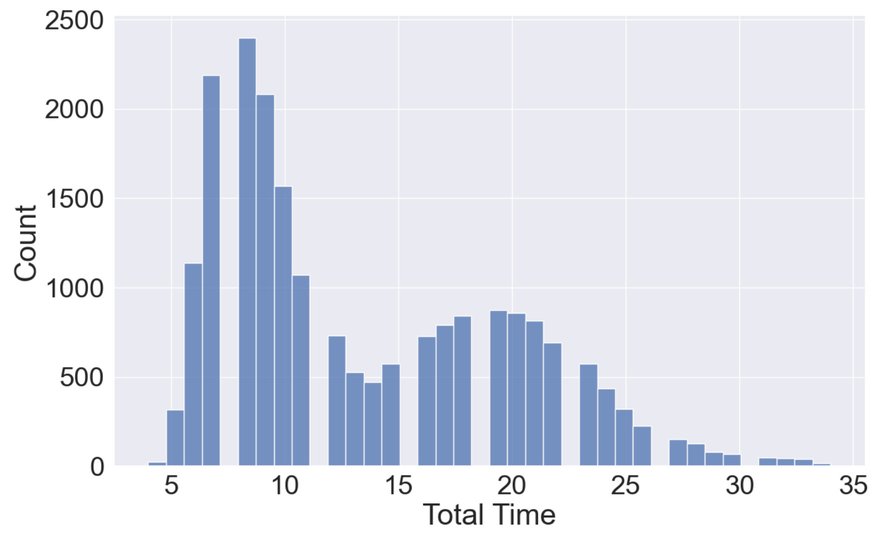

Figure 53: ApiGateway 1-to-1 db (v03)

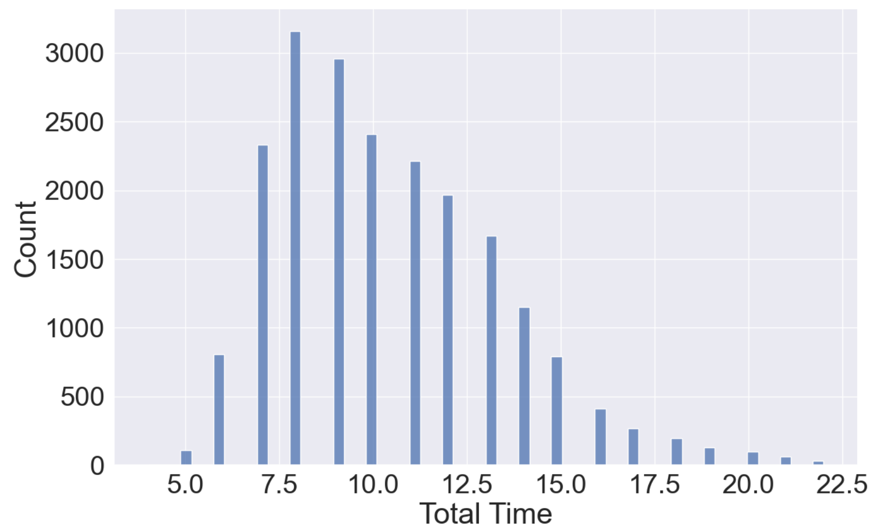

Figure 54: ApiGateway one db (v04)

