INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Análise de Impacto do GraphQL Federation

JOÃO FILIPE PEREIRA FONTÃO
Junho de 2022

POLITÉCNICO
DO PORTO

GraphQL Federation Impact Analysis

João Fontão

Dissertation to obtain a Master's Degree in Informatics Engineering
Specialisation Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, June 29, 2022

Abstract

Microservices architectures have gained more admirers due to the problems they came to solve from previous architectures, which led many companies to adopt this architecture. When this architecture emerged, REST was one of the leading architectural styles used with microservices. However, with the growth of microservices, it was noticed that it presents some problems when there is a significant increase in the complexity of applications.

Due to the problems found with the existing approaches, it was tried to find solutions that could help solve these problems, and it was then that the company Facebook launched GraphQL as one of the most promising alternatives. Although GraphQL is recent, many studies have already compared it with other approaches such as REST.

GraphQL then began to become increasingly popular, and although there is already some documentation and studies, there are still topics that have not been adequately explored. For this reason, the focus of this thesis is to explore the impact that GraphQL has on quality attributes in a microservices architecture, focusing on a new approach, GraphQL federation. There is still little documentation on this approach, and the impact of its implementation is not very well known, but despite being recent, it has already been implemented by large companies such as Netflix, PayPal and GitHub.

This dissertation also provides an insight into the current state of GraphQL in a microservices architecture, including the benefits, problems, success stories, some patterns, and a more detailed explanation of the GraphQL federation architecture. A proof of concept was developed by implementing GraphQL federation to a solution that already used GraphQL with microservices. Afterward, the solutions were evaluated in terms of performance and maintainability, and a comparison was later made between the two projects to see if federation impacted these quality attributes.

One of the conclusions of this study is that the GraphQL federation solution has better maintainability, taking into account the qualitative assessment. However, in terms of performance, the solutions do not show significant differences, presenting both good performance results.

Keywords: microservices, GraphQL, quality attributes, federation.

Resumo

As arquiteturas de microsserviços têm conquistado mais admiradores devido aos problemas que vieram resolver das arquiteturas anteriores, o que levou um grande número de empresas a adotar esta arquitetura. Quando esta arquitetura surgiu, REST era um dos principais estilos de arquitetónicos usados com microsserviços. Porém, com o crescimento de microsserviços percebeu-se que este apresenta alguns problemas quando existe um crescimento significativo das aplicações.

Devido aos problemas encontrados com as abordagens existentes, procurou-se encontrar soluções que pudessem ajudar a resolver esses problemas, e foi então que a empresa Facebook lançou o GraphQL como uma das alternativas mais promissoras. Embora o GraphQL seja recente, muitos estudos já o compararam com outras abordagens, como REST.

GraphQL começou então a tornar-se cada vez mais popular, e apesar de já existir alguma documentação e estudos, ainda existem tópicos que não foram devidamente explorados. Por esta razão, o foco desta tese é explorar o impacto que o GraphQL tem nos atributos de qualidade numa arquitetura de microsserviços, focando numa nova abordagem, GraphQL federation. Ainda há pouca documentação sobre esta abordagem, e o impacto da sua implementação não é muito conhecida, mas apesar de recente, já foi implementado por grandes empresas como Netflix, PayPal e GitHub.

Esta dissertação também fornece uma visão do estado atual do GraphQL numa arquitetura de microsserviços, incluindo os benefícios, problemas, casos de sucesso, alguns padrões e uma explicação mais detalhada da arquitetura de GraphQL federation. Uma prova de conceito foi desenvolvida com a implementação de GraphQL federation numa solução que já usava GraphQL com microsserviços. Depois foi realizada uma avaliação das soluções em termos de desempenho e de manutabilidade e foi posteriormente feita uma comparação entre as duas soluções para se verificar se federation tem impacto nestes atributos de qualidade.

Uma das conclusões deste estudo é que a solução de GraphQL federation apresenta uma melhor manutabilidade, tendo em consideração a avaliação qualitativa. No entanto, relativamente ao desempenho as soluções não apresentam diferenças significativas, apresentando ambas bons resultados de desempenho.

Palavras-chave: microsserviços, GraphQL, atributos de qualidade, federação

Acknowledgement

This thesis would not have been possible without the cooperation of all those who were, directly and indirectly, involved in this whole process, so I would like to thank all these people.

I want to thank my supervisor, Isabel Azevedo, for all the support provided and for all the motivation transmitted. Thank you also for your patience, availability, and knowledge transmitted that made this dissertation a better work.

I also want to thank my family and friends who have always supported me and given me strength in this challenging phase, never doubting my abilities.

To my girlfriend, who has been my biggest support throughout this journey. Thank you for always believing in me and giving me the strength to finish another stage of my academic life.

Finally, I would like to thank all the professors and colleagues who helped me along this academic path at ISEP for helping me to achieve academic and professional success.

Contents

List of Figures

List of Tables

List of Source Code

List of Abbreviations

AHP	Analytic Hierarchy Process.
API	Application Programming Interface.
BFF	Backends For Frontends.
CRUD	Create, Read, Update and Delete.
DGS	Domain Graph Service.
DSR	Design Science Reasearch.
FAST	Function Analysis System Technique.
FEI	Front End of Innovation.
GQM	Goal Question Metric.
HTTP	Hypertext Transfer Protocol.
ISEP	Instituto Superior de Engenharia do Porto.
JSON	JavaScript Object Notation.
MSA	Microservices Architecture.
NCD	New Concept Development.
NPD	New Product Development.
QA	Quality Attributes.
QFD	Quality Function Deployment.
REST	Representational State Transfer.
SDL	Schema Definition Language.
SOA	Service-Oriented Architecture.
SOAP	Simple Object Access Protocol.
SQuaRE	System and Software Quality Assessment and Requirement.
TOPSIS	Technique for Order of Preference by Similarity to Ideal Solution.

WAN Wide Area Network.

Chapter 1

Introduction

This chapter introduces the thesis project developed during the Master's Degree in Software Engineering at Instituto Superior de Engenharia do Porto (ISEP). Initially, it focuses on a brief contextualization and definition of the problem studied, and then the main objectives and research methodologies applied throughout the project are presented. Finally, there is a description of the general structure of the document.

1.1 Context

In recent years, there has been an increase in the use of microservices as monolithic applications had disadvantages when they had significant growth. Microservices can be easier to maintain, more flexible, and present greater availability.

APIs are always implied when discussing web application development, which are "mechanisms that enable two software components to communicate with each other using a set of definitions and protocols" [1]. Over the years, Representational State Transfer (REST) and Simple Object Access Protocol (SOAP) have become the most known and used in almost all organizations to develop applications.

In 2015, Facebook decided to launch a new query language for APIs that they had been developing for some time, called GraphQL, presented as a query language that improves flexibility, performance and memory usage. One of the incredible novelties of this Application Programming Interface (API) compared to the others is that it was aimed at client applications, as they are the ones who define in the request only the data that they want to be returned, not being necessary to receive unnecessary information.

Over the last few years, many studies have been carried out on GraphQL regarding how to apply it, the main advantages and disadvantages, and mainly studies comparing the different APIs to understand which is the best to use and in which situations [2]. This work has a different context, as it does not intend to compare the various APIs but to focus only on the use of GraphQL with microservices, specifically on the use of the GraphQL federation approach.

1.2 Problem

A Microservices Architecture (MSA) brings some challenges, such as the performance of the client application due to the need to obtain responses from multiple services [3].

Thus, some companies have been trying to overcome these difficulties, namely fetching information from multiple downstream microservices with GraphQL [4]. It allows accessing all data from a single endpoint, reducing the resulting Hypertext Transfer Protocol (HTTP) overhead, helping to improve performance, and making code easier to maintain [5].

The GraphQL adoption with Service-Oriented Architecture (SOA) has been analyzed [4], but its use in microservices-based applications is recent. Extensive studies about different strategies and their benefits and downsides are scarce [6]. On the other side, there is still a lack of a systematic understanding of the Quality Attributes (QA) associated with MSA [7].

A new form of integration has recently appeared, GraphQL federation, which is an approach to consolidating many GraphQL API services into one [8]–[10]. This new approach is an alternative to having gateway-level integration, superior to stitching, a technique initially used [11].

1.3 Objectives

GraphQL has been increasingly used in microservices implementations. However, as previously mentioned, there are still few studies on approaches to implementing GraphQL in microservices architectures and even fewer how these approaches improve the application in terms of quality attributes.

The main objective of this work is to explore the impact that the use of GraphQL federation in an application with microservices architecture has on performance and maintainability.

To achieve this goal, a set of small steps will be followed:

- Studying the different approaches currently available to implement GraphQL with microservices.
- Understanding which quality attributes are the most problematic in each approach.
- Comparing the main approaches and see what improvements the GraphQL federation architecture has brought.
- Implement GraphQL federation in a solution.
- Analyzing the solutions in terms of performance and maintainability.

The quality attributes (performance and maintainability) according to which the solution will be evaluated were chosen after evaluating those that could be of most interest to study. The explanation of the choice of these two quality attributes can be found later in section 3.7.

1.4 Research Methodology

To achieve the objectives, it is necessary to define which research method is the most appropriate to follow because only then will it be possible to have a well-designed plan.

Question: What is the impact of GraphQL federation in terms of maintainability and performance in an application with a microservices architecture and with a reduced number of microservices?

Result: One solution, with the federation approach.

Approach: Design Science Reasearch.

Dissemination: Thesis and Project available in a Github repository [12].

As mentioned, the most appropriate approach to address this problem is the Design Science Reasearch (DSR), which is a "problem-solving paradigm that seeks to enhance human knowledge via the creation of innovative" [13]. DSR seeks to improve the knowledge bases of technology by creating innovative solutions that solve problems in a given environment.

This methodology consists of six main activities:

- **Problem identification and motivation:** define problem-specific research and justify the value of a solution. The problem is used to develop an artefact that can provide a solution. Explaining the value of a solution will motivate the researcher and research audience to develop a solution.

  The outcome produced is a narrative review of the literature to validate the identified problem. As the problem is still a recent issue, some grey literature was used to help the research (chapter 2 and chapter 3). Subsequently, the value analysis is developed to justify how this work adds value to the identified problem area (chapter 4).

- **Define the objectives for a solution:** infer the goals from the problem definition and knowledge of what exists. Objectives are rationally inferred from the problem specification. It is necessary to know the resources required, including knowledge of the problem's state and current solutions.

  The outcome at this stage is to define the main objectives, the main one being to reduce the knowledge gap that exists in the use of GraphQL with microservices, but specifically with the federation approach and how this approach affects the quality attributes (section 1.3). In order to carry out a good analysis of the objectives, it is necessary to understand the problem well, so for this outcome, it was necessary and fundamental to define the problem and the context of the work (section 1.1 and section 1.2).

- **Design and development:** create an artefact, which is often potentially constructs of models, methods, or instantiations. The resources needed to move from objectives to design and development include having a great deal of knowledge about the theory of the problem and then applying it to the solution.

  The outcome is a solution that implements GraphQL federation and explains the entire architecture of the solution (chapter 5). To develop the solution, it was decided to use a project that already implements GraphQL but without federation. To look for the initial project, projects from public repositories on GitHub were analyzed that implemented GraphQL and that the language was Java to use, a language that did not bring many difficulties due to time constraints. In order to choose the most suitable project, some criteria were taken into account, such as documentation, tests, complexity, and how recent the project was. For the design and implementation of the solution, it was decided to keep the initial functionalities, thus focusing the work on changing the approach to GraphQL federation (chapter 6).

- **Demonstration:** demonstrate the use of the artefact to show that it solves the problem, or part of the problem developed. It may be necessary to demonstrate its use through experimentation, simulation or other activities, and it is essential to know how the artefact solves the problem.

  The outcome is the final solution with federation and the comparison of the results obtained by evaluating the quality attributes (chapter 6, chapter 7 and chapter 8).

- **Evaluation:** observe and measure how the artefact supports a solution to the problem. This activity involves comparing the objectives of the solution with the actual results obtained

from demonstrating the artefact. This evaluation requires knowledge of relevant metrics and appropriate analysis techniques. Evaluation can be of different types depending on the solution and the problem identified.

The outcome produced is the definition of the approaches and methods followed to evaluate the results obtained (chapter 7). One of the central parts of this dissertation, where using the Goal Question Metric (GQM) method, the new and the old solution is evaluated, and it is noticed that changes have occurred in the defined quality attributes.

- **Communication:** communicate the problem, its importance, and its usefulness to relevant audiences. For academic publications, researchers can use the framework of this process to structure the article.

  The expected outcome is the master's thesis developed and the project made available in a public repository [12].

1.5 Document Structure

´

This first chapter ends with an explanation of the structure of this document, with a description of each chapter.

- **Introduction – Chapter 1:** gives a general view of the development objective of this thesis document. There will be a contextualization of the work, the presentation of the problem that motivated the developed project, an explanation of the objectives it intends to achieve with this work and the chosen research methodology.
- **Background – Chapter 2:** the two main concepts addressed throughout the document, GraphQL and microservices, are presented in a less detailed way. Regarding GraphQL, the main concepts that one must know to use the API are mentioned, while concerning microservices, the main characteristics are presented, such as the advantages and disadvantages.
- **State of the Art – Chapter 3:** this chapter is dedicated to providing an overview of currently existing studies on the application of GraphQL in a microservices architecture, an explanation of how to implement GraphQL federation, experiences of reputable companies in migrating to this approach, and examples of technologies used. Finally, it refers to the quality attributes used to evaluate the solution.
- **Value Analysis – Chapter 4:** includes an in-depth value analysis, analyzing the product value and applying the Function Analysis System Technique (FAST).
- **Design – Chapter 5:** this chapter includes a requirements analysis of the chosen project. In this chapter, there are details about functional and non-functional requirements. It also contains the design of the initial application and the solution to be implemented with federation.
- **Implementation - Chapter 6:** the implementation of the initial solution and the solution with GraphQL federation are explained in this chapter, explaining changes made from one project to the other and with some examples that help explain the implementation.
- **Tests and Solution Evaluation - Chapter 7:** define the hypothesis and objective of the evaluation. This chapter describes and explains the evaluation method to be applied. Finally,

the results obtained are presented, and an analysis of these results is explained to understand what differences in terms of quality attributes produce the new solution.

- **Conclusion - Chapter 8:** present the results and difficulties felt, explanation of what was achieved, the contributions that this dissertation offered, and the following steps to continue this project.

At the end of the document, some appendixes can provide more detail information about some work done during this project, namely:

- **Appendix A:** Maintainability Results.
- **Appendix B:** Performance Results.

Chapter 2

Background

This chapter intends to introduce some essential knowledge about GraphQL and microservices. Initially, the main elements of GraphQL will be presented, followed by the main benefits, challenges and characteristics associated with microservices.

2.1 GraphQL

GraphQL is "a query language for APIs and a runtime for fulfilling those queries with your existing data" [14].

GraphQL is a query language for APIs that allows reducing the number of requests needed to obtain the desired data, and it grants the components or users to define their data requests furthermore, the expected responses. As the name implies, they are hierarchical like a graph.

GraphQL allows clients to consume what the server allows, and these queries are made through the list of allowed operations. Each GraphQL specification defines a query language but does not define how the operations should be resolved, and this resolution will depend on how GraphQL is implemented. However, the implementations are generally a tiny layer on the server that receives the requests and calls the respective resolvers. A resolver is responsible for populating data in the developer schema.

It was developed on providing a query language API that prioritizes the client and allows the developer to define what data they want to query in the backend. It also aimed to request all data in just one request and return only the necessary fields, thus reducing unnecessary network traffic and server resource usage [11].

While the initial reference implementation of GraphQL was written in JavaScript, open-source libraries exist, currently, for almost every notable language used in web development - from Java to Scala and Clojure [5]. Like with all open-source libraries, the maturity of these projects may vary, but their availability enables viable implementation of a GraphQL server with almost any technology stack [15].

GraphQL best practices are presented in this section, and the concepts of schema, types and introspection are also explained below.

2.1.1 Best Pratices

Although GraphQL is recent, some good practices have already been established that help guide developers who want to implement GraphQL. According to these authors [16], [17], the purpose of this section will be to show brief descriptions of promising practices about GraphQL.

HTTP

GraphQL is used to build distributed systems, so it is necessary to discuss how data is sent from the client to the server and how the response is obtained.

Typically, GraphQL runtime is used, which handles protocol binding, serialization, and deserializations. Requests and responses are associated with the HTTP protocol and JavaScript Object Notation (JSON) data format.

On the server-side, GraphQL is typically served over HTTP via a single endpoint that expresses the entire feature set of the service. A GraphQL request can be mapped to HTTP requests with JSON data structure in two ways, in the form of an HTTP GET with query parameters or an HTTP POST with a JSON document.

GraphQL responses are mapped to HTTP responses with a JSON payload, whether the response returns an error or current data. A payload is sent with a JSON object as the data when no error occurs.

Versioning

Managing changes and evolution in software systems is never easy, but managing changes in loosely coupled distributed systems such as API solutions is difficult. Even a tiny change in the API is enough to break some of the clients that consume the API. Published APIs cannot be changed quickly. At the very least, the APIs need to remain backwards compatible so that old clients do not break and new clients can use the new and improved features.

As APIs evolve, it becomes a challenge to manage them, so they should ideally be created so that changes are virtually unnecessary and that any predictable changes can be made as compatible changes. When there is limited control over the data that is returned from an API endpoint, any change can be considered a significant change and the major change requires a new version. Adding new features to an API requires a new version. A trade-off arises between releasing frequently and having many incremental versions versus understanding and maintaining the API.

GraphQL only returns explicitly requested data, so new features can be added through new types and new fields on those types without creating a disruptive change. This peculiarity of GraphQL has led to a common practice of avoiding changes and serving an API without a version. While there is nothing to stop a GraphQL service from being versionable like any other API, GraphQL advises avoiding versioning by providing the tools for the continuous evolution of a GraphQL schema.

GraphQL supports building backwards compatible APIs, which means additional fields, additional types, queries, mutations, and signatures can be added. However, existing fields cannot be changed or removed, no fields can be removed from existing types, and no type, mutation, or signature queries can be removed.

An often raised concern of avoiding versioning is the need to deal with ever-increasing API responses as evolution allows adding but not removing fields. With GraphQL, the impact is much less as only

the schema size would increase, and it can be more challenging to find the right field needed for a given order. However, the actual size of the response would not increase as the response only contains the fields explicitly requested by the client.

Nullability

Most type systems that recognize "null" provide both the standard type and the null version of that type, so standard types do not include "null" unless explicitly declared. However, in GraphQL, each field is null by default. That is because many things can go wrong with a network service backed by databases and other services. A database might crash, an asynchronous action might fail, an exception might be thrown.

When deleting all override fields, any of these reasons could result in the field returning "null" instead of having a complete failure for the request. Instead, GraphQL provides non-null variants of types that guarantee clients that, if asked, the field will never return "null". Instead, the previous parent field will be "null" instead if an error occurs.

When designing a GraphQL schema, it is crucial to consider all the issues that can go wrong and whether "null" is an appropriate value for a failing field. Usually, it is, but occasionally it is not. In these cases, use non-null types to make this guarantee.

Pagination

The GraphQL type system allows some fields to return lists of values but leaves paging longer lists of values up to the API designer. Typically, GraphQL fields that can return long lists accept "first" and "after" arguments to allow specification of a specific region of a list, where "after" is a unique identifier for each of the values in the list.

There are a couple of conventions for handling long lists in GraphQL: Plurals, Slices, and Pagination.

Plurals: When modelling a one-to-many relationship in the schema, a field with a plural name is used, this is where plurals get their name, are realized as arrays, and plurals do not employ any form of pagination.

Slicing: is introduced to limit the returned data: retrieving only the first two/three/four or only the last two/three/four is an example of a slice of the data. This slicing approach can be used for paging to retrieve the first two, then the next two, and then the next two sliced.

Caching

GraphQL can be inefficient when connecting to the backend due to resolver structures because it is designed to allow to write clean code on the server, where each field in each type has a focused function to solve that value. However, a naive GraphQL service might repeatedly load databases without further consideration.

This problem is commonly resolved by a batching technique, where multiple requests for data from a backend are collected for a short period and then sent in a single request to an underlying database or microservice. Facebook has published its library of data loaders for this purpose, called DataLoader. It allows building a business layer that can handle caching.

2.1.2 Schema

GraphQL schema informs which queries, mutations, types, and directives exist on a server. These can also contain documentation in the form of comments to explain the function of a specific operation, which client applications can later consult. The schema is the essential part of GraphQL because, without it, clients would not know how to communicate with the server, nor would they be able to make requests.

In GraphQL, the schema has the particularity of extending other schemas that can come from any API. This schema extension is a characteristic that can be used to combine microservices with GraphQL, making its implementation easier, as it can combine schemas of different services.