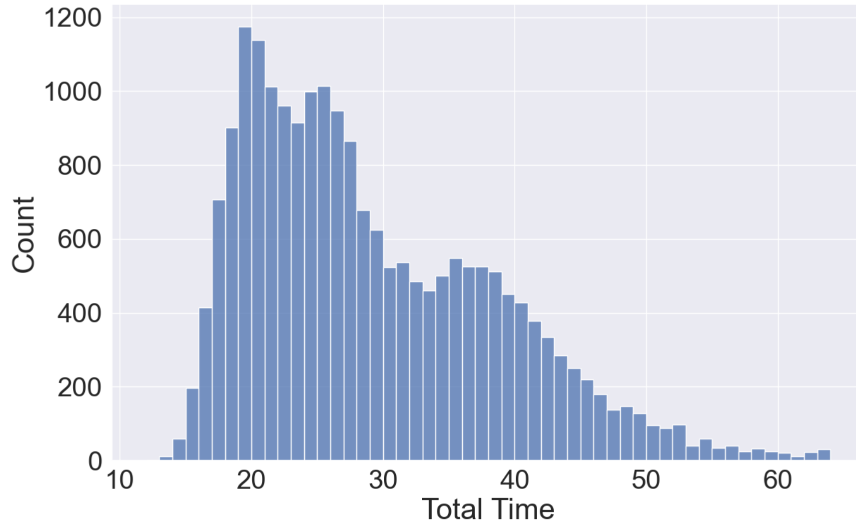

Figure 55: PubSub 1-to-1 db (v05)

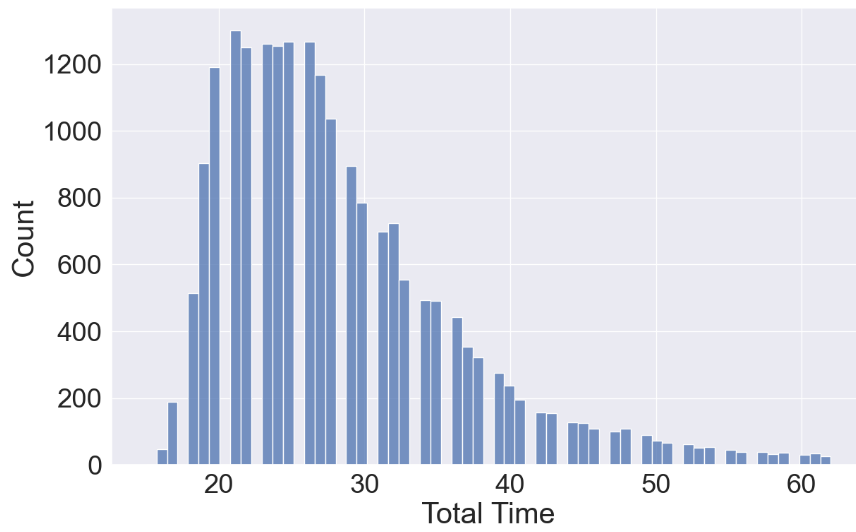

Figure 56: PubSub one db (v06)

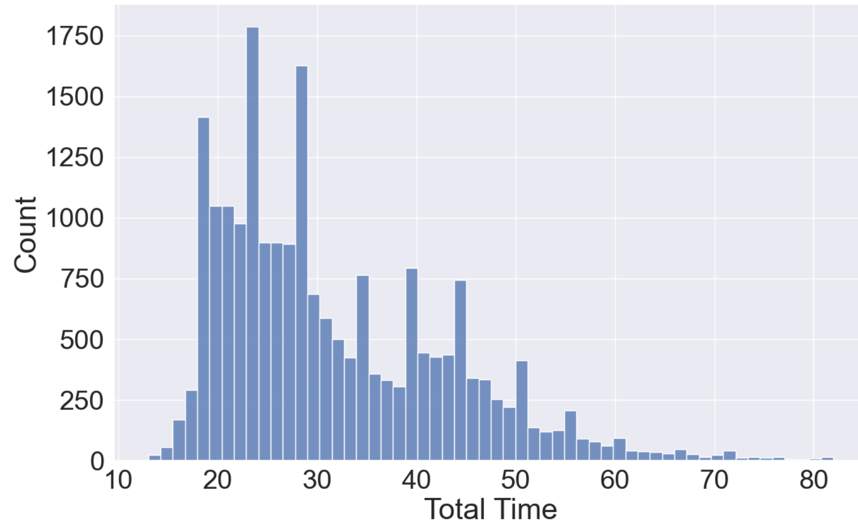

Figure 57: MessageBus 1-to-1 db (v07)

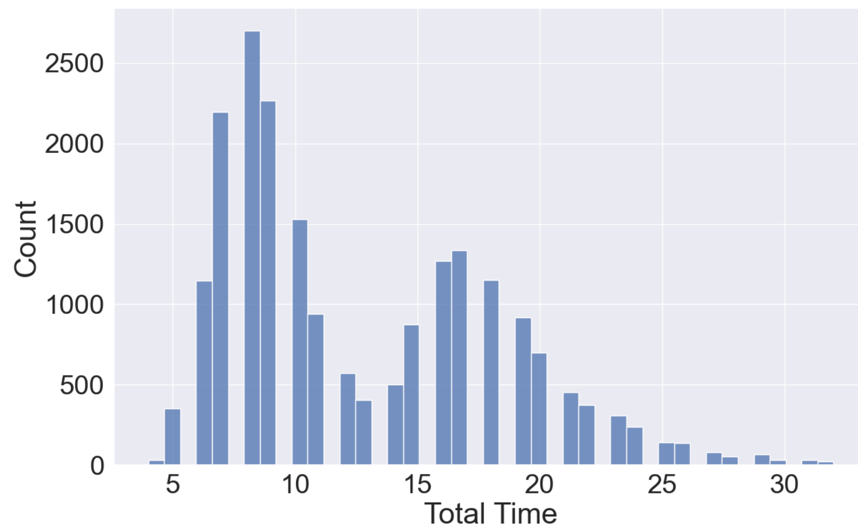

Figure 58: Orchestrate ApiGateway 1-to-1 db (v09)

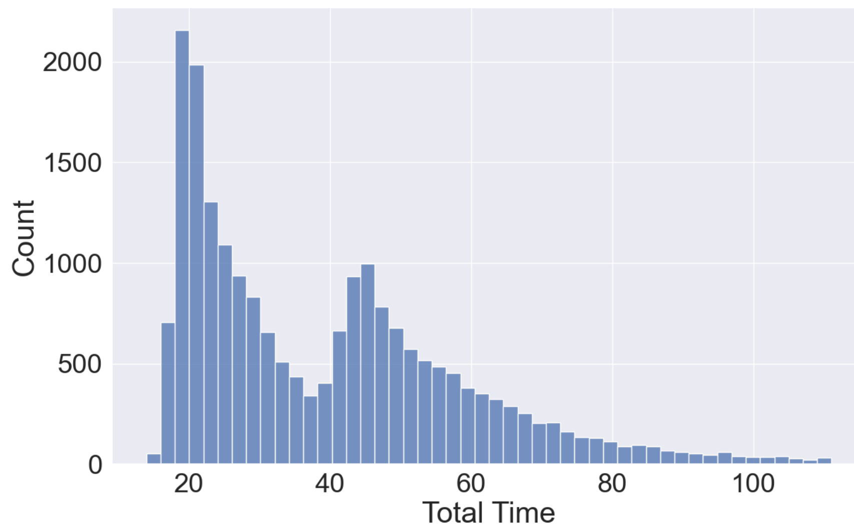

Figure 59: Orchestrate PubSub 1-to-1 db (v10)

Given the obtained data and the desired outcome the most promising approach that is able to deliver information about the individual performance/impact of the microservice patterns seems to be building a linear model by means of linear regression. In this case, the dependent variable is the runtime while the patterns describing every implementation are the independent variables. The variables have been divided into four major groups as follows:

- Communication between services (CBS) : RESTful, Database, Messaging
- API-Gateway (APG) : True , False
- Communication between the client and the application (CBCA) : Syncronus, Asyncronus
- Communication between the API-Gateway and the other services (CBAPGS) : RESTful, Messaging

Table 3 contains the encoding of the implementations by means of one hot encoding which denotes the existence of a feature (vari-

able) with the number one and the lack of the feature with the number zero. Hence the rows of the table capture the existence (i.e. where the number 1 is present) of a given pattern as well as implicitly the absence (i.e where the number 0 is present) of the pattern overall existing versions. By viewing the columns we can determine which patterns are included in every individual implementation. Thus the encoding for v01 (master-1-to-1-db) is 10001000 having as set variables CBS-REST (i.e. communication between services via REST) and CBCA-SYNC (i.e the communication between the client and the application is synchronous). At the same time, we can see that the CBS-REST pattern has been used in four different implementations (i.e. v01, v03, v05, v09).