Being such an essential part of implementing a solution with GraphQL, defining the schema is one of the first steps. Due to its importance, its design must be done carefully and well structured, as a poorly designed schema can cause future difficulties when of implementation. Figure 2.1 shows an example of a GraphQL schema.

```
schema {
  query: Query
  mutation: Mutation
}
```

Figure 2.1: Schema GraphQL [18]

GraphQL currently supports three types of operations: queries, mutations and subscriptions [5]. All these operations are defined similarly, entering the type of operation and its name. All GraphQL servers have at least one query and may or may not have mutations and subscriptions.

Queries are a GraphQL root type, which maps the available operations to get the necessary data. They describe the data requested by a client application to a GraphQL server, where the fields that the client wants to be returned in the server's response are also specified. GraphQL has this advantage: the client application can define the fields it wants, not needing to receive data it does not need.

If an error occurs in the query, the key "error" will be returned, so the client realizes that something unexpected occurred, and when everything goes as planned, the key "data" is returned. Figure 2.2 shows an example of a GraphQL query and figure 2.3 shows an example of a result of a GraphQL query.

Despite this simple explanation and example of what queries are, they can be very complicated to write for inexperienced users, so those developing applications using GraphQL may encounter some challenges [19].

```
query {
  hero {
    name
  }
  droid(id: "2000") {
    name
  }
}
```

Figure 2.2: Query GraphQL [18]