For the calculation of the Ordinary Least Square (OLS) model, a data frame has been used that contained 187225 rows and 9 columns. The rows represent the number of data points acquired during the measurement faze which have been further processed by removing all the values that are three standard deviations away from the mean within every version. The same data has been used for the calculation of the boxplots seen in Figure 50. The columns in the data frame represent the eight used patterns plus the runtime which corresponds to the ELAPSED-TIME row from Table 3. The entire code is hosted within a public Github project [23].

Variables \ Version	v01	v02	v03	v04	v05	v06	v07	v09	v10
CBS-REST	1	0	1	0	1	0	0	1	0
CBS-DB	0	1	0	1	0	1	0	0	0
CBS-MESSAGING	0	0	0	0	0	0	1	0	1
APG	0	0	1	1	1	1	1	1	1
CBCA-SYNC	1	1	1	1	0	0	0	1	0
CBCA-ASYNC	0	0	0	0	1	1	1	0	1
CBAPGS-REST	0	0	1	1	0	0	0	1	0
CBAPGS-MESSAGING	0	0	0	0	1	1	1	0	1
ELAPSED-TIME	-	-	-	-	-	-	-	-	-

Table 3: One hot encoded model description

Dep. Variable:	elapsed	**R-squared:**	0.585
Model:	OLS	**Adj. R-squared:**	0.585
Method:	Least Squares	**F-statistic:**	3.764e+04
Date:	Mon, 11 Jan 2021	**Prob (F-statistic):**	0.00
Time:	20:33:05	**Log-Likelihood:**	-6.8676e+05
No. Observations:	187225	**AIC:**	1.374e+06
Df Residuals:	187217	**BIC:**	1.374e+06
Df Model:	7		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
const	-6.606e+09	2.2e+11	-0.030	0.976	-4.38e+11	4.25e+11
cbs_rest	-1.01e+10	2.15e+11	-0.047	0.963	-4.32e+11	4.12e+11
cbs_db	-1.01e+10	2.15e+11	-0.047	0.963	-4.32e+11	4.12e+11
cbs_messaging	-1.01e+10	2.15e+11	-0.047	0.963	-4.32e+11	4.12e+11
apg	8.335e+11	5.82e+11	1.432	0.152	-3.07e+11	1.97e+12
cbca_sync	1.67e+10	4.57e+10	0.365	0.715	-7.29e+10	1.06e+11
cbca_async	-3.912e+11	2.8e+11	-1.397	0.162	-9.4e+11	1.57e+11
cbapgs_rest	-8.335e+11	5.82e+11	-1.432	0.152	-1.97e+12	3.07e+11
cbapgs_messaging	-4.255e+11	3.05e+11	-1.395	0.163	-1.02e+12	1.72e+11

Omnibus:	72005.671	**Durbin-Watson:**	1.651
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	482917.044
Skew:	1.704	**Prob(JB):**	0.00
Kurtosis:	10.091	**Cond. No.**	1.13e+15

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The smallest eigenvalue is 4.19e-25. This might indicate that there are strong multicollinearity problems or that the design matrix is singular.

Figure 60: OLS Output

Figure 60 contains the summary of the OLS computation. Based on the value of R-squared it can be concluded that approximately 58.5% of the variance of the data can be explained by the model. Because the value of Prob (F-statistic) is zero it can be concluded that the independent variables are able to predict the dependent variable (i.e. the runtime). The coefficients with the highest significance (lowest P>|t| value) are: the *apg* and the *cbapgs_rest* followed by *cbapgs_messaging* and *cbca_async*; next is the *cbca_sync*

and lastly the *cbs_ ** coefficients have the lowest significance.