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    },
    "droid": {
      "name": "C-3PO"
    }
  }
}
```

Figure 2.3: Query Result GraphQL [18]

Subscriptions are very similar to queries, but they are based on events and are usually implemented through web sockets.

Mutations, unlike queries, are used to create or modify data, following the types described in the schema. However, like queries, mutations can return response objects, and the client can define the fields it wants to receive. This mutations functionality is handy when making object changes, so the clients' applications can check if the operation went as expected. Figure 2.4 shows an example of a GraphQL mutation and figure 2.5 shows an example of a result of a GraphQL mutation.

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

Figure 2.4: Mutation GraphQL [18]

Figure 2.5: Mutation Result GraphQL [18]

The GraphQL Schema also allows directives, which are basically used to annotate parts of a GraphQL document, to say how they should be evaluated differently, basically directives to perform custom logic as appropriate. Directives should only be used where they are declared. In the schema, the values of the fields in GraphQL are nulls by default. It is necessary to specify that the given field is mandatory for this not to happen. To mark a field as mandatory, the following character "!" must be placed in front of the field.

It is possible to know the queries that are supported through introspection. Introspection about the schema is one of the features that highlight GraphQL, and this means that it can do queries on the schema, thus being possible to know the fields and types of a schema. Introspection has a significant advantage in exposing the API schema in an easily readable format.

This feature is handy for client applications that make calls to the GraphQL server to know which methods are available, which fields are needed, and how to make the respective calls. Introspection has resources like the "Schema" type, which contains all the types and directives of a server, and "Type", which is used to know the information about a specific field.

2.1.3 Types

GraphQL is strong typing, and for that reason, types are the fundamental basis of any GraphQL schema. Currently, in GraphQL, there are these six types: scalar, enumerator, union, interfaces, object, input object [5].

The most primitive is scalar, which represents a primitive value such as string or integer. In GraphQL, as in many languages, there are enumerators that are like scalar values, but they have defined the possible values that the field can take.

GraphQL also allows two abstract types, interface and unions. Declarative interfaces represent the list of fields and their arguments. Objects can implement these interfaces, which require the object to define all the interface fields. In GraphQL, an object can implement multiple interfaces. Unions represent an object that can be a list of other objects but provides guaranteed fields between those types. They also differ from interfaces in that objects define which interfaces they implement but do not know what unions they contain.

Queries are hierarchical in GraphQL, describing a tree of information. While scalar types describe the leaf values of these queries, objects are hierarchical levels. Thus, GraphQL objects represent a list of fields, each producing a specific type value.

The fields in GraphQL have the particularity that they can accept arguments to configure their behaviour. For this type of situation, there are InputObjects that are a set of input fields, and these fields can be scalar, enumerated or even InputObjects, which will customize a specific behaviour in a field.

GraphQL also allows lists with a particular collection type that declares the type of each element in the list. Lists are serialized as ordered lists. To represent a list of a type in GraphQL is used "[]".

2.2 Microservices

Microservices architecture is defined as: "a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists" [20].

Initially, monolithic applications were used where all the logic was in a single application and had the great advantage of simplicity. However, it entailed many disadvantages that made many companies change their monolithic architectures to a solution with microservices. This new architectural solution brought many advantages that made it very popular today.

Microservices architecture appeared some time ago as part of service-oriented software development, promoting high isolation and finding a better alternative to monolithic applications.

This type of architecture is known for dividing the business logic into small autonomous services, each responsible for implementing a part of the business logic, which different services or clients can later use.

This section will present the main features (section 2.2.2), advantages (section 2.2.3), disadvantages (section 2.2.4) and patterns (section 2.2.1).

2.2.1 Patterns

API Gateway

Microservices often provide the functionality to other services or clients through an API consumed. However, when developing an application based on microservices, it is customary to use an aggregation mechanism on the server-side for all microservices. In these situations, the API Gateway pattern (figure 2.6) appeared to provide a solution for this aggregation of services. The Gateway API pattern is defined as "the entry point of the system that routes the requests to the appropriate microservices, also invoking multiple microservices and aggregating results" [11].

The API Gateway refers to implementing a service on top of microservices to combine all the microservices functionality and expose them for client application consumption. API Gateway is the system entry point that forwards requests to the appropriate microservices, invoking various microservices and aggregating results. It may also be responsible for other tasks, such as authentication and monitoring.

API Gateway adds an abstraction layer between the clients and the server, and this allows the microservices to evolve without affecting the clients because it will only be necessary to make changes in API Gateway, and these changes will be reflected in all clients. In this pattern, there is a need to create an API for each client only to see the resources they need.

According to Davide Taibi [21], the API Gateway provides the following benefits:

- Ease of Extending to Other Clients: because API Gateway is customized, implementing new features is more effortless than other architectures.
- Market-centric Architecture: services can be easily modified based on customer needs, as only need to make changes to Gateway APIs without changing client applications.
- Backward Compatibility: the gateway guarantees for evolving services that interface endpoint changes do not hamper existing clients.

And some disadvantages:

- Potential Bottleneck: the gateway layer is the single point of access to all requests, which means that if not correctly designed and implemented, it could be a vulnerability because if something goes wrong, the entire application will stop working.
- Implementation Complexity: this new layer added will increase the complexity as it requires the implementation of several interfaces for each of the services.
- Scalability: when there is an explosion in the number of microservices, management becomes more complex, and changes may need to be made to handle the increase in data traffic better.

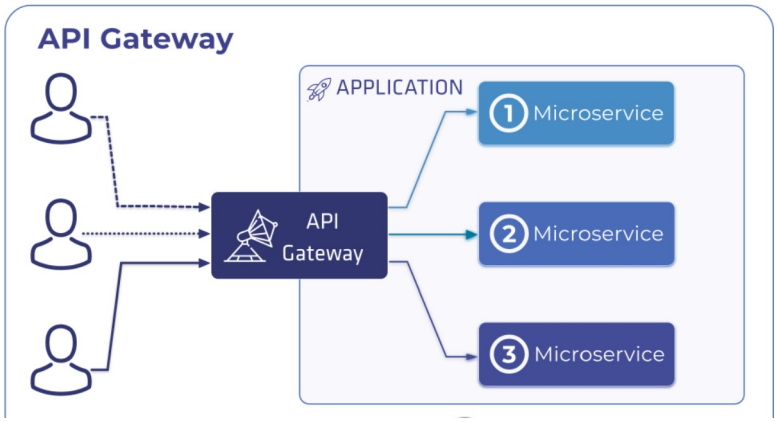

Figure 2.6: API Gateway Pattern [22]

Backends For Frontends

One of the problems encountered with microservices is that there is usually a giant layer for all services, and this will start to lose isolation in the various user interfaces, thus limiting freedom. One solution is to use a model in which backends are restricted to a specific interface [23].

Backends For Frontends (BFF) (figure 2.7) is a standard that is often associated with the development of microservices and is characterized by being an API placed on the server-side that contains all the logic provided by the backend application. The standard is applied to remove all business associated with the frontend, thus reducing responsibility complexity and thus making client applications more autonomous. BFFs make possible changes in applications less invasive, as they can be made by modifying only one side, thus reducing the coupling between the different layers.

Like API Gateway, it allows the aggregation of different microservices so that when a request is made via the frontend, it will fetch the information from different locations and aggregate the response for

the visualization in a single resource. It will respond to the requestor, allowing cancelling the multiple requests for the different endpoints and controlling the response sent, providing only the necessary data to the client.

This pattern allows the development team to focus on any user interface and handle server-side issues. The big problem with this approach is the same as in the other aggregation layers because it can assume logic that it should not. BFFs should only contain specific behaviours to deliver the data needed by a specific .

Using this standard will eliminate direct calls to microservices, thus allowing teams that build customer-facing applications to be responsible for their fate. BFF thus supports the evolutionary design, thus moving the entire system to a less coupled state and providing single-purpose APIs.

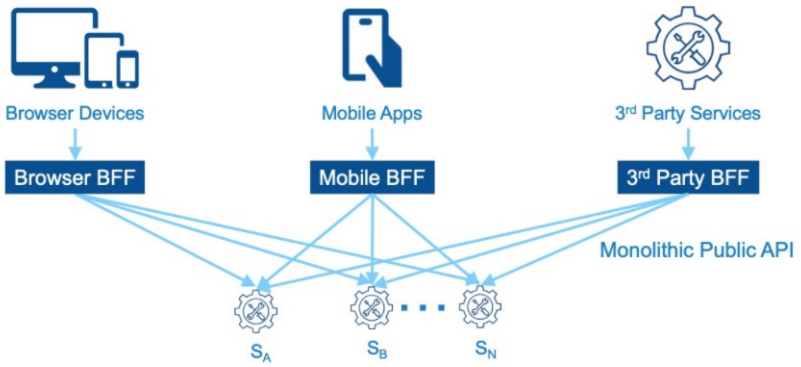

Figure 2.7: BFF Pattern [24]

2.2.2 Characteristics

Microservices (figure 2.8) have many characteristics that distinguish them from different architectures, making them stand out from the rest. In this subsection, according to [20], [23], [25] the main features are presented:

- Small, independent services that different teams can develop without dependency issues.
- Each service is a separate code managed by a small team and can be developed in any language independently of the language of the other services.
- Responsibility for persistence data, utilizing your database and your persistence layer that other services must not share. For this reason, each microservice manages its database without any dependence or concern for others, facilitating the use of different approaches in the same application.
- Specialized services, each microservice is passed to support a well-defined part of the business logic, so deployment is geared towards solving a specific problem.
- They are designed to have the ability to fail, as with so many microservices there can be some failure, so today's client applications have to be able to handle these situations, this being a relative disadvantage to previous approaches, for example monolithic, as it was much easier to deal with failures of a single application.

- In this type of implementation, the services communicate with each other through APIs, and the internal details of each API are hidden from other services, thus making the code more secure and increasing the solution's coupling.
- API Gateway is a very used component, which is the entry point for client applications or other services, and it forwards incoming requests to the respective services.

Despite talking about features, not all microservice architectures have most of these features as with any architecture.

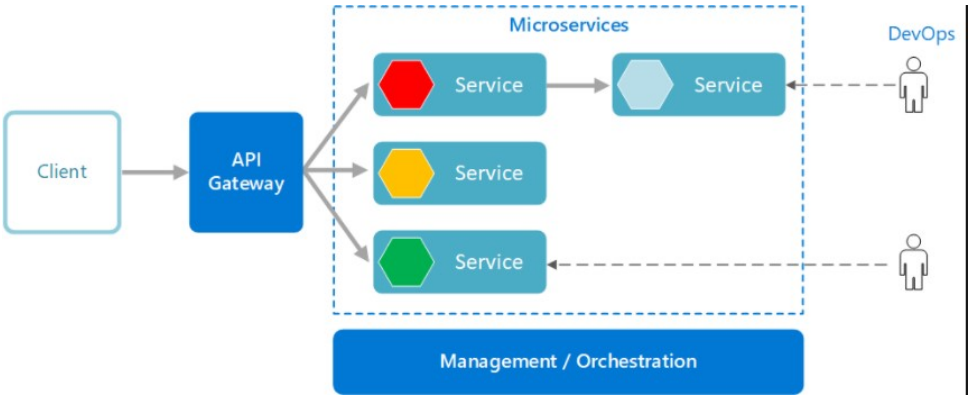

Figure 2.8: Microservices Architecture [26]

2.2.3 Benefits

As mentioned, according to [20], [23], [25] microservices have advantages that are listed below:

- Agility, as microservices are implemented independently, facilitates when it is necessary to add new features or correct errors, and this makes it possible to change an entire microservice without having to change the entire application.
- Less code in a single service because microservices have their code without sharing it among themselves, which makes each one only have code for the business part for which it was designed, which facilitates future changes.
- Scalability, because each microservice is entirely independent, makes it possible to expand some microservices that need more resources without influencing the entire application.
- Data isolation, as each microservice has only a tiny part of the entire business logic, and as each has its database, there will be more splendid data isolation, making the data more secure and more accessible to undergo any change.
- Technological freedom, as they are independent of each other, each team is free to choose the technologies that best apply to the microservice they are working on, without impacting other microservices.
- Reusable code, as this division into small and well-defined services often allows features developed in a microservice to be used by others or even for other applications.
- Ease of deployment, the introduction of microservices, instead of the old monolithic, made deploying an application and making changes much more manageable. When applications were monolithic, when a change was made, whether small or large, the entire application had to be

deployed, whereas a microservices solution allows us to deploy only the changed microservices. Also, make it easier in cases that happen error because it is easier to identify what caused the error, and it is also easier to roll back if necessary.

- Resilience, an essential concept for all applications and concept refers to a system being able to tolerate failures without being inoperable. As microservices are independent applications, they have already increased the resilience of applications because if an error occurs in one microservice, the others will continue to function normally, making the application not completely stopped as would happen in a monolithic application.

2.2.4 Challenges

A microservices architecture does not only have advantages, but its implementations also bring challenges. Furthermore, it should always evaluate the pros and cons of designing an application to understand the best architecture.

According to [20], [23], [25] the main challenges:

- Increased complexity, an application that uses a microservices architecture has several microservices to control and monitor. Each service is very simpler, but the application as a whole becomes more complex to control.
- Difficulty in carrying out the tests because there is a dependency between the services, it will be more complicated to carry out the tests because it is necessary to consider the integration between them.
- Data integrity, as each microservice is responsible for managing its data, being totally independent of the others and often even using different technologies, data consistency can become difficult. This issue can be resolved by doing good initial application planning.
- Network congestion, the application is divided into many services, which will cause more communication between them. Moreover, when the application design was done, this problem was not thought about, there can be significant dependence between different microservices, and this additional latency can become a big problem.
- Lack of governance, the tremendous technological freedom and standards that a microservices architecture provides, can also bring a significant disadvantage: the application becomes so diversified that it makes all its management difficult. So for that reason, it is advisable to define some standards at the beginning of the project.

Chapter 3

State of the Art

This chapter is dedicated to showing state of the art, where advantages and disadvantages of using GraphQL will be presented, use cases of some companies that implement GraphQL with microservices will also be presented some patterns, approaches and frameworks. The most relevant quality attributes are presented at the end of the chapter.

3.1 Microservices with GraphQL

In this section, it was discussed how the implementation of GraphQL can change the use of microservices and how this can mitigate some of the challenges presented in section 2.2.4 or what new challenges may also arise from its use.

This approach of using GraphQL with microservices has started to be explored to mitigate some microservices related problems. The main benefits of using GraphQL in microservices implementations and some disadvantages that arise from this use will be presented.

According to [23], [27], [28] the main benefits of GraphQL with microservices are the following:

- A single access endpoint: an implementation with GraphQL provides both clients and servers with a single access point to obtain all data, regardless of which microservice contains the information, so that GraphQL can hide the microservice implementation used from clients in the server. Also, depending on the data model, this will give us the advantage of having fewer requests made, with greater availability on microservices.
- Possibility of describing the data needed (over-fetching): GraphQL can offer customers a way to describe the data accurately they are requesting, as the customer has the possibility when requesting to define which fields are returned. This functionality of returning only the exact fields needed minimizes the amount of data transferred between a client and the server. This will also allow the server to offer a wide selection of data without worrying about the excess of data sent, as this responsibility for selecting the desired fields is on the client's side.
- No query optimization required: the hierarchical queries provided by GraphQL allow to eliminate the need to write queries optimized for specific use cases. This will help keep the server-side as general as possible with less specific optimizations for the clients.
- Easy version control: version control or publication is unnecessary, as all queries are handled through a single terminal. As the customer only receives the fields specified in the query, it is only necessary to notify the customer if some fields have been changed.

- Easy documentation: the data models and all operations that can be applied and defined can be documented in the schema, and as a result, the GraphQL-based API will automatically generate inherent design documentation.

Moving on to the disadvantages, are present the ones that can have the most impact:

- Denial of service attacks: this GraphQL vulnerability is because the clients can freely define overly complex queries. Some solutions have already been presented, such as limiting the possible size of the query and predefining all the allowed queries, these solutions are easy to implement, but they may not be enough to identify attacks. To solve the problem, it is possible to use security features that can be integrated with libraries that implement GraphQL. [23].
- Only brings advantages in some solutions: being a new approach to application development will take some time, as there are few specialist professionals. Therefore, another approach is advisable when the microservice objective is mainly to serve other services. Reducing the size of data transfers is no longer critical because services are more likely to bring as much data as possible.
- Lack of documentation: one of the most significant disadvantages comes from being a technology so recent that there is not enough documentation to support the developments, and even in terms of problems that can be found, there are still no forums entirely dedicated to helping solve problems.

3.2 Patterns with GraphQL applied before federation

When GraphQL appeared a few years ago, and organizations decided to start experimenting with implementing this new technology, they were trying to use a simple architecture where a client application queries a single GraphQL server, and then this server makes requests to the backend and returns the desired data to the client. Moreover, companies started to make their approaches with this simple architecture in mind [29].

3.2.1 Client-only GraphQL

This pattern appeared because client application teams were eager to reap the benefits of GraphQL's centralized queries, so they implemented a GraphQL API in the context of their application. So the teams wrapped all existing APIs as a single GraphQL API endpoint (figure 3.1).

This solution was adopted because it improved the customers' development experience, layering the GraphQL API over other APIs. However, although the developer experience was improved, the application incurred performance costs, as it is still necessary to make multiple requests to various services to get all the data.

Figure 3.1: Client-only GraphQL [29]

3.2.2 Backend to Frontend

GraphQL can also be applied to solutions that use the BFF pattern (figure 3.2), which tries to solve the problem of having different clients, as explained before, section 2.2.1.

The use of BFF adds a layer where each customer has a dedicated layer that is highly focused on that user's experience. So the teams that applied BFF added GraphQL to build that middle layer focused on each customer.

Like the previous standard, BFF is a customer-focused approach and has the advantage of providing a better experience for developers, but it also manages to overcome the performance issues that occurred in the previous standard. That is because they provide a unified interface for every client application.

This pattern has the disadvantage that when each client team can create a BFF to meet that client's needs, there will be an inevitable duplication of effort between teams.

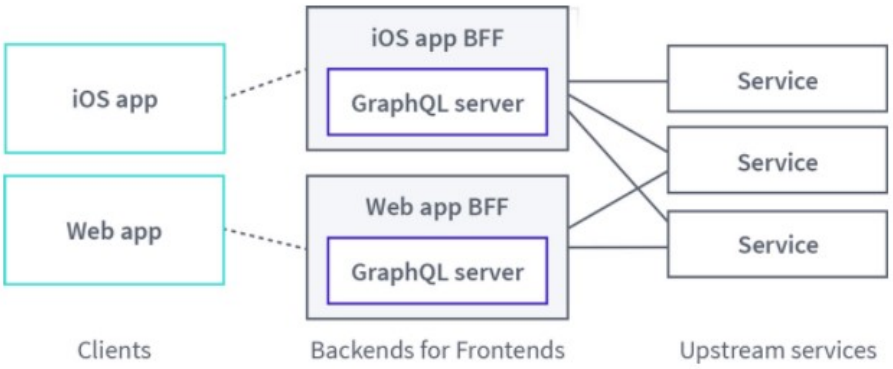

Figure 3.2: BFF Pattern GraphQL [29]

3.2.3 Monolithic GraphQL

This pattern can take two forms, and the first one teams can share code for a GraphQL server that multiple clients use. This code is organized and has the property that different developers share the graph.

The other way is where there is a single team that has a GraphQL API that multiple clients access, this team usually sets the standards to follow.

As with GraphQL-based BFF, maintaining a single, monolithic GraphQL API (figure 3.3) can help prepare an application for consolidation efforts and moving to a federation approach.

The challenges encountered in the BFF pattern are intensified with monolithic GraphQL servers that have shared properties across teams. Because part of the chart may be well designed to meet the needs of one customer's team, but others may not.

Even in the other application scenario of a monolithic GraphQL server, the challenges arise when more than one graph definition is needed for a single product to support different clients. It also presents the difficulty that the team responsible for developing the GraphQL API is overloaded with the task of building and maintaining the necessary tools and at the same time without breaking compatibility for any clients that consume the API.

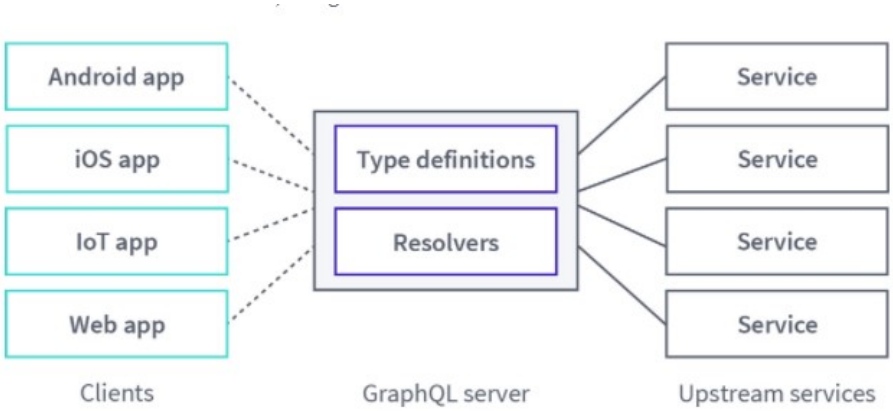

Figure 3.3: Monolithic GraphQL [29]

3.2.4 Summary

All the patterns explained and studied previously have a critical deficiency in common as their implementations result in a lack of data consistency.

The federation is generally not a starting point for most companies that decide to start adopting GraphQL. The federation requires large-scale education and integration efforts for the teams responsible for managing the chart.