The communication between services (*cbs_ **) has the least overall impact on systems architecture. All three options reduce the runtime by the same amount. Choosing one over another will not improve or worsen the runtime behavior. Thus focusing on the communication between services for improving the overall runtime should be considered only after all options have been exhausted. As expected the introduction of the Api-Gateway does increase the overall runtime of the system. Because the *apg* is a standalone pattern and it does not have a counterpart it is harder to determine how it would impact the system if it would be left out. But it can be observed that it is comparable to the *cbapgs_ rest* which would decrease the runtime by the same amount as the *apg* would increase it. The communication between the client and the application (*cbca_ **) from a runtime perspective does favor the asynchronous which outperforms the synchronous one. Lastly the communication between the API-Gateway and the backend (*cbapgs_ **) provides a higher benefit when using synchronous information exchange in comparison to asynchronous communication.

The here constructed OLS model has not been build with the purpose of predicting the system's runtime but rather to determine how the individual patterns impact the runtime of the existing implementations. The model is complementary to the boxplot that provides a comparison of the versions as a whole while the model tries to provide a finer-grained view of the patterns and go beyond just measuring different aggregates of patterns.

8 Conclusion

8.1 Summary

The thesis presents a roadmap that addresses **RQ1** and can be used to make the transition from a monolithic architecture to a microservice architecture in a streamlined manner. The journey begins with identifying the underlying use cases of the application, how they interact which each other, and their boundaries. The next step

consists of determining how good is the existing code base managing to mirror the previously identified use cases and their boundaries. At this point, the objective is to determine what actions can be taken towards achieving the previously mentioned modular monolith. The steps should be small and incremental thus ever slightly improving the system without rendering it non-functional.

After managing to achieve a microservice ready module composition it is time to determine which microservice patterns are better suited to fulfill the existing and future requirements of the application. The next step is to implement parts or the entire system based on the microservice architecture and the previously selected patterns. The implementation should not be production-ready but rather a proof of concept thus reducing the needed resources to accomplish these tasks and making it more feasible. The next iteration consists of putting all the available implementations to the test and determining how they perform. The metric chosen in the theses was the time needed for a use case to complete a request received from a client. By making use of visual representations of the data like distribution- and boxplot-diagrams it is already possible to determine which implementations have the biggest potential and which ones are underperforming. The information provided by the data is of every implementation as a whole thus it is not really possible to determine the impact of the individual patterns within the implementation. To overcome this shortcoming the next stage foresees building a model by means of multilinear regression which should be able to provide a more fine-grained understanding of the individual patterns. With such a model it should be possible to determine how a pattern performs in comparison to another and evaluate its impact on the system. Thus the architecture can be tailored for the specific needs of the application. The above-mentioned procedure offers an answer for **RQ2** that goes beyond an academic solution and offers a more practical resolution based on empirical data. Additional benefits of applying this approach are: the team can get first-hand experience with the microservice architecture and the additional tools; the team can evaluate itself and determine if they are

up to the task, what are their shortcomings, and the challenges they will have to face going further.

Integrating the measurements within a deployment pipeline alongside other test suits (e.g., integration tests) could further strengthen the confidence of the team within their development cycle.

The runtime measurements and the resulting boxplot and OLS model are tailored to fit the existing implementations. They are not meant as a generic solution for every possible microservice implementation in the wild. The objective is to help newcomers to the microservice architecture to make the transition and provide a means of quality assurance along the way. In a real-world scenario, the measurement setup, as well as the OLS model, may wary based on the architecture, the requirements, and the key performance indicators which are deemed relevant for the development team. Nonetheless, the provided guide can act as a starting point towards the evaluation of an existing microservice implementation.

8.2 Threads of Validity

The implementations and the data generated in this thesis are based on a constructed example of an application that has been deployed in a test environment. It does not correspond to a real-life application thus a series of caveats apply. These caveats are:

- There were no explicit requirements for the system as a whole or for the individual use cases.
- The hardware on which all the services have been deployed consisted of five VMs that were not configured to provide production-ready support. The used VMs had an identical configuration running Ubuntu 18.04.5 LTS as the operating system, with two CPUs at 2.20GHz and 11GB of memory each.
- The Client as well as all the other services ran in the same network thus although the application itself was distributed it did not simulate the latency encountered when using the public internet.

Given these caveats, it was decided to use the runtime as a means of comparing the different implementations and the patterns of which they are comprised. Usually, a system has a specified load it has to handle as well as a response time for every incoming request. Having these parameters it is possible to configure the performance measurements accordingly and determine if the requirements are being met. Due to the lack of such clear requirements regarding load and runtime and the absence of any best practices or standardized reference values an exploratory approach has been taken. Thus a range of measurements has been performed whit arbitrary JMeter configurations testing different loads. Having the possibility to monitor the VM resource consumption (CPU and RAM usage) it became evident that some of the configurations were pushing the VMs at their limit. This overload was prominent in the VM hosting the Api-Gateway which became a bottleneck. The worst runtimes could be observed in the implementation versions with an Api-Gateway and where a callback address was used. In these versions, the Api-Gateway had to handle incoming requests from the Client, send requests to the backend, process the responses from the backend, and pushing the final response to the Client. As a result, the distribution plots were not normally distributed but rather exhibited three distinct columns having two at the edges and one in the middle. Thus if the load became too high we were testing the limitations of the hardware and not so much of the patterns. To eliminate the hardware limitations out of the equation the load has been reduced by continuously adapting measurement configuration until the VM's resource consumption stopped reaching 100% load. As a consequence, the runtime distributions among the different versions started having a similar shape thus allowing us to better compare the implementations and their patterns via OLS modeling.

8.3 Future Work

The runtime measurements and the statistical model have been devised based on an exemplary application with limited requirements. It would be interesting to test the proposed solution on a real-world

example. This would certainly offer more insight into how to set up the runtime measurements as well as identifying additional potential parameters that could be included in the statistical model thus increasing its significance. Figuring out how to automate the process as much as possible and integrating it in a deployment pipeline.

List of Figures

1 Use Cases CoCoME [16], Fig.4. An overview of all considered use cases of the Trading System. 13
2 Data Model CoCoME [16], Fig.8. The data model of the TradingSystem. 14
3 Master with one DB per service 21
4 Master with one common DB 22
5 Api-Gateway with one DB per service 22
6 Api-Gateway with one common DB 23
7 Pub-Sub with one DB per service 23
8 Pub-Sub with one common DB 24
9 Message bus with one DB per service 24
10 Api-Gateway orchestration with one DB per service 25
11 Pub-Sub orchestration with one DB per service . . . 25
12 UC 5 sequence diagram for version v01 and v02 . . . 26
13 UC 5 sequence diagram for version v03, v04 and v09 27
14 UC 5 sequence diagram for version v05, v06, v07, v10 28
15 UC 7 sequence diagram for version v01 and v02 . . . 29
16 UC 7 sequence diagram for version v03, v04 and v09 30
17 UC 7 sequence diagram for version v05, v06, v07, v10 31
18 UC 3 sequence diagram for version v01 and v02 . . . 32
19 UC 3 sequence diagram for version v03, v04 and v09 33
20 UC 3 sequence diagram for version v05, v06, v07, v10 34
21 UC 4 sequence diagram for version v01 35
22 UC 4 sequence diagram for version v02 36
23 UC 4 sequence diagram for version v03 36
24 UC 4 sequence diagram for version v04 37
25 UC 4 sequence diagram for version v05 38
26 UC 4 sequence diagram for version v06 39
27 UC 4 sequence diagram for version v07 40
28 UC 4 sequence diagram for version v09 42
29 UC 4 sequence diagram for version v10 43
30 UC 6 sequence diagram for version v01 46
31 UC 6 sequence diagram for version v02 47

32 UC 6 sequence diagram for version v03 47
33 UC 6 sequence diagram for version v04 48
34 UC 6 sequence diagram for version v05 48
35 UC 6 sequence diagram for version v06 49
36 UC 6 sequence diagram for version v07 50
37 UC 6 sequence diagram for version v09 51
38 UC 6 sequence diagram for version v10 52
39 Master with one DB per service 54
40 Master with one common DB 54
41 Api-Gateway with one DB per service 55
42 Api-Gateway with one common DB 55
43 Pub-Sub with one DB per service 56
44 Pub-Sub with one common DB 56
45 Message bus with one DB per service 57
46 Api-Gateway orchestration with one DB per service 57
47 Pub-Sub orchestration with one DB per service . . . 58
48 JMeter test plan 59
49 JMeter measurement execution 60
50 Box Plot 62
51 Master 1-to-1 db (v01) 66
52 Master one db (v02) 66
53 ApiGateway 1-to-1 db (v03) 67
54 ApiGateway one db (v04) 67
55 PubSub 1-to-1 db (v05) 68
56 PubSub one db (v06) 68
57 MessageBus 1-to-1 db (v07) 69
58 Orchestrate ApiGateway 1-to-1 db (v09) 69
59 Orchestrate PubSub 1-to-1 db (v10) 70
60 OLS Output 72