However, as services logic increases and teams technology stacks expand, pain points emerge as other approaches are applied. That is why many of them decide to change from these standards applied initially to a complete approach that can end the pain that emerged, and that is where GraphQL federation appears as the great solution found so far.

3.3 GraphQL Federation

As previously studied, GraphQL came to bring some advantages with its implementation with microservices, but its simple implementation without adequate architecture also brought different challenges that companies began to face. So in 2019, Apollo released Apollo Federation, an implementation that came to solve some of the old problems found in a simple implementation of GraphQL.

"Apollo Federation is a set of tools to compose multiple GraphQL schemas declaratively into a single data graph" [30].

In order to take full advantage of GraphQL, companies should only expose a single graph that provides an interface to query any data. However, it can be very challenging to represent just a single graph, and it was to solve this difficulty Apollo decided to release the Apollo Federation to make it possible to divide the implementation of its graph into several backend services, called subgraphs by Apollo.

The architecture of the Apollo Federation (figure 3.4) has two main points:

- Subgraphs represent the different backend services that define a distinct GraphQL schema.
- A gateway that uses a supergraph scheme is composed of aggregating all the subgraph schemes to execute the different queries requested by the clients.

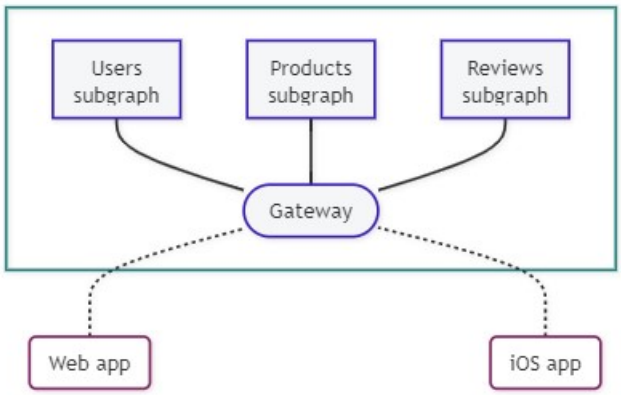

Figure 3.4: Apollo Federation Architecture [30]

3.3.1 Design principles

A federation GraphQL architecture has two fundamental principles, incremental adoption and separation of concerns.

Like any GraphQL schema, it must be built incrementally and evolve. If teams still use a monolithic GraphQL server or another GraphQL architecture, the federation allows dividing each functionality into small subgraphs, and thus it becomes possible to incrementally define service boundaries and identify appropriate connection points between subgraphs. Customers will not notice that implementations have changed in this situation, and an incremental migration can therefore be done.

The federation also applies the principle of separation of interests, which is also one of its main architectural advantages. Federation allows teams to depart from the unified chart using concern-based separation instead of type-based separation, differentiating the federation from other approaches.

The separation of concerns allows each service to define the fields it can be responsible for from its backend data store. The other services can then directly reference and extend these types in their schema. The result will allow teams to maintain repeatable parts of the chart without relying on others and allow customers to access a product-centric layout.

3.3.2 Federated Schemas

A single federated graph uses different types of GraphQL schemas (figure 3.5).

Subgraph Schemes

Each subgraph has a unique scheme that indicates which types and fields its composite supergraph are responsible for solving. When defining subgraph schemes, it must be taken into account that each subgraph must be implemented as an independent service.

When developing subgraph schemas, some important conventions must be understood. The first refers that a subgraph can refer to a type defined by another subgraph, a subgraph can also extend a type defined by another subgraph, and finally, a subgraph must add the directive when defining an object type that other subgraphs can refer to or extend, the directive to use is "@key" which represents an entity.

In the Apollo Federation, an entity is an object type that is canonically defined in a subgraph and can reference and extend in other subgraphs.

After defining an entity in a subgraph, other subgraphs can then reference that entity and add the field of type to its type. A subgraph can also add fields to an entity defined in another subgraph, called an entity extension.

Supergraph Schema

This is the schema resulting from performing the composition of subgraph schemes. It will match all types and fields of subgraph schemas and define specific directives that will tell the gateway which subgraphs are responsible for resolving specific fields.

API Schema

This schema is similar to the supergraph schema but will omit type, fields, and directives considered "machinery" and are not part of the public API. This is the schema that the gateway exposes to GraphQL API consumers, so they do not need to know specific implementation details. The gateway is only for planning and executing GraphQL operations through its subgraphs.

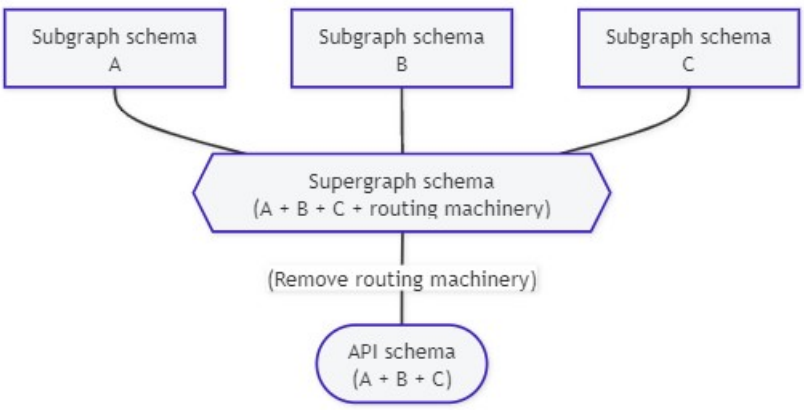

Figure 3.5: Federation Architecture [30]

3.3.3 Consolidation Decision Matrix

As explained earlier, GraphQL federation is not a simple approach to implement, and it may not always be beneficial to make the switch to a federative architecture, so when considering whether to implement federation for the first time, the most critical responsibility of the software architect is to understand why this change is happening. Therefore, at a strategic level, a matrix has been developed that reliably measures the adoption and evolution of GraphQL towards a federated implementation. The consolidation decision matrix consists (table 3.1) of a series of questions that those responsible must periodically answer to have an ongoing assessment of how and when their consolidation efforts should proceed.

Apollo's recommendation is to keep this exercise stable and straightforward, with stakeholders using the matrix as a regular process to facilitate formal decision-making.

Table 3.1: Consolidation Decision Matrix [29]

CONCERN	YES/NO	REMEDIATION/GUIDANCE
Consensus	Are multiple teams contributing to your graph?	If this is an initial federated implementation, identify your "Graph Champions" and establish education, review, and governance processes.
Responsibility	Are contributions to your graph by multiple teams regularly causing conflicts with one another?	Are contributions to your graph by multiple teams regularly causing conflicts with one another?
Delivery	Is there a measurable slowdown or downward trend in GraphQL service change delivery?	If there isn't a measurable, negative impact to product or service delivery, consider the additional complexity and support for this change.
Delivery	Is there a concrete security, performance, or product development need to deliver portions of your existing schema by different teams or different services?	If consumers or internal stakeholders are not currently affected, consider revisiting the driving factors for this change.
Consensus	Is there a single source of governance for your GraphQL schema within the organization?	An initial Federated implementation, or an early expansion of Federation, are good opportunities to create support systems for education, consensus-building, governance, and quality control.
Consensus	Does your GraphQL governance process have a reasonably robust education component to onboard new teams?	Apollo has found that a robust education plan is a leading indicator of constant improvement and success.
Delivery	Is your existing GraphQL schema demand-oriented and driven by concrete product needs?	Changes driven by data-modelling or internal architectural requirements may not have an ROI when weighed against the costs of infrastructure and organizational change.
Responsibility	Do you have a strong GraphQL change management, observability, and discoverability story, and do providers and consumers know where to go for these tools?	Graph administration and tooling such as Apollo Studio are key elements in a successful, organization-wide GraphQL initiative.
Consensus	Is your existing GraphQL schema internally consistent, and are your GraphQL schema design patterns well-understood by providers and consumers?	Dividing responsibility or adding new schema to your Graph without strong governance may exacerbate existing friction or product/service delivery challenges.
Performance	Can you be reasonably sure that the cost of additional latency, complexity, and infrastructure management will have a positive ROI when bound by business timelines and objectives?	Ensure that the requirements for separating concerns have a performance and optimization budget.

If answering the matrix questions, all answers are "yes", should continue to establish a clear path to successful implementation. If the answers are unclear or "no," leaders must be careful in evolving their implementation. Each "no" must be used to identify and monitor metrics and indicators that demonstrate that change is necessary, and each one must be approached with the desire to connect with the team that is developing the work and understand how it can become in a "yes".

Because GraphQL can be an organizationally transformative technology, care must be taken to involve all stakeholders in planning and implementing a federated graph. As a result, education plays a crucial role in the success of federated implementations.

3.4 Frameworks

In this section, some of the frameworks used to implement GraphQL federation are presented.

3.4.1 DGS

The Domain Graph Service (DGS) Framework is a GraphQL server framework for Spring Boot and was developed by Netflix to implement federation [31].

The DGS framework was developed in Kotlin but is primarily aimed at helping Java developers. The framework is based on Spring Boot and is only focused on this type of environment, not being possible to use it in other situations. The framework is designed to have minimal external dependencies.

The structure that makes up the DGS framework was built on top of graphql-java. Graphql-java is and should be a low-level building block for handling the execution of operations. DGS provides everything graphql-java offers with a convenient spring boot-specific programming model [32].

The DGS is based on the Apollo Federation subgraph specification. The Apollo Federation subgraph specification defines what needs to be done to make a subgraph capable.

According to [31] the main features of the DGS include:

- Annotation-based Spring Boot programming model.
- Framework for writing different types of tests.
- Gradle Code Generation plugin to create types from schema.
- Easy integration with GraphQL federation.
- Integration with Spring Security.
- GraphQL subscriptions.
- File uploads.
- Error handling.

3.4.2 AWS AppSync

AWS AppSync is a fully managed service that makes it easy to develop GraphQL APIs while handling the work of securely connecting to data sources. Using AWS AppSync to build GraphQL APIs gives client application developers the ability to query multiple databases, microservices, and APIs from a single endpoint [33].

The AppSync APIs have three fundamental components to define: a schema, resolvers, and data sources. These three components allow API to interact with resources and produce responses in the desired format [34].

Using WAS AppSync, when a request arrives, it is verified using the schema. Once the requested object is determined, the respective resolver is called, which defines request models and translates the data between GraphQL and the data source. After the resolver has translated the query, it is sent to the data source, which will return a result that will be translated back to a schema compatible format.

Like the DGS, AWS AppSync is based on the Apollo Federation subgraph specification.

According to [33], AWS AppSync has the following features:

- Add caches to improve performance.
- Subscriptions to support real-time updates.
- Client-side data stores that keep offline clients in sync are just as easy.
- Automatically scale GraphQL API execution engine up and down to meet API request volumes.
- Access control directly in the GraphQL schema.

3.4.3 Apollo Server

Apollo Server is an open-source GraphQL server. It works with many NODE.js HTTP server frameworks, or it can run on its own with a built-in Express server. Apollo Server works with any GraphQL schema built with GraphQL.js or defines schema types using Schema Definition Language (SDL) [35].

Apollo server can be used in a few different ways. It can be used as a completely standalone GraphQL server, it can be a complement to the already existing middleware node, or it can even be a gateway to a federated graph.

Apollo Server [35] have the following main features:

- Simple configuration so that client developers can perform data fetching quickly.
- Incremental adoption, allowing new features to be added only when needed.
- Compatible with any data source, any compilation tool and any GraphQL client.
- Production readiness, allowing for shipping resources faster.

Another framework provided by Apollo is the federation-JVM. This framework is very similar to the apollo server but developed to be used with Java code. This framework is an alternative for those who want to develop microservices with GraphQL federation in Java but will face difficulties finding documentation on how to implement it.

3.4.4 Mercurius

Mercurius is a GraphQL adapter for Fastify. Mercurius is used to implement GraphQL servers and gateways. Fastify is an open-source Node.js framework that provides the best developer experience with less overhead and a robust plugin architecture [36].

According to [37] the features of Mercurius are as follows:

- Caching of query parsing and validation.
- Automatic loader integration to avoid 1 + N queries.
- Just-In-Time compiler via graphql-jit.
- Subscriptions.
- Federation support.
- Federated subscriptions support.
- Gateway implementation, including subscriptions.
- Batched query support.
- Customisable persisted queries.

One of the major problems of this framework is the lack of documentation.

3.4.5 Summary

The Apollo Server and DGS frameworks will be used to implement the chosen architecture. The Apollo Server framework will be used to develop the gateway, and the DGS will be used to implement the subgraphs. Apollo Server was chosen for use in gateway development because it is developed in typescript, and this framework is used with javascript. The other reasons are that there is a lot of very detailed documentation and many examples of implementation, which will be essential to help in the developments.

The DGS was the other framework chosen because it is used in java applications, the same language used in microservices. Other advantages of this framework are the existing structured documentation and some examples of implementing it and later testing it. Another option that could be chosen to apply federation in Java was federation-jvm, but this framework has almost no documentation, which could bring great difficulties in its implementation.

Both frameworks chosen do not require any license, which is a tremendous advantage because anyone who wants to replicate the project will not have any problems with the technologies used.

3.5 Enterprise Usage

3.5.1 Netflix

Netflix is known in the tech world for its highly scalable, loosely coupled microservices architecture. Using standalone services makes it possible to evolve at different paces and scale these services independently. However, they add complexity to use cases that span multiple services [8].

As the complexity of the domain increased, the development of the API aggregation layer became more and more difficult. So to solve this growing problem, they decided to implement GraphQL federation to feed their API layer, and this solves many of the consistency and development velocity challenges with minimal tradeoffs on dimensions like scalability and operability.

They were able to implement GraphQL federation for their Netflix studio ecosystem successfully. This ecosystem refers to the entire process of producing original Netflix content, from the moment they decide to make a movie or show until the moment it is made available, and this process includes all the events that take place behind the scenes.

A few years ago, one of the biggest headaches of the Netflix studio space was the increasing complexity of data and its relationships and to solve this problem, implement two architectural patterns, single-use aggregation layers and materialized views. However, despite the initial benefit they had in the performance of the API, data inconsistency began to occur, which was not acceptable for the most critical workflows in the Studio.

In order to improve this previous solution, the Netflix team started building a Graph API called "Studio API". The goal was to provide a unified abstraction over data and relationships. To accomplish this implementation, Studio API used GraphQL as the technology, which helped significant leverage to access key shared data. Studio API consumers had the opportunity to create new features faster. Additionally, fewer instances of data inconsistency were observed across different UI applications as each field in GraphQL is resolved into a single piece of data-fetching code.

This solution found ("One Graph") exposed by Studio API was a breakout success. Product teams loved the reuse and easy and consistent access to data, but new bottlenecks emerged as the number of consumers and amount of data in the graph increased. The first problem encountered was that the Studio API team was disconnected from the domain expertise and the product needs, which negatively impacted the schema's health. The other involved manually connecting new backend elements to the Graph API contradicted the rapid evolution promised by a microservices architecture.

With these problems identified, Netflix knew there had to be a better way to solve these problems, so Apollo released the GraphQL federation Specification, which promised a unified scheme with distributed ownership and implementation and with this GraphQL specification, Netflix released its new "Studio Edge" architecture, emerged with the federation as a critical element.

Differences from One Graph to Federation

The GraphQL federation has emerged, offering a unified API for consumers while offering developers flexibility and service isolation [8]. In a simple implementation of GraphQL, the GraphQL framework would receive queries from clients and orchestrate calls to the resolvers in a wide traversal, while in a federated architecture, it is necessary to transfer ownership of these resolvers to the respective domains without sacrificing the unified schema, for it often becomes necessary to do some extension across GraphQL service limits.

Studio Edge Architecture

The DGS is a spec-compliant standalone GraphQL service. Each developer defines their federated GraphQL schema in a DGS, and the DGS is owned and operated by a domain team responsible for that subsection of the API (figure 3.6).

Schema Registry is a component that stores all schemas and schema change for each DGS. It also exposes Create, Read, Update and Delete (CRUD) APIs for the schemas, which developer tools will use. This component is also responsible for schema validation, both for DGS schemas and the combined schema. Last, the registry composes the unified schema and provides it to the gateway.

GraphQL gateway is primarily responsible for serving GraphQL queries to the consumers. Takes a query from a client, breaks it into smaller sub-queries, and executes that plan by proxying calls to the appropriate downstream DGSs.

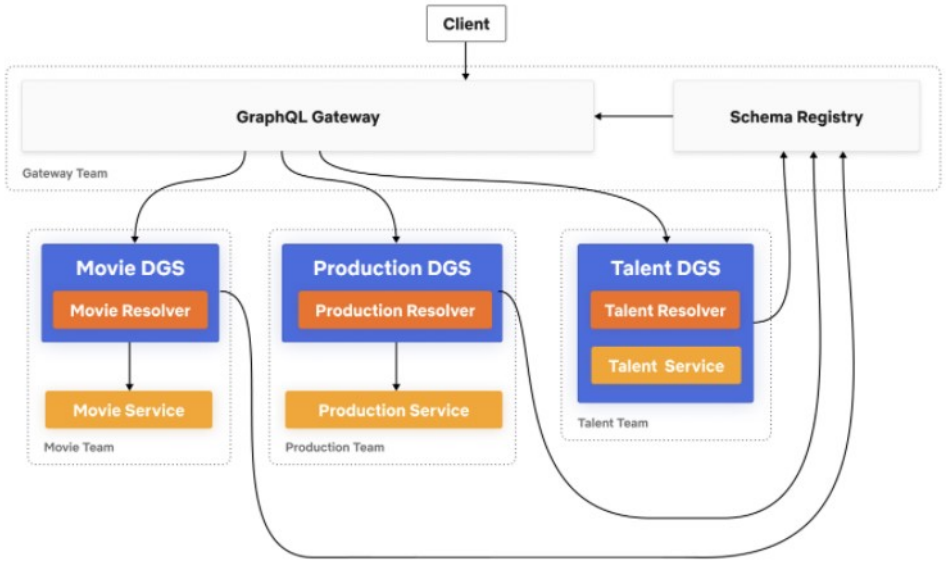

Figure 3.6: Studio Edge Architecture [8]

Implementation Details

There are three main business logic components in the implementation performed with GraphQL federation [8].

Schema Composition is the phase that takes all DGS schemas and aggregates them into a single unified schema, and the gateway later exposes these schemas to consumers. At this stage, the Schema Registry validates that the new schema is a valid GraphQL schema. It also validates that this new schema composes seamlessly with the rest of the DGSs schemas to create a valid composed schema, and finally, this schema is backwards compatible.

Query planning and execution refer to the component where the gateway uses the federation configuration and the client's query to generate a query plan. The federation configuration consists of

the individual DGS schemas and the composite schema. The query plan broke the client's query into smaller subqueries and sent to downstream DGSs for execution, along with an executive order that includes what needs to be done in sequence versus running in parallel.

Entity Resolver is the final part and implements how subqueries parallel to the DGS are performed. For this logic, Apollo introduced the @key directive second, DGSs have to implement a resolver for a generic query field, "_entities", and queries to that generic field return a union type of all types federated in that DGS.

3.5.2 RetailMeNot

RetailMeNot is part of Ziff Media, INC, and aims to make the days more accessible for shoppers because it also makes cashback offers for its users [38].

As in many companies, the ReatilMeNot teams used REST APIs to transport their data, but as the system grew, it started to have some problems with versioning, over-fetching, and under-fetching. They decided to implement GraphQL as a monolithic solver to solve these problems (figure 3.7).

This monolithic GraphQL was a single server used by its clients, and there was only one team responsible for designing and implementing the patterns for the new data graph. Due to the system's growth, the team responsible for maintaining the monolithic graph began to have a considerable burden, and problems began to arise when adapting to the new changes. So the organization realized that all teams should have the autonomy to expand the chart and not just a single centralized team. That is when they started to consider other solutions for their approach.

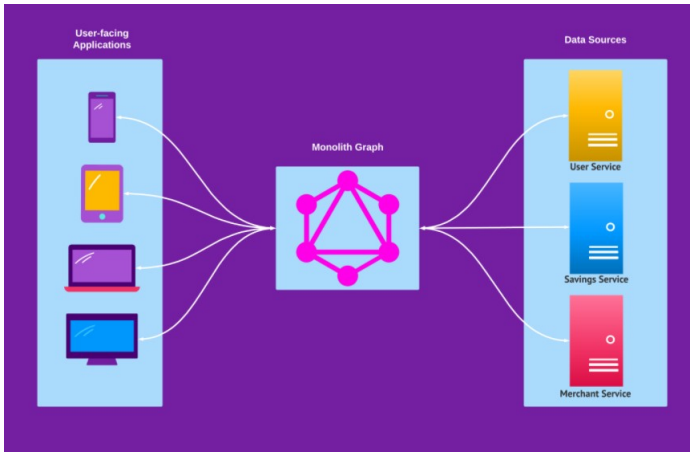

Figure 3.7: RetailMeNot Old Architecture [38]

Using GraphQL Federation

Due to the sever problems identified with monolithic GraphQL, RetailMeNot decided to change its approach to GraphQL federation (figure 3.8), as this approach allows each subgraph team (microservice) to be responsible for implementing and maintaining their part of the schema in the unified graph.

In order to make this change the best way possible and get help dealing with the issues, they partnered with Apollo GraphQL and used their Managed Federation architecture.

One of the difficulties they faced when applying federation was ensuring zero downtime while migrating from the old monolith to federated architecture. They followed the incremental migration approach suggested by the Apollo Federation in their documentation.

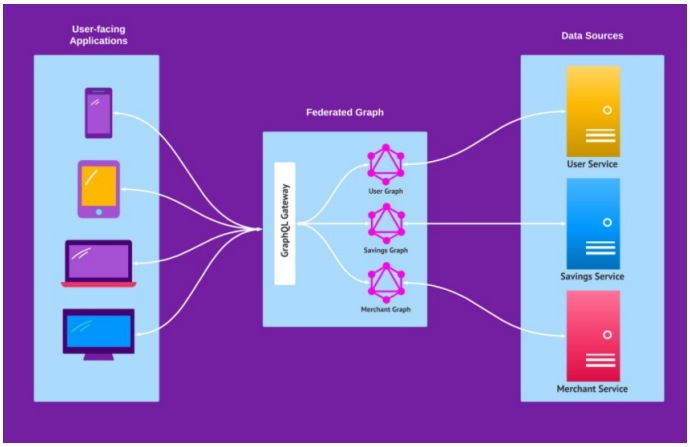

Figure 3.8: RetailMeNot New Architecture [38]

To achieve this process, they followed a set of incremental steps:

- Transform the existing monolithic GraphQL schema to support the specifications of a federated architecture.
- Create a new gateway that will only serve to forward traffic.