List of Tables

1 Implementations versions 61
2 Boxplot Values 61
3 One hot encoded model description 71

References

[1] Alliance, A. Manifesto for agile software development. https://www.agilealliance.org/agile101/the-agile-manifesto/, 2020. [Online; accessed 24-March-2021].

[2] Amazon. What is devops? https://aws.amazon.com/devops/what-is-devops/, 2021. [Online; accessed 24-March-2021].

[3] Baresi, L., Garriga, M., and Renzis, A. D. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing*. Springer International Publishing, 2017, pp. 19–33.

[4] Beck, K. https://twitter.com/KentBeck/status/250733358307500032?ref_src=twsrc%5Etfw, September 2012. [Twitter Post].

[5] deRoss, D. Acid versus base data stores. https://www.dummies.com/programming/big-data/hadoop/acid-versus-base-data-stores/, 2021. [Online; accessed 24-March-2021].

[6] Education, I. C. Soa (service-oriented architecture). https://www.ibm.com/cloud/learn/soa, 2019. [Online; accessed 24-March-2021].

[7] Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[8] Evers, M., and Westgeest, R. Responsibility driven design with mock objects. http://www.methodsandtools.com/archive/archive.php?id=90, 2009. [Online; accessed 24-March-2021].

[9] Faura, M. V. Orchestration of microservices, services - humans and robots. https://www.bonitasoft.com/news/orchestration-microservices-services-humans-robots, Feb 2020. [Online; accessed 24-March-2021].

[10] Fowler, M. Monolithfirst. https://martinfowler.com/bliki/MonolithFirst.html, June 2015. [Online; accessed 24-March-2021].

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[12] Gnatyk, R. Microservices vs monolith: which architecture is the best choice for your business? https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/, 2018. [Online; accessed 24-March-2021].

[13] Group, T. O. Service oriented architecture : What is soa? https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm, 2016. [Online; accessed 24-March-2021].

[14] Haddad, E. Service-oriented architecture: Scaling the uber engineering codebase as we grow. https://eng.uber.com/service-oriented-architecture/, 2015. [Online; accessed 24-March-2021].

[15] Hassan, S., Ali, N., and Bahsoon, R. Microservice ambients: An architectural meta-modelling approach for microservice granularity. In *2017 IEEE International Conference on Software Architecture (ICSA)* (2017), pp. 1–10.

[16] Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., Krogmann, K., Koziolek, H., Mirandola, R., Hummel, B., Meisinger, M., and Pfaller, C. *CoCoME - The Common Component Modeling Example*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 16–53.

[17] HOHPE, G., AND WOOLF, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.

[18] ISMAIL, K. 7 tech giants embracing microservices. `https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices/`, 2018. [Online; accessed 24-March-2021].

[19] KNORR, E. What ebay looks like under the hood. `https://www.infoworld.com/article/3041064/what-ebay-looks-like-under-the-hood.html`, 2016. [Online; accessed 24-March-2021].

[20] KOFLER, P., AND RIEGLER, G. Levels of modularity. `https://gregorriegler.com/2020/08/08/levels-of-modularity.html`, Aug 2020. [Online; accessed 24-March-2021].

[21] KOVACS, G. micro-cocome. `https://github.com/pufarin/micro-cocome`, 2019. [Online; accessed 24-March-2021].

[22] KOVACS, G. mono-cocome. `https://github.com/pufarin/mono-cocome`, 2019. [Online; accessed 24-March-2021].