- Then controll the amount of traffic that would hit the new gateway service vs the monolithic service using a weighted routing technique. Once confident with the changes and validations in the lower environments (stage/pre-production), then there is the switch to 100% of the traffic going through the gateway service in the production. At this point, the monolithic service was entirely behind the gateway.
- Finally, split the old monolithic schema into subgraphs and migrate the entities to the subgraph services.

After following these steps to migrate to the federation, they were left with a unified data graph consisting of several fully federated and highly cohesive subgraph services independently developed by different teams.

Advantages of moving to a federated architecture

- **Faster product iteration:** the user-facing application teams can move faster as the bottleneck from a single "GraphQL API team" is removed.
- **Worry-based separation:** using a federated architecture allowed different teams to work on different product services autonomously while contributing to a single graph.
- **Similar tech-stack across services:** this move to federated architecture has helped have well-defined standards across multiple services in RetailMeNot.
- **Developer experience:** with the standardization of tools and a standard technology stack, the developer experience has improved.

3.5.3 Walmart

Like many companies, Walmart needed to make changes to its current architecture and opt for a GraphQL federation architecture, and here the motivation that led them to change will be explained, as well as some cases and patterns they encountered in their migration from REST to GraphQL federation. Walmart, like other companies, used the Apollo Framework to apply GraphQL federation [39].

The main problems found at Walmart with the REST architecture

- Domain model consistency: in the architecture used previously, they used pages powered by BFF orchestrators, these orchestrators that responded to each page were developed based on REST, but each orchestrator needed custom visualization models for the pages. There was no enforcement of commonly shared data models resulting in bespoke implementation for a product title on each page.
- Developer Ergonomics: the Walmart team wanted developers to have a consistent data model to work with that was always up to date and readily available. With federated GraphQL, this became possible, as it is possible to extend schemas and add functionality without excellent coordination between teams because there is great autonomy.
- Performance: is a significant concern of all applications, and with the federation model, the gateway is on the same Wide Area Network (WAN) as the services generating calls to services quickly. In addition, the gateway can make its network path optimizations, minimizing calls to services.

Learned with Federated Graphs

While GraphQL alone managed to solve many of the problems at Walmart, GraphQL federation is not a miracle solution that will solve all problems of sharing distributed graphs. Some cases do not fit the pattern very well, and it may be necessary to make changes to improve [39].

Shared entities

Shared entities are models that are cross-cutting across multiple domains. A straightforward GraphQL schema would initially get the modules and only later enrich the modules of the federated services. However, this approach may not work in some cases as it can bring additional latency, which is not desired, so different approaches should be considered depending on the situation.

Federation Directives

The Walmart use case refers to two essential guidelines for achieving federation, "@extends" and "@requires".

The "@extends" directive is a specific directive that can be used in GraphQL and indicates that the entity is a remote entity referenced in the current service. Basically, it is a remote entity, and it is used to fetch it from service, and the entity also needs to specify the control of how to fetch it.

According to Walmart, there are two reasons why a dependency should be specified, when it is needed to decorate the module to add functionality and when it is used as a remote entity as a composite field in a type.

The "@requires" directive is used to specify fields required internally for the business logic, this field that uses this directive may not be consulted by the client, but it is fundamental in the service to resolve the field in which it is defined.

Execution of the query plan

The gateway parses the queries and prepares the orchestration path, for which it analyzes the dependencies between the services as specified by the "@extends" directive, so all dependency fetch calls are performed before calling the service. Although this can be a good approach, as services that have dependencies resolve them before the service itself is called, there can be complications when, for performance reasons, initiate the call to resolve the service before need the dependencies are run in parallel [39].

Service Integration Patterns

To facilitate the quick identification of GraphQL patterns, Walmart decided to define names for some patterns:

- Atomic Entity: simple integration when a service has no dependencies.
- Mini Orchestrator: the GraphQL service orchestrates directly with other services due to a complex dependency to resolve the schema.
- Aggregate Entity: aggregate entities reference remote entities and aggregate the types. They need to do this because they do the pre-process before resolving the remote entities.
- Extended Entity: an extended entity decorates a remote type by adding new fields.

3.5.4 RS Components

Currently, most organizations that have adopted federation feel that the architecture behind it is confusing, and in this case study, the concept of schema services is introduced and how they can provide an easy transition to a federated architecture [40].

RS Components developed a transitional architecture to overcome the confusing architecture and the skills gap over GraphQL. The company's architectural change started with migrating their monolithic application to a microservices architecture. It became even more complicated when they realized that a mix of protocols was being used in the various services (REST and GraphQL), and the company wanted to change all to native GraphQL. However, these changes would take a long time, and there was no ability to update all the services, so they had to find a solution that would allow them to create the federated schema focused on products, which would lead them to a cleaner architecture. The solution to this problem was to introduce a set of services between the gateway and the underlying services, called schema services.

Schema Services

Schema services only deal with a single concern, turning multiple microservice responses into a single schema (figure 3.9).

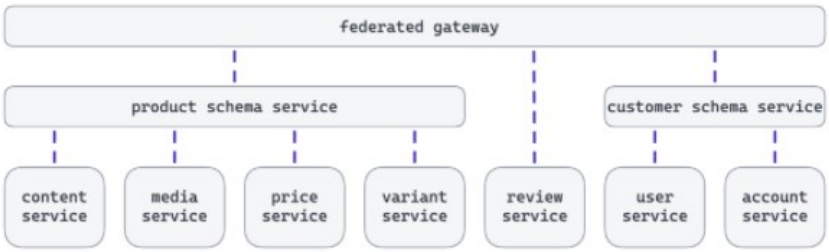

Figure 3.9: RS Schema Services [40]

Schema services are kept thin and are nothing more than an Apollo Server instance containing a schema backed up by simple resolvers that call underlying services. With the introduction of schema services, the underlying services can now be made available through the federated gateway without changing the services. So instead of converting all the services, only enter the schema services to create the federated data graph.

Schema services can be phased out of the transient architecture, as when the service team can take over the work of federating their service, they can be integrated directly into the gateway. And this step-by-step approach to federated services allowed RS to transfer the current architecture to the target architecture gradually.

Advantages

- There is no impact on existing services as schema services will make calls like any other client.
- The federated schema can grow as more requirements arise.
- Schema services are simple services with no business logic and are the easiest to develop.
- Federated schemas can be designed from scratch without accommodating any languages already used.
- The cache of schema services can be used to limit traffic to the underlying services.

Disadvantages

- There is an additional spike in all requests, adding latency to requests.
- There are more costs with the introduction of additional services in the architecture.
- Synchronized changes in schema services with changes in underlying services means there is coupling between layers.
- Ownership of schema services can become complex as it is challenging to maintain schema consistency.

3.6 Architectural Possibilities

When designing the architecture for a solution, it is always essential to consider some architectures to understand which one might be best suited to apply. They are also a starting point and a guide to designing the architecture in the best possible way. In this section, the architectures developed by Apollo and Netflix to apply federation will be presented.

3.6.1 Apollo Federation Architecture

The architectural reference used was developed by Apollo (figure 3.10), which presents an approach where we have different subgraphs, one for each microservice, each subgraph has its schema and contains all the logic that this microservice needs (resolvers). Another fundamental component is the gateway that will receive the requests and forward them to the respective microservices (subgraphs). In order to be able to do this forwarding and expose the current application schema to the client, the supergraph schema is used, which is the collection of all subgraphs schemas. This supergraph is responsible for telling the gateway which fields each subgraph is responsible for and helping the gateway in the forwarding process.

An important point when discussing the supergraph schema is how it is managed. It is crucial to have an automatic mechanism to drive it to facilitate this whole process and make it easy to manage its update and maintenance. Apollo automatically performs composition whenever one of its subgraphs registers an updated schema with the managed federation. This schema management provided by Apollo allows the various teams working on a graph to coordinate when it is updated. Apollo offers this federated schema management through Apollo Studio, completely free. With managed federation, the solution gateway is no longer responsible for fetching the schemas of the subgraphs because the subgraphs are responsible for sending their schemas to Apollo Studio, which will compose the federated schema. After the schemas are composed, Apollo Studio makes an endpoint available with the federated schema so that the gateway can search for schema updates.

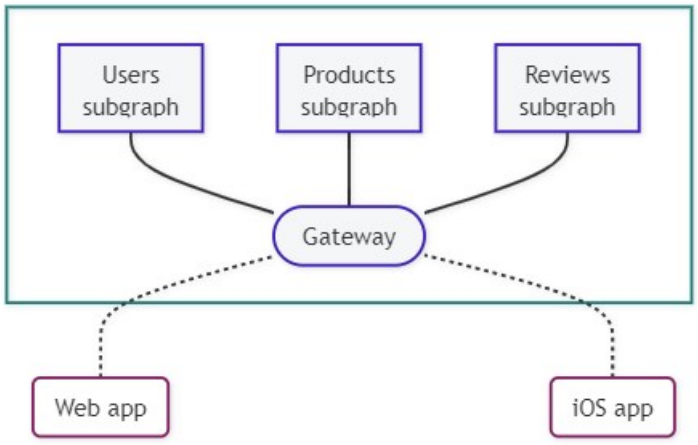

Figure 3.10: Architecture Apollo Federation [30]

3.6.2 Netflix Architecture

The architecture presented was applied by Netflix (figure 3.11) when it decided to migrate the Studio API to a federative architecture. This architecture is very similar to the architecture presented by Apollo, with some differences. The architecture is divided into three essential components. The first component refers to the DGS, the spec-compliant standalone GraphQL service, schemas and domain logic. The other component is the schema registry which is a component that stores all schemas and schema change for each DGS. It also exposes CRUD APIs for the schemas, which developer tools will use. This component is also responsible for schema validation, both for DGS schemas and the combined schema. The last component is the gateway primarily responsible for serving GraphQL queries to the consumers. It takes a query from a client, breaks it into smaller sub-queries, and executes that plan by proxying calls to the appropriate downstream DGS. To implement, Netflix uses the DGS Framework in the part of the federated schemas, but it uses the apollo gateway to manage its gateway.

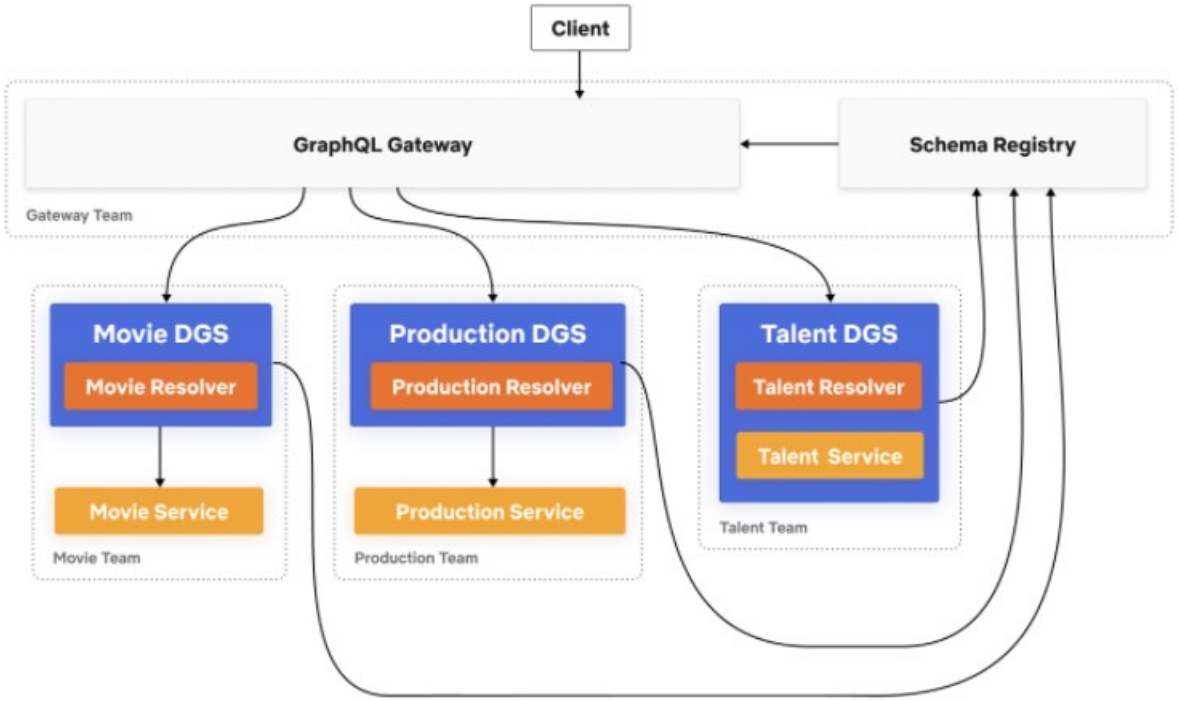

Figure 3.11: Netflix Architecture [8]

3.6.3 Summary

Of the architectures presented, the one that was chosen to apply was the one developed by Apollo, as Apollo itself has an external cloud component that manages all the management of the federated schemas and generates the superschemas, also managing to validate them. All this logic in the architecture developed by Apollo can be done automatically using Apollo Studio. In the architecture

developed by Netflix, the management and validation logic of the federated schemas has to be done manually to develop a new component in the system.

The Apollo Server and DGS frameworks will be used to implement the chosen architecture. The Apollo Server framework will be used to develop the gateway, and the DGS will be used to develop the subgraphs.

3.7 Quality Attributes

As explained earlier in section 1.3, the main objective of this document is to analyze in terms of quality attributes the implementation of GraphQL approaches with microservices, with the main focus being GraphQL federation. Therefore, in this section, the quality attributes chosen to evaluate the solution will be discussed, considering which may have the most significant impact on the federation application [41].

The ISO/IEC 25000 series of quality standards known as System and Software Quality Assessment and Requirement (SQuaRE) was used to choose the quality attributes, which focuses on the specification, measurement and evaluation of quality requirements of the software product. The ISO 25010 standard from this series is used, a standard whose main objective is to create a framework for evaluating software quality and refers to the quality division model - the quality characteristics considered when evaluating software [4]. ISO 25010 considers eight quality characteristics (figure 3.12), each with sub-characteristics.

Figure 3.12: ISO 25010 [41]

In order to identify the quality attributes that could be of greater interest in this dissertation, they were identified through the analysis of studies carried out in microservices where it was noticed that, despite a microservices architecture improving many quality attributes, there are still some that perhaps the current approach does not favour or that have not been adequately studied. Quality attributes were also identified through the difficulties mentioned above that forced them to opt for the federative approach and its advantages.

According to a study carried out by [7], there are quality attributes that are of great focus when one wants to study a microservices architecture "Quality attributes (QAs) are inevitably discussed in the practices of migration from monolithic applications to MSA, for example, maintainability, reuse, and scalability" [7]. In the same article, a study was also carried out to understand the most concerned QAs for MSA, to identify the most frequently emphasised QAs challenges related to MSA, and it was noticed that there are six, two of which stand out, "six most concerned QAs from reviewed

studies related to MSA: scalability, performance, availability, monitorability, security, and testability. Scalability and performance are the two most concerned QAs addressed in MSA" [7]. It was also identified that more study is needed to understand how to improve maintainability, "more empirical research is needed to study these QAs, in particular for improving maintainability of MSA" [7].

In another article made by the authors [42], who did a study on the challenges of microservices, it was also possible to perceive that maintainability is something essential and that worries many developers, "Code Management issues, especially around dealing with and preventing breaking API changes concerned almost all our interview participants (95%)" [42]. This problem has an impact when thinking about making changes "Our study participants make an extra effort to avoid breaking API changes, unless those are security-related. Breaking changes that face external customers are especially discouraged" [42]. Finally, they found other studies where "maintainability and scalability are the primary motivations for adopting microservices" [42].

Now analyzing the cases presented in section 3.5, maintainability is also one of the main focuses that led companies to change from their previous architecture to a federated architecture. The company RetailMeNot reports that due to the growth of its system, the team that was responsible for maintaining the monolithic graphic of its first implementation started to have a considerable load, and problems began to arise in adapting to new changes. Netflix also reported that as domain complexity increased, the development of the aggregation layer of the API became increasingly challenging to manage. The RS company presents maintainability as one of the advantages they found when using federation because the federated schemas can quickly grow.

Another quality attribute that companies mentioned as a significant concern were performance. The Walmart company refers to performance as a significant concern to consider in all applications.

After the analysis, it is realised that the quality attributes that can be more interesting and important to study are **maintainability** and **performance**, so the dissertation will focus on analysing these two quality attributes.

3.7.1 Maintainability

This characteristic represents the level of effectiveness and efficiency with which the system can be modified to improve, correct, or adapt to the necessary changes. In this feature, the focus is on the following sub-characteristics:

- Modifiability: level at which a system can be modified effectively and efficiently without introducing new defects and negatively impacting the quality of the existing system.
- Modularity: the degree to which a system is composed of discrete components such that a change in one component has almost no impact on other components.

3.7.2 Performance Efficiency

This characteristic represents performance about the number of resources used. From this feature, the following sub-characteristics were chosen for analysis:

- Time behaviour: degree to which a system's response and processing times take to perform its functions.
- Resource utilization: level of values and types of resources a system uses when performing its functions meet requirements.

Chapter 4

Value Analysis

This section presents value analysis, starting with developing the business process and innovation where the NCD model will be applied. Then the customer value, the value proposition, and finally, the application of FAST will be presented. After discussing with the teacher, it was concluded that applying the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS), Quality Function Deployment (QFD) and Analytic Hierarchy Process (AHP) methods made no sense.

Value analysis is defined as "A philosophy implemented by the use of a specific set of techniques, a body of knowledge, and a group of learned skills. It is an organized, creative approach which has as its purpose the efficient identification of unnecessary cost, i.e., cost which provides neither quality, nor use, nor appearance, nor customer features" [43].

4.1 Business Process and Innovation

The innovation process may be divided into three areas (figure 4.1): the Front End of Innovation (FEI), the New Product Development (NPD) process and commercialization [44].

The first FEI area is often regarded as one of the greatest opportunities for improving the innovation process. Front-end activities use the formal and structured process to increase the value, quantity, and likelihood of success of high-profit concepts that go into product development and commercialization [45].

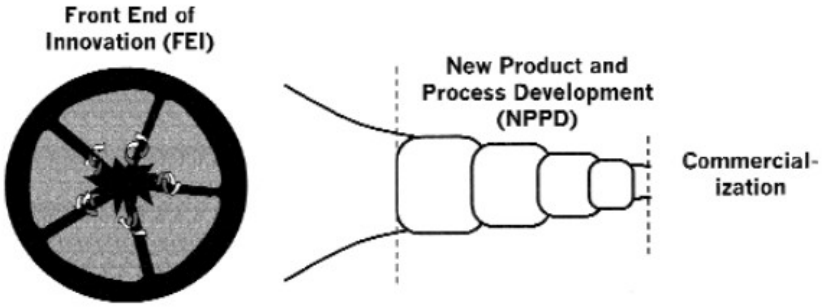

Figure 4.1: Business Process and Innovation [46]

The New Concept Development (NCD) model emerged to address some shortcomings in understanding the FEI and is intended to provide a shared vision and terminology for the entire FEI [45].

NCD Model features:

- The internal parts are called elements.
- The model has a circular shape to suggest that ideas should flow, circle and iterate between all the elements.
- The flow can even cover all elements in any order or combination and use elements more than once.
- Interactions and blending between the influencing factors, the five key elements and the engine are expected to occur continuously.

The NCD model is composed of 3 distinct areas (figure 4.2):

- Engine or bull's-eye: refers to the organization's leadership, culture and business strategy, which will drive the key elements that are controllable by the organization [45], [47].
- Inner: this is where the five key elements of activity are, which are the following:
  - **Opportunity identification:** this element identifies the opportunities that can be pursued through the market definition. The sources and methods used to identify existing opportunities are essential to this element.
  - **Opportunity analysis:** the defined opportunities are evaluated through the market study and analysis of the competitors to try to understand the potential. However, this is an element of uncertainty because uncertainty will always remain no matter how much it analyzes the market.
  - **Idea generation and enrichment:** this element concerns an idea's birth, development, and maturation. The idea generation element is evolutionary because it can go through many changes and iterations as it is examined. The generation of ideas can also feed the identification of opportunities, thus demonstrating that the elements do not follow a sequence.
  - **Idea selection:** in this element, the idea is selected through a process. The problem for most companies is to know which is the best idea to select to achieve more excellent commercial value, as there is no single process that guarantees a good selection.
  - **Concept definition:** this is the final element, providing the only output for the NPD. In this element, convincing arguments for investment in the business must be presented.
- Influencing factors: consists of organizational capabilities, the surrounding world and the enabling sciences involved. These factors affect the entire process from innovation to commercialization and are not controllable by the organization.

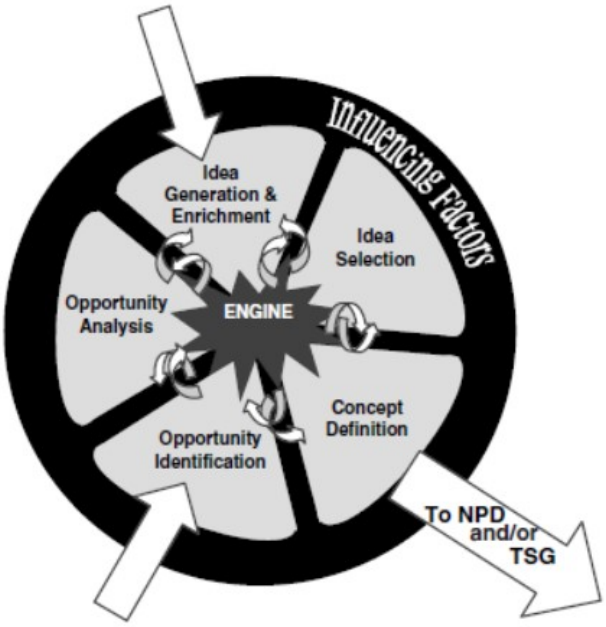

Figure 4.2: NCD Model [44]

4.1.1 Opportunity Identification

More and more companies are using microservices, leaving behind monolithic applications, as explained in section 2.2, due to the advantages it has brought to application development. Moreover, the emergence of GraphQL as a great alternative to implementing with microservices has made companies rethink the use of the previous APIs because of the innovations that GraphQL provides. There was a need to understand how GraphQL with microservices can improve application development.

The IBM Market Development and Insights team recently conducted a series of surveys that recorded microservices users' perceptions and real-world experiences and those considering adoption. These included more than 1,200 IT executives, developer executives and developers from large and midmarket companies currently using a microservices approach and nonusers exploring or planning to adopt this approach shortly [48] .

As a result of the research, the following results were obtained:

- 56% responded saying that in the next two years, they are very likely to use microservice.
- 78% of current users say their business will likely increase the money/time/effort invested in microservices.
- 59% (on average) of applications created by companies used microservices in the next two years.
- 48% (on average) have been created in the last two years.

A study by Matt Asay shows that there is growing interest in GraphQL, saying that GraphQL is behind a surprising number of well-known sites, including Facebook, Google, Airbnb, Pinterest and others.

A 2019 survey of over 20,000 JavaScript developers found that GraphQL adoption is exploding. A year after its release, 36% of developers surveyed had never heard of GraphQL. In 2017, that number was halved to 17.9%. Today it is 7.1%, with 62.5% of respondents saying they want to learn GraphQL [49].

Also, as we can see, since GraphQL appeared in 2015, there have been more and more questions on the topic of GraphQL on the StackOverflow (figure 4.3).

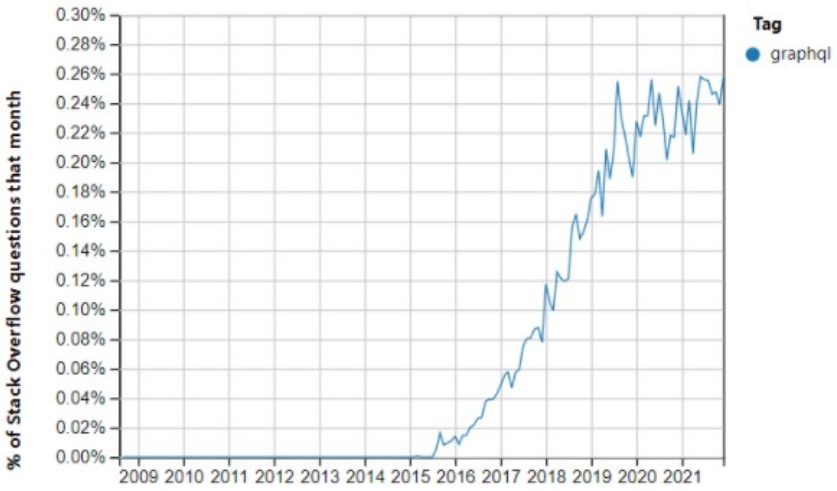

Figure 4.3: Analysis Stackoverflow [50]

4.1.2 Opportunity Analysis

For this element, a study was carried out on alternatives that already exist in the market, presented in the subsection 3.5 where it can be seen that it is still a little-explored market, and it is verified that more and more companies need to improve your applications continuously.

The use of microservices with GraphQL, can be a great opportunity because there is great hope that GraphQL can help make microservices even better, improving some of the deficiencies it had with other APIs or with less structured approaches.

4.2 Customer Value

In this section, the meanings of value for the customer, perceived value, value creation, and the benefits and sacrifices will be presented to understand better the concepts essential in value analysis and that help enter the concept of value proposition. These concepts may contain different definitions depending on the literature reviewed.

- **Value creation:** "is a concept that is difficult to achieve, understand, model and/or conceptualize. Some authors consider value creation a trade-off between benefits and sacrifices perceived by customers during a supplier's offering" [51].
- **Value for the customer:** "can be an ambiguous concept since it is defined based on how a customer interacts with a given product and its experience in fulfilling its needs" [52].

- **Perceived value:** "how a customer sees the utility of the product based on perceptions of what is received and what is given" [53].

The value for the customer concept presents benefits and sacrifices presented in table 4.1.

Table 4.1: Benefits and Sacrifices

	Product/Services	**Relationship**
Benefits	Product quality and Flexibility	Trust
Sacrifices	Price	Time/Effort/Energy

The search for these new solutions will benefit product quality because it will improve performance, and it will be easier over the years to increase improvements, as there is still much to explore. However, to achieve these benefits, there will also be sacrifices in this case in terms of prices, time, effort and energy, as it is a new solution that few people are yet fully aware of, it will be necessary to invest money and much time to study and apply it.

4.3 FAST

Fast is a technique that provides "a graphical representation of how functions are linked or work together in a system (product, or process) to deliver the intended goods or services" [54]. This technique is based on "How" and "Why" questions. The core of FAST is functions, so it is essential to understand what if these functions, using a more formal definition, a function is defined as "a product or process must do to make it work and sell" [55], but in FAST the description of a function is restricted to two words (Active Verb + Measurable Noun) [54].

FAST thus helps to think about the problem and to identify the objective of the project, showing the logical relationships about the functions. This diagram can verify that a proposed solution meets the project's needs and helps identify unnecessary, duplicate or missing functions.

Thus, through the analysis of the FAST diagram, the critical path can be identified: Critical path represents the sequences of functions that are the essence of the process, that is, which functions characterize the process. Any process that performs the critical path functions will be an administrative assembly process.

This solution has the following main benefits:

- Develop a shared understanding of the project.
- Identify missing functions.
- Define, simplify and clarify the problem.
- Organize and understand the relationships between functions.
- Identify the essential function of the project, process or product.
- Improve communication and consensus.
- Stimulate creativity.

When developing a FAST diagram (figure 4.4), it is essential to bear in mind that there are three questions to answer:

- How do you achieve this function?
- Why do you do this function?
- When you do this function, what other functions must you do?

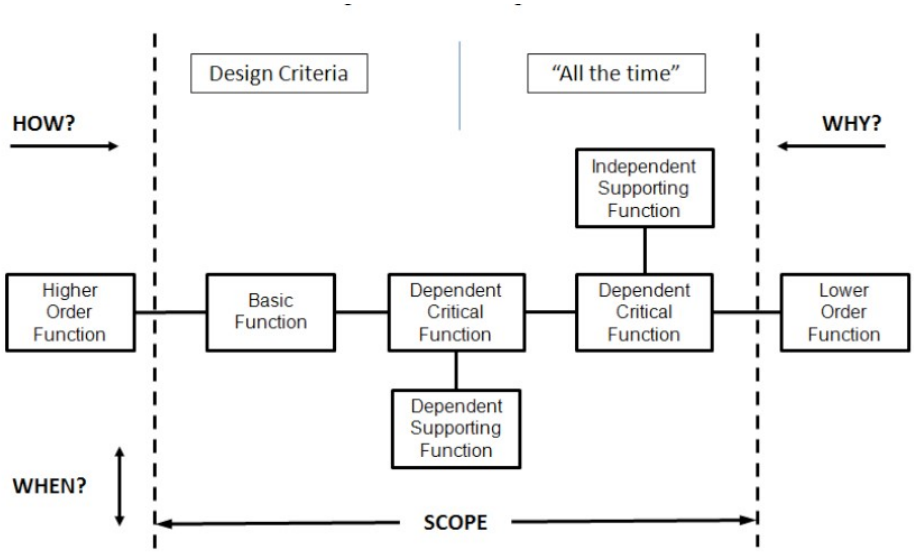

Figure 4.4: FAST Diagram [54]

For the construction of the diagram, there is a set of steps that must be followed:

- Expand the functions in the "How" and "Why" directions.
- Build along the "How" path by asking 'how is the function achieved'? Place the answer to the right in terms of an active verb and measurable noun.
- Test the logic in the direction of the "Why" path (right to left) by asking "why is this function undertaken?".
- When the logic does not work, identify any missing or redundant functions or adjust the order.
- To identify functions that happen simultaneously, ask, "when this function is done, what else is done or caused by the function?".
- The higher-order functions (functions towards the left on the FAST Diagram) describe what is being accomplished, and lower order functions (functions towards the right on the FAST Diagram) describe how they are being.
- "When" does not refer to time as measured by a clock, but functions that occur together with or as a result of each other.

A few things need to be pointed out about FAST diagrams: there is no single "correct" FAST Diagram for a product, process, service or system, as they can vary depending on the focus of analysis or even technology or customer focus. Also, FAST diagrams frame all functions in a positive sense.

4.3.1 FAST Diagram GraphQL Federation

This subsection presents the FAST Diagram (figure 4.5) referring to the federation implementation that will be carried out to understand the main functions to be performed.

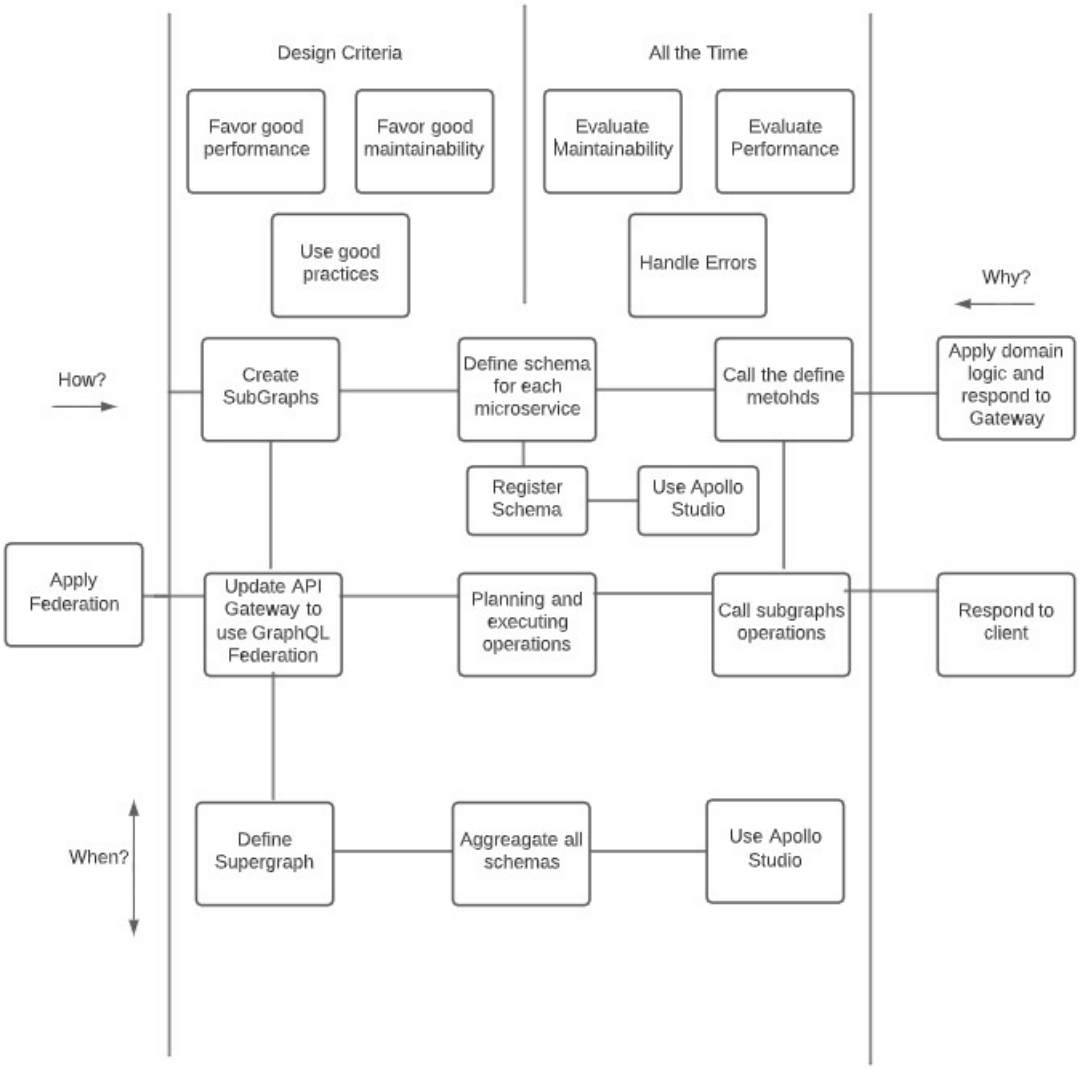

Figure 4.5: FAST Diagram GraphQL

Chapter 5

Design

As mentioned earlier in the objectives, section 1.3, the design and implementation of the solution started from an open-source project that already implemented graphQL and to which federation was applied to understand how this approach affects the performance and maintainability of a solution.

5.1 Selected Project

As mentioned earlier (section 1.4), it was decided to use the public repositories available on Github to select one project.

The main characteristics taken into account when choosing the project were the following:

- Complexity.
- Documentation.
- Tests.
- Technologies.

The chosen project was Payday [56], which is in a public repository on Github and complied with specific requirements defined for its choice. The restriction for this phase is the time available to develop the solution. Therefore, when defining the requirements, the time needed to implement and test the solution was taken into account.

After analyzing many projects, one was selected mainly because of its good documentation, which helps to save time in understanding everything that was previously developed.

Characteristics of the chosen project:

- 4 microservices.
- Well-documented and developed architecture.
- Tests were not developed.
- Technologies: Java, Angular, Javascript, and GraphQL without federation.
- 2 Stars.
- 18 commits.
- created 6 Mar 2020.

5.2 Requirements Analysis

In this section, the solution requirements are defined. This section provides the information necessary to understand the functional and non-functional requirements of the system.

The functional requirements that have been defined were already implemented in the initial solution of the chosen project (without federation). The objective is to implement federation but keep the exact requirements so that it is possible to compare the results of the two solutions later. Table 5.1 identifies and describes the functional requirements, and the figure 5.1 presents the use case diagram.

Table 5.1: Functional Requirements

Identification	**Requirement Description**
FR-01	The user must be able to register into the system.
FR-02	The user must be able to log into the system.
FR-03	The user must be able to search accounts by user, account type and ID.
FR-04	The user must be able to create a new account.
FR-05	The user must be able to disable an account.
FR-06	The user must be able to search for all users.
FR-07	The user must be able to update a user's information.
FR-08	The system must be able to notify the user whenever there is a new transaction.
FR-09	The user must be able to search for all transactions on an account, or transactions by ID.
FR-10	The user must be able to create a new transaction.

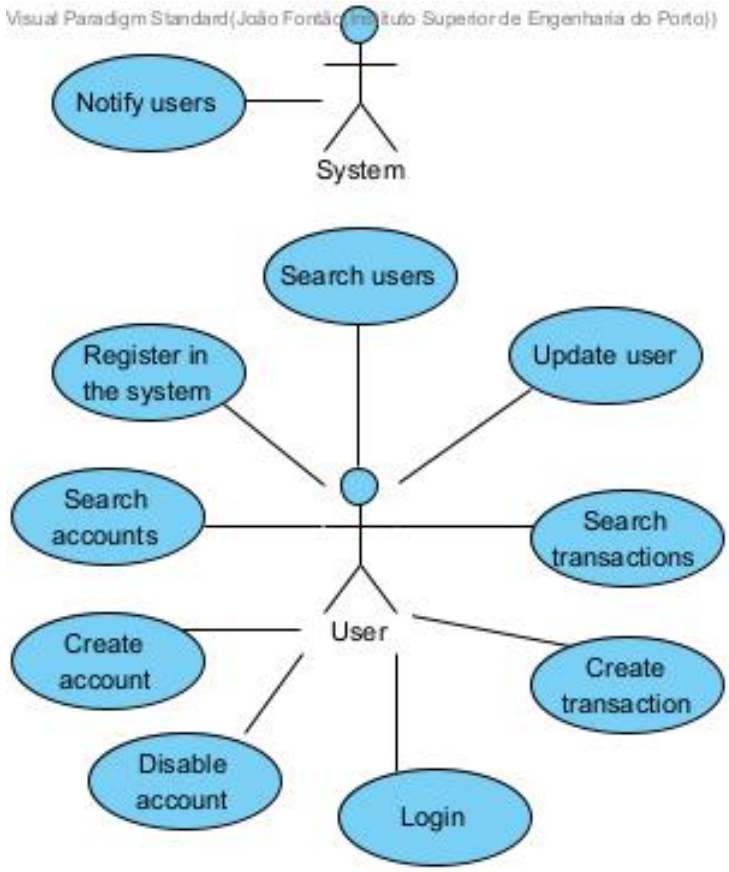

Figure 5.1: Functional requirements

When developing the solution design, we must also consider the non-functional requirements we have to implement the solution because they are essential factors that influence the development of the entire solution. This work focuses on implementing a GraphQL federation architecture with microservices in a solution that already implements GraphQL, so the solution provided must follow this type of architecture's good standards and practices. The necessary tests must also be developed to discard the influence of bad implementation on the result analysis. The table 5.2 presents the non-functional requirements of the system.

Table 5.2: Non-functional Requirements

Identification	**Requirement Description**
NFR-01	Implement GraphQL federation.
NFR-02	The GraphQL federation patterns and best practices must be respected.
NFR-03	The microservices architecture patterns and guidelines must be respected.
NFR-04	Develop the necessary tests.

5.3 Solution Architecture

The architectural solution chosen is a microservices solution with GraphQL using the federation approach developed by Apollo. As explained above, it was chosen to study this approach because it is still a recent approach, so there are still few studies, and it is seen as an excellent alternative to previously existing approaches when using a microservices architecture. Therefore, a solution was designed using the apollo federation architecture to understand its effects on performance and maintainability.

This section initially explains the application and the main components with which the system communicates. Subsequently, the initial solution of the open-source project will be presented before applying the federation, and then the architecture of the solution that will be implemented with federation.

The web application chosen is a small banking application, where the frontend component was developed in angular, and the backend consists of four microservices, three of them created using spring-boot and the other developed in Python. The gateway was developed using Node.js. From a more external and general viewpoint, the application can be accessed through the browser and uses an external database to store data (Oracle and Mongo) and an email service (SendGrid). From the point of view of applications external to the systems, the only change made about the solution initially chosen is that another external application will be used, Apollo Studio, which will help manage federated schemas (figure 5.2).

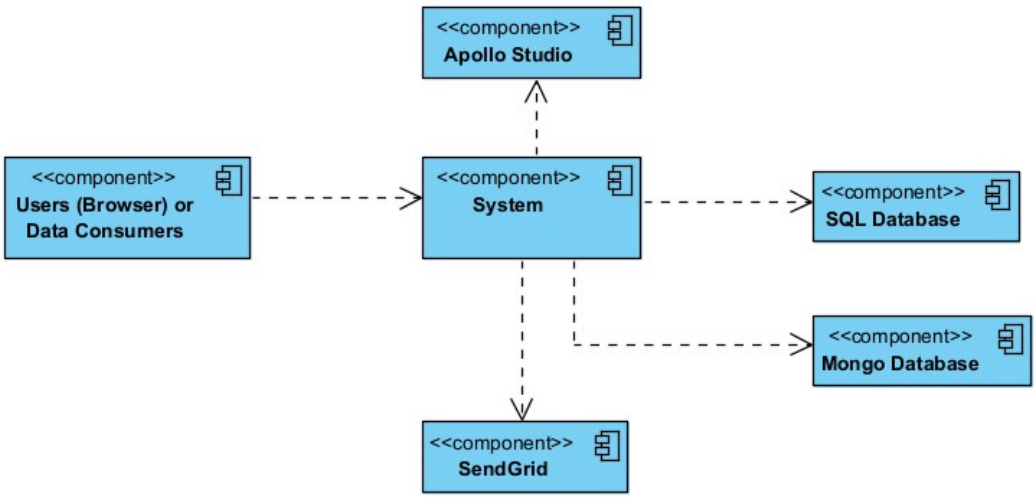

Figure 5.2: Solution Overview

5.3.1 Initial Solution

In this section, the architecture of the initially chosen application to which federation will be applied later is presented generally. Each component of the solution is presented with a short explanation of responsibilities. The initial solution has six essential components (figure 5.3): the client-app, which is the frontend application, the entry service, which serves as the gateway and the four microservices, notifications, transactions, accounts and customers, and uses some components outside the system.

In this initial solution, GraphQL is only used in the component that serves as gateway (Entry), which exposes the schema that the client-app will use to make requests and is also responsible for containing the schema resolvers. After receiving the request, it will forward requests to the respective microservices. The remaining components, in this case, the microservices, do not yet have any GraphQL implementation, and in the new solution, this will change with each microservice using GraphQL.

- **Client-app:** represents the front end of the application. It is responsible for presenting an interface to the user and sending requests to the GraphQL endpoint provided by the entry service (gateway).
- **Entry:** responsibility to receive all requests from the client application or other applications and forward them to the respective microservices. Also, have to send the response to the client application. This component also implements all the resolvers for each of the microservices, and these resolvers will call the microservices endpoints.
- **Accounts:** the microservice is responsible for receiving requests from entry through REST requests, doing its business logic and responding to entry. Also, have the responsibility to create accounts for a user, list the respective ones and disable them.
- **Customers:** the microservice is responsible for receiving requests from entry through REST requests, doing its business logic and responding to entry. Also, have the responsibility to create new users, list all users, get only one user, update users, remove them and also is responsible for the login process.
- **Transactions:** the microservice is responsible for receiving requests from entry through REST requests, doing its business logic and responding to entry. Also, have the responsibility to create the transactions and list them as history for each account.
- **Notifications:** the microservice is responsible for receiving requests from entry through REST requests, doing its business logic and responding to entry. Also, have the responsibility to send the email to notify when there was a transaction. This notification microservice uses SendGrid to send the emails.
- **Oracle Database:** responsibility for keeping account and customer data.
- **Mongo Database:** responsibility for storing transaction data.
- **SendGrid:** responsibility for sending the emails.

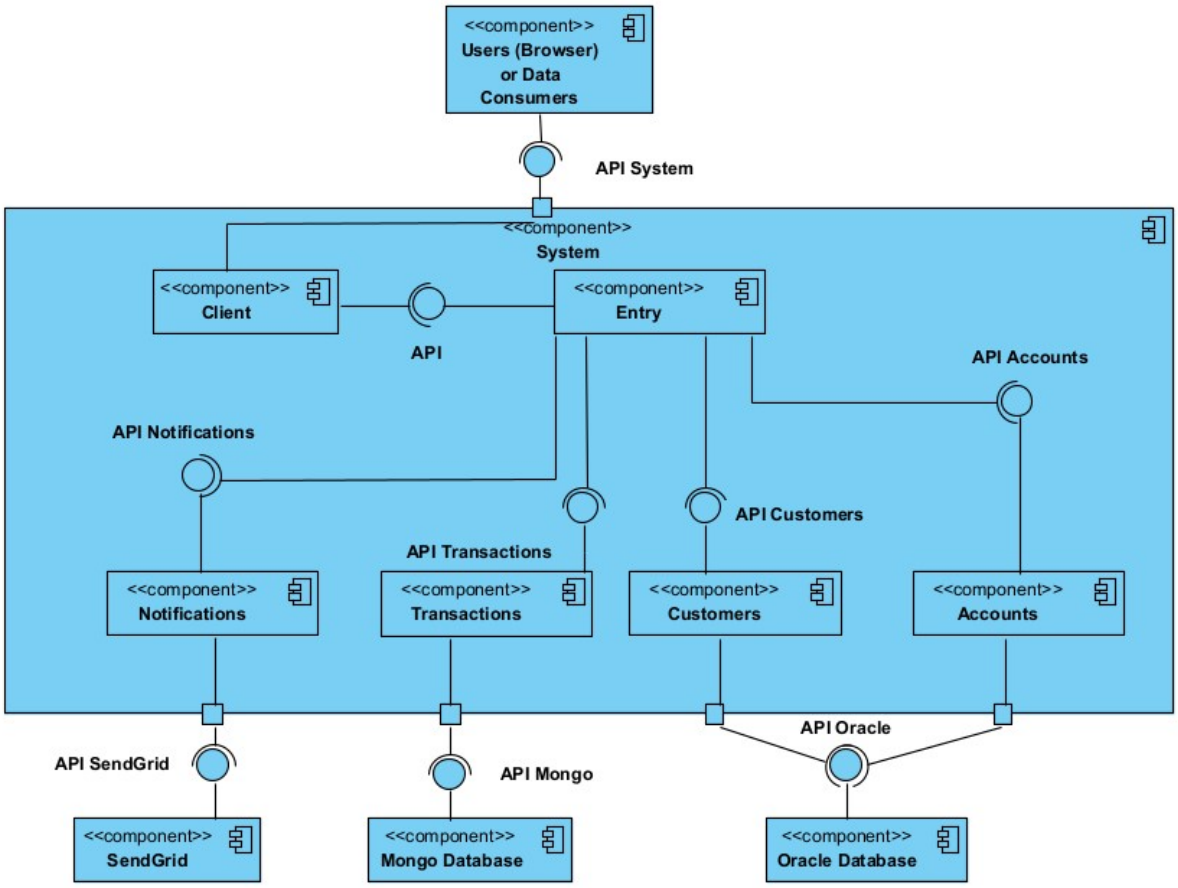

Figure 5.3: Initial Architecture

Doing now for a more detailed analysis of how the components are implemented. Account, customer and transaction microservices are divided into the controller, model and repository. The controller is responsible for receiving requests made to the microservice and executing the logic associated with the microservice. The model has domain classes, and the repository is responsible for data persistence and obtaining data from the database when requested by the controller.

The entry has a package for each microservices with the schemas' definition and the resolvers' implementation that will make the microservices call to perform the necessary logic.

5.3.2 Solution with Federation Architecture

In this section, the Apollo Federation architecture is applied to the previously chosen application, which was previously detailed. Each of the components parts of the new solution is also presented. Finally, the example design of one of the microservices is presented to understand better how the solution is organized.

Like the initial solution, the federation solution has the same six components but is implemented differently (figure 5.4). This section presents and explains only the components that have changed with the federation implementation.

One difference between this solution (figure 5.3) and the solution (figure 5.4) is that in the previous solution, only the gateway used GraphQL, and in this new solution, the gateway will still use GraphQL but implemented differently. The entry no longer has the resolvers' logic and starts to have a superschema that is responsible for the aggregation of all schemas.

Another difference is that the microservices will change, all of them starting to implement GraphQL, each having its schema and the resolvers responsible for carrying out the microservice logic.

Finally, a new component external to the system will be used, Apollo Studio, which is used to help the gateway manage the federated schemas.

- **Entry:** responsibility to receive all requests from the client application or other applications and forward them to the respective microservices. Also, it has to send the response to the client application. With the implementation of GraphQL federation, this component is no longer responsible for having the logic of the resolvers of each microservice and is now responsible for aggregating the microservices schemas. This aggregation and management are done by Apollo Studio. He should also be responsible with the help of Apollo Studio for knowing how to forward requests to the respective microservices.
- **Accounts:** responsibility to create accounts for a user, list the respective ones and disable them. With the implementation of GraphQL federation, it now had its schema definition and became responsible for implementing its respective resolvers and managing the schema.
- **Customers:** responsibility to create new users, list all users, get only one user, update users, remove them and also responsible for the login process. With the implementation of GraphQL federation, it now had its schema definition and became responsible for implementing its respective resolvers and managing the schema.
- **Transactions:** responsibility to create the transactions and list them as history for each account. With the implementation of GraphQL federation, it now had its schema definition and became responsible for implementing its respective resolvers and managing the schema.
- **Apollo Studio:** responsible for managing federated schemas. It also validates schemas when there are changes. It has access to all microservices schemas (subgraphs) and keeps the API Gateway supergraph up to date.

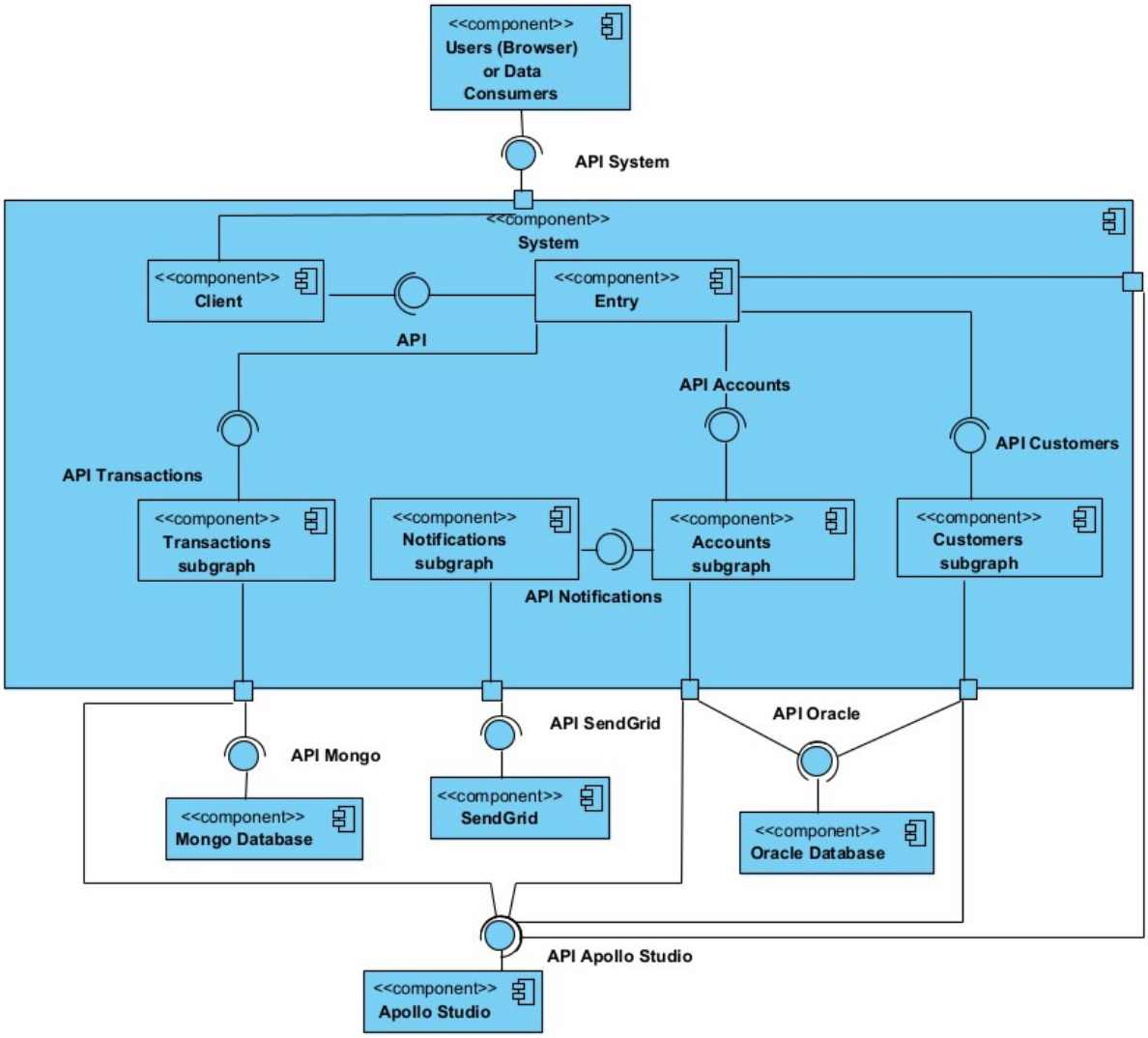

Figure 5.4: Solution Architectural

After showing a diagram of the general architecture of the solution, with the internal and external components that make up the developed application, and explaining the responsibility of each one, it is also necessary to show an example of how the microservices are organized. Only the organization diagram of the accounts microservice will be represented because the organization of the transactions and customers microservices are similar, and the remaining components are simple. There is no need to proceed with their representation through a diagram.

Microservices will consist of five main packages (figure 5.5) :

- **Service:** responsible for executing the necessary business logic for the microservice.
- **DataFetchers:** responsible for returning data for a query or mutation and populating the schema data by calling the services to get the necessary data.
- **DataLoaders:** responsible for mapping fields of external entities. Being used to encapsulate fetching data from a particular source.

- **Model:** contains the domain classes, and it is also responsible for making the necessary mappings to domain classes.
- **Repository:** responsible for data persistence in the respective database.
- **Schema:** contains the microservice schema.

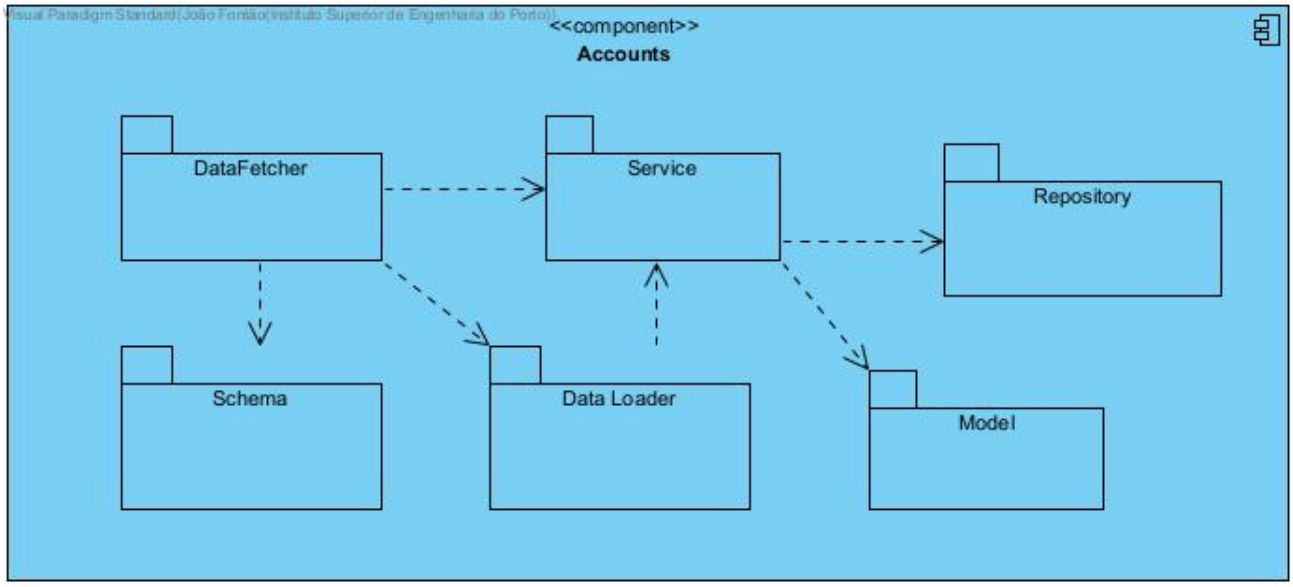

Figure 5.5: Microservice Accounts

Spring Data is used for this microservice and for the customer microservice to access the database, which uses CRUD repositories that automatically generate the main methods to interact with the database for each entity. Regarding the resolvers, they receive the requests they have to resolve and call the respective service, which is responsible for carrying out the defined logic and calling the repository when needed. The service calls whenever it needs classes responsible for mapping objectives into domain classes. For the transactions microservice, the only change that happens is that the access to the database is performed using the MongoRepository.

Chapter 6

Implementation

The main focus of this chapter is to explain how the implementation of the solution was carried out by applying GraphQL federation. It starts with a brief explanation of the initial solution, where the most relevant aspects are mentioned, such as the technologies used, how to run the microservices, and the main characteristics of the components' implementation. Finally, the implementation is explained and how the final solution with GraphQL was reached.

6.1 Initial Solution Implementation

This section begins by referring to the main technologies used in the initial solution and then a description of its implementation.

Table 6.1: Initial Solution - Components

Component	**Technology**	**GraphQL or REST**	**How to Run**
Customers-service	Java - Spring Boot	REST	mvn spring-boot:run
Accounts-service	Java - Spring Boot	REST	mvn spring-boot:run
Transactions-service	Java - Spring Boot	REST	mvn spring-boot:run
Notifications-service	Python - Flask	REST	python ./server.py
Entry-service (Gateway)	JavaScript - Apollo	GraphQL	npm run start:ts
Client-service	JavaScript - React	Use GraphQL for request	yarn start

Table 6.2: Initial Solution - Database Technology

Database	**Component that use**
Mysql-database	Customer-service and Accounts-service
Mongodb	Transactions-service

As we can see in the tables 6.1 and 6.2, the microservices customers, accounts, and transactions are implemented in Java with REST, so any request made to this microservice is made through a

request REST (GET, POST, PUT), with no implementation of GraphQL in the microservices of the initial solution.

Figure 6.1 shows the flow of a login request in the initial solution. The consumer makes a POST (GraphQL) request with the query to be executed and the expected output. The request is received at the gateway, and the customer's resolver will handle the request. After receiving the information, the resolver will perform a GET (REST) request to the customer's microservice to validate the login.

The microservice receives the request through the controller and will validate the password and username using the repository class, and if everything is correct, the customer who logged in will be returned to the gateway.

Now that the user is validated, the gateway through JWT and a secret key will generate a token that will be returned in the response.

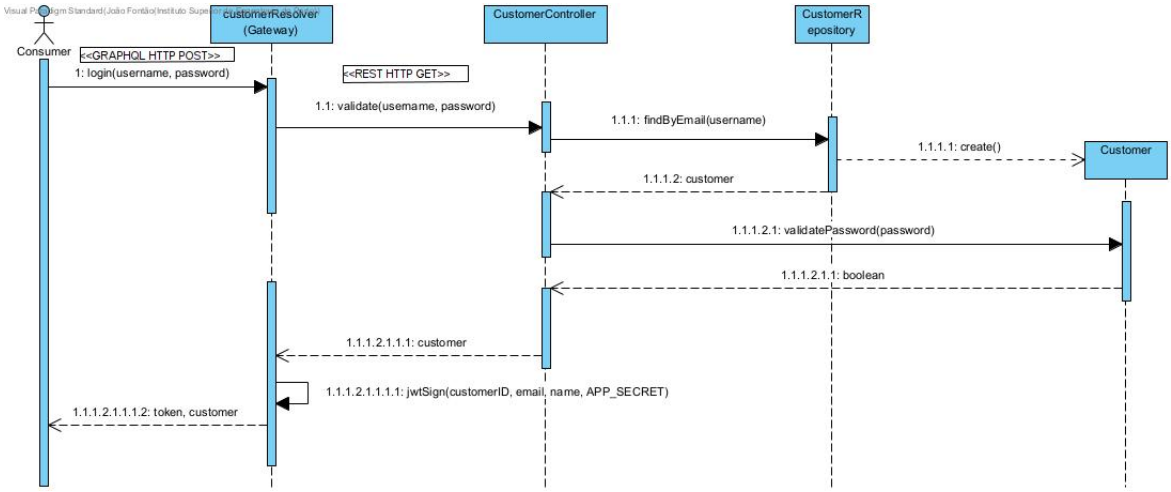

Figure 6.1: Request Example Initial Solution

The possible requests to make to each microservice can be seen through the Controller class (listing 6.1) of each of the microservices, where it is possible to understand the requests that can be made through the defined methods.