[23] KOVACS, G. micro-cocome-statistics. `https://github.com/pufarin/micro-cocome-statistics`, 2020. [Online; accessed 24-March-2021].

[24] LEVCOVITZ, A., TERRA, R., AND VALENTE, M. T. Towards a technique for extracting microservices from monolithic enterprise systems, 2016.

[25] LEWIS, J., AND FOWLER, M. Microservices. `https://martinfowler.com/articles/microservices.html`, 2014. [Online; accessed 24-March-2021].

[26] MARTIN, ROBERT, C. The principles of ood. `http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`, 2005. [Online; accessed 24-March-2021].

[27] MICHAELS, G. How to deal with technical debt: Tips and strategies. https://unito.io/blog/technical-debt/, 2020. [Online; accessed 24-March-2021].

[28] MUELLER, E. What is devops? https://theagileadmin.com/what-is-devops/, 2019. [Online; accessed 24-March-2021].

[29] O'RIORDAN, M. Publish-subscribe: Introduction to scalable messaging. https://thenewstack.io/publish-subscribe-introduction-to-scalable-messaging/, 2020. [Online; accessed 24-March-2021].

[30] OSSES, F., MÁRQUEZ, G., AND ASTUDILLO, H. Exploration of academic and industrial evidence about architectural tactics and patterns in microservices. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (May 2018), ACM.

[31] PANYAM, S. Orchestration – a symphony of microservices. https://www.comakeit.com/blog/microservices-orchestration-symphony/, 2020. [Online; accessed 24-March-2021].

[32] RICHARDSON, C. *Microservice Patterns: with examples in Java*. Manning, 2019.

[33] RICHARDSON, C. Pattern: Api gateway / backends for frontends. https://microservices.io/patterns/apigateway.html, 2020. [Online; accessed 24-March-2021].

[34] RICHARDSON, C. Pattern: Database per service. https://microservices.io/patterns/data/database-per-service.html, 2020. [Online; accessed 24-March-2021].

[35] RICHARDSON, C. A pattern language for microservices. https://microservices.io/patterns/, 2020. [Online; accessed 24-March-2021].

[36] Richardson, C. Pattern: Shared database. https://microservices.io/patterns/data/shared-database.html, 2020. [Online; accessed 24-March-2021].

[37] Richardson, C. What are microservices? https://microservices.io/, 2020. [Online; accessed 24-March-2021].

[38] Riggins, J. Effective microservices architecture with event-driven design. https://thenewstack.io/event-driven-design-will-drive-microservices-clarity/, 2017. [Online; accessed 24-March-2021].

[39] Taibi, D., Lenarduzzi, V., and Pahl, C. Architectural patterns for microservices: A systematic mapping study. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science* (2018), SCITEPRESS - Science and Technology Publications.

[40] Zdun, U., Navarro, E., and Leymann, F. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *Service-Oriented Computing*. Springer International Publishing, 2017, pp. 411–429.

A Zusammenfasung

Im letzten Jahrzehnt hat die Microservice-Architektur eine zunehmende Akzeptanz bei großen Tech-Giganten (z.B. amazon, Netflix) gestoßen als auch bei kleineren Unternehmen, die von den versprochenen Vorteilen profitieren wollen. Während viel Arbeit geleistet wurde, um die Architektur und ihre Muster zu beschreiben, gibt es noch wenig bis gar keine Arbeit darüber, wie man den Übergang von einer monolithischen zu einer Microservices-Architektur leisten soll. Dies ist besonders für kleine und mittlere Unternehmen eine Herausforderung, die nicht über die scheinbar endlosen Ressourcen großer Unternehmen verfügen. Diese Arbeit beschreibt eine Roadmap, die es einem Team ermöglicht, neues Know-how zu erwerben und zu erweitern, das Team dabei unterstützt, die beste architektonische Entscheidung in Übereinstimmung mit ihren Bedürfnissen zu treffen und das implementierte System kontinuierlich zu evaluieren. Die vorgeschlagene Lösung ist kostengünstig, liefert nach jedem Schritt Ergebnisse und schafft schließlich eine Arbeitsumgebung, die Veränderung und architektonische Evolution fördert.