```
@RestController
@RequestMapping(path="/api/v1")
public class AccountController {
  @Autowired
  private AccountRepository accountRepository;

  @GetMapping(path="/accounts/customer/{customer}/type/{type}")
  public ResponseEntity<ApiResponse> getAllAccounts(@PathVariable("
    customer") int customer, @PathVariable("type") String type) {
     try {
       List<Account> accounts = new ArrayList<Account>();
       accountRepository.findByCustomerAndTypeAndStatusOrderByIdDesc(
    customer, type, 1).forEach(accounts::add);

       if (accounts.isEmpty()) {
         ApiResponse response = new ApiResponse(accounts, null);
         response.addError("Empty list.");
         return new ResponseEntity<>(response, HttpStatus.OK);
       }

       return new ResponseEntity<>(new ApiResponse(accounts, null),
    HttpStatus.OK);

     } catch (Exception e) {
       return new ResponseEntity<>(new ApiResponse(null, null),
    HttpStatus.INTERNAL_SERVER_ERROR);
     }

  }
}
```

Listing 6.1: Account Controller

The entry service is the only component that applies GraphQL. This component was developed in Node.js and had the function of forwarding received requests to the respective microservices.

Moving on to its implementation, this component for each of the microservices to which it has to forward requests has a defined schema (listing 6.2), with defined queries, mutations, and types, and that is what determines the types of requests that any application can perform. Then we also have the resolvers implemented (listing 6.3) for each of the operations. These resolvers allow us to forward requests to the respective microservices.

The gateway had a package for each microservice where each of the schemas and their respective resolvers was defined, which brings a lot of complexity to a component that shouldn't have any logic associated.

```
export default `
type Account {
    id: ID!
    customer: Int!
    accn: String!
    name: String!
    type: String!
    date_of_creation: Int!
    status: Int!
    transactions: [Transaction]
}

extend type Query {
    accounts(type: String!): [Account]
}
extend type Mutation {
    createAccount(name: String!, type: String!): Account
    disableAccount(id: ID!): Account
}
`;
```

Listing 6.2: Schema Account

```
export default {
  types,
  resolvers: {

    Query: {
      accounts: async (root, args, context) => {
        const userId = getUserId(context);
        let { host, port } = context.config.services.accounts;
        const URL = `${host}:${port}/api/v1/accounts/customer/${userId
   }/type/${args.type}`;

        let accounts = []

        const res      = await axios.get(URL);
        const response = res.data;

        if (response.data && response.data.length) {
          accounts = response.data;
        } if (response.errors) {
          // throw new Error(response.errors[0]);
        }

        return accounts;
      }

}
```

Listing 6.3: Resolver Account

Finally, a brief description of the client app, which refers to the application's front-end. It was implemented in React that uses javascript and the Apollo libraries to make GraphQL requests to the gateway.

This component makes all requests to the gateway to obtain the necessary information. These requests are made through the operations defined in the schemas. For this, a GraphQL request is sent in the correct format with the information that is considered relevant (listing 6.4). The client application also uses JWT to do the authentication part to find out if a particular user has access to a specific resource.

```
const httpLink = createHttpLink({
  uri: 'http://localhost:4000',
});

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem('AUTH_TOKEN');

  return {
    headers: {
      ...headers,
      authorization: token ? `Bearer ${token}` : "",
    }
  }
});

const client = new ApolloClient({
  link: authLink.concat(httpLink),
  cache: new InMemoryCache()
});

let AppContext = React.createContext();

const MUTATION_LOGIN = gql`
  mutation Login($login: String!, $password: String!) {
    login(login: $login, password: $password) {
      customer {
        id
      }
      token
    }
  }
`;
```

Listing 6.4: Client App

6.2 GraphQL Federation Solution Implementation

This section explains how GraphQL federation was implemented in the initial solution that was explained above (section 6.1).

Table 6.3: GraphQL Federation Solution - Components

Component	**Technology**	**GraphQL or REST**	**How to Run**
Customers-service	Java - Spring Boot - DGS Framework	GraphQL	mvn spring-boot:run
Accounts-service	Java - Spring Boot - DGS Framework	GraphQL	mvn spring-boot:run
Transactions-service	Java - Spring Boot - DGS Framework	GraphQL	mvn spring-boot:run
Notifications-service	Python - Flask	REST	python ./server.py
Entry-service (Gateway)	JavaScript - Apollo	GraphQL	npm run server
Client-service	JavaScript - React	Use GraphQL for request	yarn start

Comparing this table 6.3 and the table 6.1, from a more general view, we can see that the main changes occurred at the level of microservices (Customers, Accounts, and Transactions) where REST was stopped, and GraphQL started to be used, for the implementation of GraphQL federation the DGS framework was used. Regarding the Notifications microservice, it was decided not to implement GraphQL because it is such a simple microservice that it only serves to send emails. There is no difference in the gateway level tables because the same technologies were used but implemented in different ways to support federation.

The sequence diagram of figure 6.2 shows the flow of a login request to the new solution using GraphQL federation. The consumer makes a POST (GRAPHQL) request with the query to be executed and the expected output. The request is received at the gateway and forwarded to the respective microservice that will handle the request.

The microservice receives the request through the defined resolver. This calls the service to validate the password and username using the repository class. Once validated, the resolver calls the JWTUtils class that will generate a token using the customer's secret key and id. To finish the resolver, build the object that will be returned with the information requested by the user.

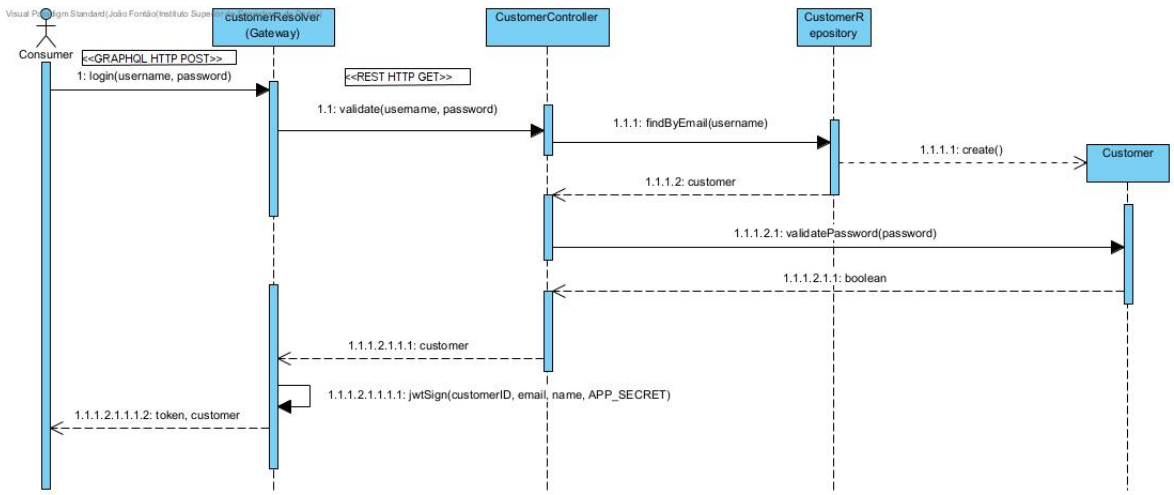

Figure 6.2: Request Example Final Solution

In these developments to apply federation, the Single Responsibility Principle was used ("In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility" [57]), which means that a class must always have only one responsibility.

This principle was applied throughout the project, mainly in microservices where each class has only one responsibility. The data fetcher will only receive the requests, and the data loaders are responsible for mapping fields of external entities, the services contain all the business logic, and the repository is responsible for accessing the database.

The Controller pattern was also applied, referring to "a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation" [58]. In each microservice, a class is defined that is responsible for handling system events (data fetcher). This class has implemented resolvers that will handle requests made by other client applications. For example, to login or get data from a customer, only one class will receive these two requests, and then this class will call the others responsible for carrying out the necessary logic.

6.2.1 Gateway

Some libraries use to implement federation:

- **apollo/gateway**: the core class of Apollo Server's federated gateway implementation.
- **apollo-server**: it's use to build a production-ready, self-documenting GraphQL API that can use data from any source.
- **dotenv**: loads environment variables from ".env" file. Used to connect to Apollo Studio through the keys defined in the ".env" file.
- **jsonwebtoken**: for managing the authentication (tokens).

In this new solution implementation with GraphQL federation, one of the significant changes was at the gateway level, which previously had much logic associated with the resolvers, becoming very

complex. However, in this new solution, the gateway only has the primary function of forwarding requests to the respective microservices (no longer having any resolvers) (listing 6.5). The gateway will receive the GraphQL request, and depending on the received request, it will request the information from the respective microservices and then return the response.

```
const gateway = new ApolloGateway({
    buildService({ name, url }) {
        return new RemoteGraphQLDataSource({
            url,
            willSendRequest({ request, context }) {
                request.http.headers.set(
                    "user",
                    context.user ? JSON.stringify(context.user.sub) :
    null
                );
            }
        });
    }
});

const server = new ApolloServer({
    gateway,
    subscriptions: false,
    context: ({ req }) => {
        const user = req.auth || null;
        return { user };
    }
});
```

Listing 6.5: Gateway Federation

For the gateway to forward requests and build the supergraph schema, which is the aggregation of all the subgraph schemas of the microservices, Apollo Studio was used to help manage this schema.

As explained before (subsection 5.3.2), Apollo Studio will aggregate all the subgraph schemas of the microservices and build a single supergraph schema that the gateway will be able to access and make available to clients (listing 6.7). To obtain this supergraph schema, it is only necessary to define the Apollo Studio API access key in a ".env" file that the apollo gateway framework will use by default when services are not defined (listing 6.6).

```
APOLLO_KEY=service:TMDEI-Graph:RI3ejXMHP-UMsp97nFEgpg
APOLLO_GRAPH_REF=TMDEI-Graph@current
```

Listing 6.6: Configuration Apollo Studio

```
directive @apollo_studio_metadata(launchId: String, buildId: String,
    checkId: String) on SCHEMA

type Account {
  id: ID!
  customer: Int!
  accn: String!
  name: String!
  type: String!
  date_of_creation: Int!
  status: Int!
  transactions: [Transaction]
}

type AuthPayload {
  token: String
  customer: Customer
}

type Customer {
  id: ID!
  name: String!
  last_name: String!
  phone: String!
  email: String!
  gender: String!
  date_of_birth: String!
  debitAccounts: [Account]
  creditAccounts: [Account]
}

type Mutation {
  createTransaction(account: ID!, amount: Int!, description: String):
    Transaction
  signup(name: String!, last_name: String!, password: String!, email:
    String!, gender: String!, phone: String!, date_of_birth: String!):
    AuthPayload
  login(login: String!, password: String!): AuthPayload
  createAccount(name: String!, type: String!): Account
  disableAccount(id: ID!): Account
}

type Query {
  transactions(account: ID!): [Transaction]
  customers: [Customer]
  customer: Customer
  accounts(type: String!): [Account]
}

type Transaction {
  id: ID!
  customer: Int!
  account: Int!
  description: String
  date_of_transaction: String
  amount: Int!
}
```

Listing 6.7: API Schema

The other function of the gateway is to help manage authentication because when the user login, the request will return a token that will then be used to prove that the user is authenticated in the system. The gateway, when receiving a request, will check the token and see if it is valid (using the secret key) and at the same time get the user id that is encrypted in the token because in specific requests (listing 6.8), will forward this id as a header to be used inside of microservices, to identify the user who made the request.

```
app.use(
    expressjwt.expressjwt({
        secret: "f1BtnWgD3VKY",
        algorithms: ["HS256"],
        credentialsRequired: false,
        ignoreExpiration: true
    })
);

const gateway = new ApolloGateway({
    buildService({ name, url }) {
        return new RemoteGraphQLDataSource({
            url,
            willSendRequest({ request, context }) {
                request.http.headers.set(
                    "user",
                    context.user ? JSON.stringify(context.user.sub) :
    null
                );
            }
        });
    }
});
```

Listing 6.8: Gateway Authentication

6.2.2 Microservices

Some libraries and plugins use to implement federation:

- **graphql-dgs-spring-boot-starter**: DGS Framework Dependency allows the use of DGS components.
- **graphql-dgs-client**: used in tests to perform GraphQL requests.
- **graphql-dgs-mocking**: used to perform unit tests with mocks.
- **jjwt**: for managing the authentication (tokens).
- **graphqlcodegen-maven-plugin**: used to generate constructors for entities defined in schemas.

With the implementation of GraphQL federation, microservices stopped using REST and started using GraphQL. The DGS framework was used to help implement federation in microservices developed in Java. The use of the DGS framework was the big technological difference in the microservices because the technologies already used in the initial solution were maintained to have the least possible changes, thus trying not to influence the final results.

For each microservices, it was necessary to create a new schema (subschema GraphQL) to support the operations existing in the initial solution. In the implementation, it was essential to look at the existing schema and divide it for each of the respective microservices, making the necessary changes to support federation.

In the schema files (".grapghqls"), the types for the respective microservices and also the possible operations (queries and mutations) were defined. As we are implementing federation, when we think about the schemas that we will have, and we realize that it will be necessary to have a type that will be used in another microservice, the type has to be annotated as an entity, and that happens by putting the "@key" and defining the key. With these annotations, it becomes possible to make extends of this type (listing 6.9).

```
type Customer @key(fields: "id") {
    id: ID!
    name: String!
    last_name: String!
    phone: String!
    email: String!
    gender: String!
    date_of_birth: String!
}

type AuthPayload {
    token: String
    customer: Customer
}

type Query {
    customers: [Customer]
    customer: Customer
}

type Mutation {
    signup(name: String!, last_name: String!, password: String!, email:
     String!, gender: String!, phone: String!, date_of_birth: String!):
    AuthPayload
    login(login: String!, password: String!): AuthPayload
}
```

Listing 6.9: Customer schema

On the other hand, to use an entity from another microservice in a schema, it was necessary (listing 6.10):

- Represent that entity.
- Add the annotation "@key" and"@extends".
- In the key (which is always the ID in this case) put the "@ external".
- Defined the new properties we want to add to that microservice.

In these cases, the fields defined in the initial schema are not defined again because when there is the aggregation of the subschemas, everything will be together, as shown in the listing 6.11.

```
type Account @key(fields: "id") {
    id: ID!
    customer: Int!
    accn: String!
    name: String!
    type: String!
    date_of_creation: Int!
    status: Int!
}

type Customer @key(fields: "id") @extends {
    id: ID! @external
    debitAccounts: [Account]
    creditAccounts: [Account]
}

type Mutation {
    createAccount(name: String!, type: String!): Account
    disableAccount(id: ID!): Account
}

type Query {
    accounts(type: String!): [Account]
}
```

Listing 6.10: Account schema

```
type Customer {
  id: ID!
  name: String!
  last_name: String!
  phone: String!
  email: String!
  gender: String!
  date_of_birth: String!
  debitAccounts: [Account]
  creditAccounts: [Account]
}
```

Listing 6.11: Api Schema Customer

In terms of implementation, when we create an external entity represented in another microservice, we define in the class DataFetchers a method with the annotation of "@DGSEntityFetcher" and with the entity's name, where we encode how the entity is created. For the fields that we add to this entity, we define a method for each one, with the annotation "@DGSData", and the parent type, which is the entity name which the field name refers in the schema. This method has the logic of getting the information we want to return (listing 6.12).

In the federation solution to contain the logic of filling the fields, a class called DataLoaders was created (one for each property). This class must have the annotation "@DGSDataLoader" and make the calls to the methods necessary to obtain the data (listing 6.13).

```
@DgsEntityFetcher(name = "Customer")
public Customer game(Map<String, Object> values) {
    String id = String.valueOf(Long.parseLong((String) values.get("id")
    ));
    return Customer.newBuilder().id(id).build();
}

@DgsData(parentType = "Customer", field = "debitAccounts")
public CompletableFuture<Account> accountDebitByCustomer(
    DgsDataFetchingEnvironment dfe) {
    DataLoader<String, Account> accountByCustomerDataLoader = dfe.
    getDataLoader(AccountsDebitByCutomerDataLoader.class);

    Customer customer = dfe.getSource();

    return accountByCustomerDataLoader.load(customer.getId());
}
```

Listing 6.12: External Entity Customer

```
@DgsDataLoader(name = "ACCOUNTS_CREDIT_FOR_CUSTOMERS")
@RequiredArgsConstructor
public class AccountsCreditByCutomerDataLoader implements
    MappedBatchLoader<String, List<Account>> {
    private final AccountService accountService;

    @Override
    public CompletionStage<Map<String, List<Account>>> load(Set<String>
    customerIds) {
        return CompletableFuture.supplyAsync(() -> this.accountService.
    getAccountsByCustomerAndTypeAndStatus(new ArrayList<>(customerIds),
    "credit", 1));
    }
}
```

Listing 6.13: DataLoader Account Customer

Schemas in Java microservices (using the DGS framework) must be defined in the folder "resources/schema/", so it is unnecessary to write the path to the schema in the properties file as it is the default path.

In the new solution, the microservices no longer have controllers, as they were only used to receive REST requests and later call the methods responsible for obtaining the data. To replace the controllers, we now have resolvers containing GraphQL operations' logic defined in the schema (queries and mutations).

A class (DataFetcher) was created to place the implementation of the resolvers. This class must have the annotation "@DGSComponent" to be recognized by the DGS framework. In this class, to define how to handle mutations and queries, the DGS framework forces us to have annotations "@DGSMutation" or "@DGSQuery" depending on the type of operation. Also, the name must be the same as that defined in the schema (listing 6.14).

```
@DgsComponent
@RequiredArgsConstructor
public class AccountDataFetcher {
    private final AccountService accountService;

    @DgsQuery
    public List<Account> accounts(@InputArgument String type,
    @RequestHeader("user") String user) {
        Integer customer = Integer.parseInt(user.replaceAll("\"",""));
        return this.accountService.getAllAccounts(customer, type);
    }

    @DgsMutation
    public Account createAccount(@InputArgument String name ,
    @InputArgument String type, @RequestHeader("user") String user){
        Integer customer = Integer.parseInt(user.replaceAll("\"",""));
        return this.accountService.addNewAccount(new Account(customer,
    name, type), user);
    }
}
```

Listing 6.14: DataFetcher Account

Another change was the generation of the authentication token, which is no longer done at the gateway and is now performed in the customer's microservice, which is responsible for validating the user's login and generating the token sent in response. Authentication was implemented in the microservice using the JJWT library that allows us to generate a token from a secret key (it was decided to use the same secret key defined in the initial solution, one difference is that the secret in Java has to be encoded in base 64).

The generation of the token was one of the significant challenges encountered, and to be able to perform this functionality, it was necessary to create a JwtUtil class with utility methods, such as generating the token or even its validation (listing 6.15). It was also essential to create another class that extended the WebSecurityConfigurerAdapter, to be able to override some methods that were used in the generation of the token. Finally, a class was created to implement the UserDetailsService interface to override the loadUserByUsername method and get user information from database (using the repository class) (listing 6.16).

```
@Service
public class JwtUtil {
    private String SECRET_KEY = "ZjFCdG5XZ0QzVktZ";

      public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, userDetails.getUsername());
    }

    private String createToken(Map<String, Object> claims, String
    subject) {
        System.out.println("Subject: " + subject);
        byte[] decodedSecret = Base64.getDecoder().decode(SECRET_KEY);
        String decoded = Jwts.builder().setHeaderParam("typ", "JWT").
    setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.
    currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() +
    1000 * 60 * 60 * 10))
                .signWith(SignatureAlgorithm.HS256, decodedSecret).
    compact();
        System.out.println(decoded);
        return Jwts.builder().setHeaderParam("typ", "JWT").setClaims(
    claims).setSubject(subject).setIssuedAt(new Date(System.
    currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() +
    1000 * 60 * 60 * 10))
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact
    ();
    }
}
```

Listing 6.15: JWUtil Class

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    CustomerRepository jwtUserRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws
    UsernameNotFoundException {
        Optional<Customer> jwtUser = jwtUserRepository.findByEmail(
    email);
        if (jwtUser == null || jwtUser.isEmpty() || !jwtUser.isPresent
    ()) {
            throw new UsernameNotFoundException("email Not found" +
    email);
        }
        return new User(jwtUser.get().getEmail(), jwtUser.get().
    getPassword(), new ArrayList<>());
    }
}
```

Listing 6.16: Custom User Details Service Class

After all the logic is implemented, it is only necessary to call the method to generate a token with the user's id (so that we can later identify the user who made the request) and return the object (type defined in the schema) with the token and the user that logged in (listing 6.17).

```
@DgsComponent
public class CustomerDataFetcher {

    private final JwtUtil jwtUtil;

    private final AuthenticationManager authenticationManager;

    private final CustomerService customerService;

    @DgsMutation
    public AuthPayload login(@InputArgument String login,
    @InputArgument String password){
        Customer customer = this.customerService.validateCustomer(login
    , password);
        UsernamePasswordAuthenticationToken uspas = new
    UsernamePasswordAuthenticationToken(login, password);
        Authentication authentication = authenticationManager.
    authenticate(uspas);
        SecurityContextHolder.getContext().setAuthentication(
    authentication);
        UserDetails userDetails = new User(customer.getId().toString(),
     password, new ArrayList<>());

        final String jwt = jwtUtil.generateToken(userDetails);
        return AuthPayload.newBuilder().token(jwt).customer(bank.payday
    .customers.generated.types.Customer.newBuilder().email(customer.
    getEmail()).id(customer.getId().toString()).gender(customer.
    getGender()).date_of_birth(customer.getDate_of_birth()).name(
    customer.getName()).last_name(customer.getLast_name()).phone(
    customer.getPhone()).build()).build();
    }
}
```

Listing 6.17: DataFetcher Customer Login

Finally, one of the significant innovations in this project was the use of Apollo Studio, and it has been explained previously how the gateway communicates with Apollo Studio to obtain the super-schema (subsection 6.2.1). Regarding microservices, this is very simple because what is done is the microservice publishes its schema in Apollo Studio, which then the rest of the management is done by the application itself.

To publish the schema, it is only necessary to run the command (**"rover subgraph publish TMDEI-Graph@current –name accounts –schema ./account.graphqls –routing-url http://localhost:9001/graphql"**) and we have to replace required parameters:

- **"–schema":** define the schema we want to publish.
- **"TMDEI-Graph@current":** refers to the name that identifies the workspace created in Apollo Studio.
- **"–name":** defines the name to be given to the schema.

- **"–routing-url":** define the URL to access the microservice.

Chapter 7

Tests and Solution Evaluation

In this chapter, the developed solution will be evaluated in terms of quality attributes, and for this, the GQM (Goal, Question, Metric) method will be used to measure performance and maintainability in quantitative terms because it is widely used to measure the quality attributes referring to the 25010 standards [59]. Subsequently, a qualitative assessment of maintainability will also be carried out.

The hypothesis formulated for this project was defined after having a good understanding of the project, that is, after well exploring the topic of GraphQL and microservices. The hypothesis is that the implementation of GraphQL federation in a microservices architecture improves the quality attributes, performance and maintainability.

In order to try to prove the hypothesis, an open-source project was found that uses GraphQL but without federation, and then federation is applied, and later trying to understand how the performance and maintainability were influenced if the federation improved these quality attributes.

This study will provide work that can help future developers understand how federation can help them when it comes to performance and maintainability.

7.1 Methodology

To evaluate the quality attributes quantitatively, the method GQM will be used. GQM is a method that uses metrics to measure a specific objective in a certain way. The GQM measurement model consists of three levels figure 7.1:

- Conceptual (Goal): a goal is defined for an object, about various quality models from various points of view and to a specific environment.
- Operational (Question): define a set of questions based on the objective, and the questions aim to verify that the objective has been achieved. The questions aim to characterize the object being measured to a given quality problem and establish the quality from the point of view.
- Quantitative (Metric): define a set of metrics that can be objective and subjective so that it is possible to answer each question quantitatively.

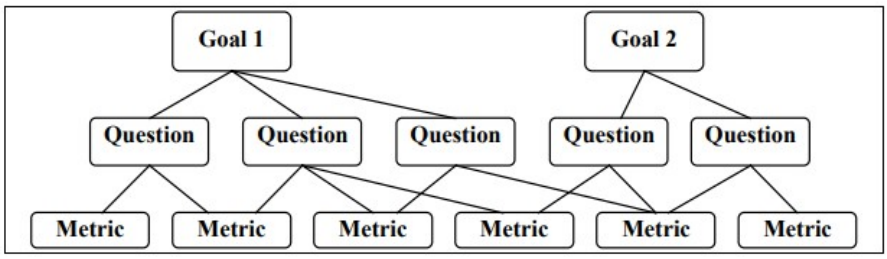

Figure 7.1: GQM [60]

The procedure for applying this method begins with identifying a set of quality goals, and from these goals, the questions are defined. The next step will be to specify the measures (metrics) taken to answer the questions raised. After this entire process and measures are defined, information compilation procedures must be developed, including validation and analysis procedures. This evaluation will realize for each of the individual solutions, and later comparison of the results obtained will be made to understand what the variations were. The same objectives, questions and metrics will be used to evaluate both solutions, allowing for comparing the results obtained with the tools.

Despite this quantitative assessment being able to help us understand the impact of GraphQL federation on these quality attributes, it was also decided to carry out a qualitative evaluation of maintainability because quantitative assessment alone is not enough to assess the maintainability of a solution since it is influenced for the quality of the code developed and for the good practices used. Therefore, it was also decided to conduct a qualitative evaluation because it is more focused on the approaches used and not on how the code was implemented.

To carry out this qualitative analysis of maintainability, two changes will be defined to be carried out in both solutions later, it will be explained what would have to be done in each solution to apply these changes, and finally, it will be possible to understand the difficulty of making the changes. Then it will be possible to compare both solutions and see if there is one that is less difficult to make changes.

7.1.1 Maintainability - Quantitative Evaluation

Table 7.1, was built with the application of the GQM method for the maintainability quality attribute. Although these metrics are defined, they can later change if there is a need to choose other metrics.

Table 7.1: GQM Maintainability.

Goal	Question	Metrics
Evaluate the maintainability of the solution from a software developer's viewpoint.	Is the application easily maintainable?	Average time to fix a line of code (Maintainability Rating).

Tool: SonarQube's.

To evaluate the maintainability, the tool SonarQube will be used, specifically, SonarQube's Maintainability Rating that considers a set of metrics, the main one being the technical debt ratio, which refers to the cost of developing a line of code. It has the advantage of analyzing the whole code and

not just some files. To achieve the objective, it is necessary to evaluate the metrics before and after changing the design to understand if the metric has changed significantly.

To evaluate a project in terms of maintainability, SonarQube has a scale from A to E using the Maintainability Rating, taking into account the value of the Technical Debt Ratio. The best value of the maintainability scale provided by SonarQube is A and the worst is E.

7.1.2 Performance

As in the previous subtopic, table 7.2 is shown below, with the application of the GQM method for the performance quality attribute.

Table 7.2: GQM Performance.

Goal	Question	Metrics
Evaluate the performance of the solution from the software developer's viewpoint.	Is the solution's performance acceptable?	Response time.

Tool: PRTG Network Monitoring.

This tool allows monitoring the network and can help evaluate an application's performance. It is possible to assess the API response time and other metrics by defining sensors.

7.1.3 Maintainability - Qualitative Evaluation

For the qualitative assessment of maintainability, it was decided to analyze two situations that help us understand how both solutions are designed to support changes, thus being able to assess maintainability. The two situations to be evaluated are the following:

- What to do to add new microservice?
- What to do to update a microservice? (Add a mutation)

For each situation, it is analyzed what needs to change in the components (modularity) and how much this can influence the current system (modifiability) to understand if there are maintainability differences between the solutions and if the use of federation has improved the maintainability of the solution.

7.2 Technical quality

The first step in evaluating the solution was to develop some tests to minimally guarantee that the solution results are not affected by bad implementation. The tests were only developed for the final solution (GraphQL federation) since the initial solution was already fully developed and implemented.

The technical evaluation is established through unit tests and tests of the use of the client application. All the client application features were tested to verify that all operations are still available and implemented correctly (table 7.3).

Table 7.3: Tested Features Client-app

Functionality	**Information**
Signup	Register user, entering valid data. To validate login with this user.
Login	Login with the registered user and check if he enters the application.
Create debit account	Reloading the page and verifying the account appears on the page in the correct location.
Create credit account	Reloading the page and verifying the account appears on the page in the correct location.
Create transaction for debit account	Reload the page and validate if the transaction appears in the chosen account.
Create transaction for credit account	Reload the page and validate if the transaction appears in the chosen account.
Logout and login again	Validate that all previously created information appears correctly.

Unit tests were applied to the service classes and data fetchers which are the ones that contain business logic that can cause problems. Unit tests were developed to test the following aspects:

- Return lists or objects with the right data for each attribute.
- Return the custom exception, with the right message.
- Validation of methods inputs.

So that the unit tests are always valid independently of the data that exists in the respective databases, the information was simulated using mocks (Mockito) (listing 7.1).

```
@Test
void getTransactionIdNotExistsTest() {
    String id ="4";
    when(transactionRepository.findById(id)).thenReturn(Optional.empty());
    TransactionException exception = assertThrows(TransactionException.class,
    () -> {
        transactionService.getTransaction(id);
    });
    assertEquals("Invalid Transaction ID", exception.getMessage());
}
```

Listing 7.1: Exemple of a Unit Test

To test the data fetchers classes containing the GraphQL resolvers, the DGSClient was used, which allows sending GraphQL requests and building the necessary objects (listing 7.2).

```
@Test
void transactions() {
    int account =2;
    int customer = 1;
    when(transactionService.getAllCustomerTransactions(account, customer)).
    thenReturn(List.of(transaction, transaction2));
    GraphQLQueryRequest graphQLQueryRequest = new GraphQLQueryRequest(
            new TransactionsGraphQLQuery.Builder().account("2").build(),
            new TransactionsProjectionRoot().description().id().account().
    amount().customer()
    );
    MultiValueMap<String, String> map = new LinkedMultiValueMap<>();
    map.add(HEADER_NAME, "lm24lrtm4ltm4lmt34sdlt");
    HttpHeaders httpHeaders = new HttpHeaders(map);

    List<String> descriptions = dgsQueryExecutor.executeAndExtractJsonPath(
    graphQLQueryRequest.serialize(), "data.transactions[*].description",
    httpHeaders);
    List<String> ids = dgsQueryExecutor.executeAndExtractJsonPath(
    graphQLQueryRequest.serialize(), "data.transactions[*].id", httpHeaders);
    List<Integer> customers = dgsQueryExecutor.executeAndExtractJsonPath(
    graphQLQueryRequest.serialize(), "data.transactions[*].customer",
    httpHeaders);
    List<String> amounts = dgsQueryExecutor.executeAndExtractJsonPath(
    graphQLQueryRequest.serialize(), "data.transactions[*].amount", httpHeaders)
    ;
    List<String> accounts = dgsQueryExecutor.executeAndExtractJsonPath(
    graphQLQueryRequest.serialize(), "data.transactions[*].account", httpHeaders
    );
    assertEquals(descriptions,List.of(description1, description2));
    assertEquals(ids,List.of(String.valueOf(id1), String.valueOf(id2)));
    assertEquals(customers,List.of(customer1, customer1));
    assertEquals(amounts,List.of(amount1, amount2));
    assertEquals(accounts,List.of(account2, account2));
}
```

Listing 7.2: Exemple of a Unit Test Data Fetchers

7.3 Experiments

This section provides an insight into the results of the experimetns performed on each defined quality attributes.

7.3.1 Maintainability - Quantitative Evaluation

After doing the tests that minimally guarantee that the solution has no problems in the implementation and that the features are implemented correctly, move on to the maintainability assessment using the metric: average time to fix a line of code. The SonarQube tool was used to obtain the necessary information, which allows for evaluating the maintainability of applications (figure 7.2). It was decided not to make corrections that SonarQube suggested having the most realistic results possible.

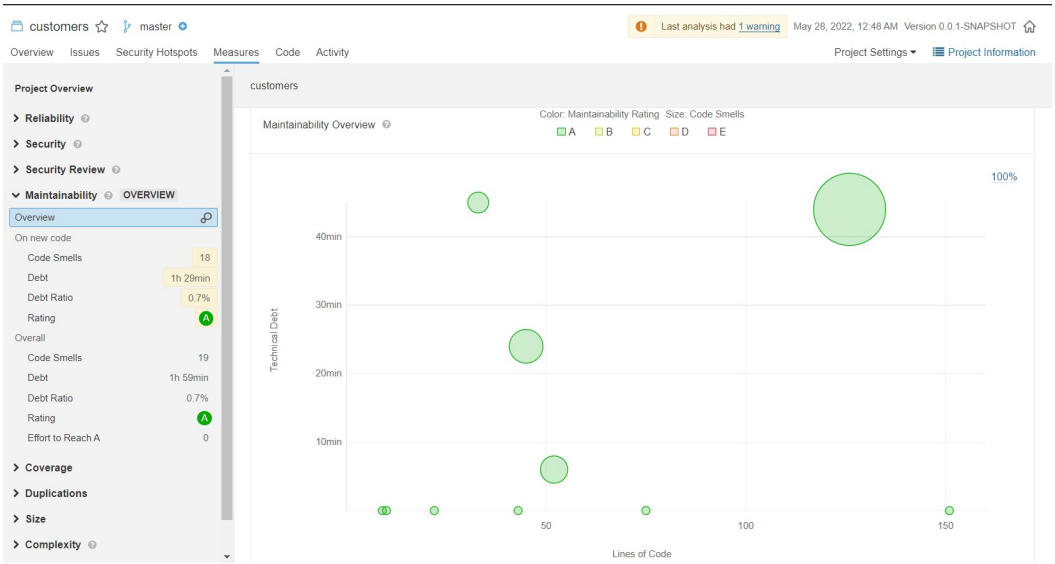

Figure 7.2: SonarQube Overview Customer

SonarQube was installed locally to perform the evaluation, and two projects were created, one for each solution. The component evaluation is done automatically after deploying to the tool using the following command:

- Command: mvn clean verify sonar:sonar -Dsonar.projectKey=KEY -Dsonar.host.url=http://localhost:9000 -Dsonar.login=LOGIN.

Tables 7.4 and 7.5 provides a summary of the analysis. Annex A shows the complete results of the maintainability analysis.

Table 7.4: Results Maintainability Initial Solution

Component	Rating	Debt Ratio
Customers-service	A	1.1%
Accounts-service	A	1.1%
Transactions-service	A	0.3%
Entry-service (Gateway)	A	2.5%

Table 7.5: Results Maintainability GraphQL Federation Solution

Component	Rating	Debt Ratio
Customers-service	A	0.7%
Accounts-service	A	0.3%
Transactions-service	A	0.3%
Entry-service (Gateway)	A	0.8%

Question: Is the application easily maintainable

- The analysis results show that both solutions have all components with a maintainability rating of A, resulting in an average Maintainability Rating of A for the overall proof of concept. With these results, it is possible to say that both solutions are easy to maintain, even because by doing a more in-depth analysis, it is possible to see that most of the existing problems would be easily solved.

Looking now at the Debt Ratio, it is possible to see that the final solution is slightly better, but the difference is minimal, and it is not possible to say that none of the solutions is much better than the other. It is also possible to perceive that these minor differences and even the current debt ratio could be easily corrected because, as it's possible to see in SonarQube, most of the problems found are simple to solve and would not take long to be dealt with. Therefore, it appears that the quantitative assessment of the maintainability of the initial solution has been influenced by a bad implementation, which may have compromised the results obtained (table 7.4).

The most significant difference happens in the gateway component, and this is because, in the initial solution, this component had much logic associated (as explained above), which in the future may cause problems when it is necessary to change the solution because it is a component used by all microservices and will always need to be updated.

The new gateway that applies federation does not have any associated logic, and if there is a growth or change in the microservices, it is not necessary to change this component because it is entirely independent of the microservices. It will only be essential to inform Apollo Studio through the deployment of the schemas.

7.3.2 Performance

The performance was measured for each of the solutions, as mentioned in Evaluation Methodology, and it is evaluated using metrics of time. The PRTG Network tool was used, allowing the creation of sensors to obtain the necessary information. The sensors in this tool allow configuring requests to an API from time to time, recording the average response time of requests (figure 7.3).

Sensors were created for each of the nine operations defined in the schemas of the two solutions. Using the same body was important for the same operations because the amount of information brought is very similar, not negatively influencing the performance tests. For the evaluation of the results, there is the verification of the average time of ten requests to reduce the error that could happen, and the first request was always ignored because it always takes longer.

Figure 7.3: Sensors GraphQL Federation Solution

Table 7.6 provides a summary of the analysis. Annex B shows the complete results of the Performance analysis.

Table 7.6: Results Performance Solutions

Queries	**Initial Solution (ms)**	**Federation Solution (ms)**
Signup	284	301
Login	114	209
Get Customer	39	39
Get Customers	122	61
Create Account	65	71
Disable Account	34	39
Get Accounts by Type	42	35
Create Transation	38	48
Get Transactions by Account	39	24

Question: Is the solution's performance acceptable?

- Observing the results obtained with the tool, it's possible to see that both solutions have response times that it can consider acceptable for most queries, although many have low complexity. But the average response times are very short, thus showing satisfactory performance.

With this information, it is possible to observe that most requests have more or less the same response time, which shows that the GraphQL federation solution does not seem to bring significant advantages in terms of performance.

Two cases stand out in this table. The first is in the login query, where there is a big difference in times, which can be justified by the difference in implementation that exists to obtain the login token, in the initial solution, the token is received in the gateway simply, while in the final solution the token is obtained from the Customer microservice, which presents a slightly greater complexity, which may explain this big difference.

The other highlight is the get customers query, which in this case, the first solution takes longer than the GraphQL federation solution. In this situation, there is no significant difference in implementation other than the use of federation. The time difference can be explained because this is the most complex query, as it brings much information associated with customers. This can show that the use of federation can improve the performance in highly complex operations. There is no certainty about this analysis because it would be necessary to carry out a more in-depth study using different complexities of queries and perhaps a more complex solution.

7.3.3 Maintainability - Qualitative Evaluation

To complete the solution's evaluation, it was decided to conduct a qualitative analysis of the maintainability. Then it was realized that assessing the maintainability using only quantitative metrics such as the one made with SonarQube is not enough to evaluate a solution's maintainability. As it's possible to see in subsection 7.3.1, a quantitative assessment depends on how the developer decides to implement the solution, and a less careful implementation can negatively influence the results.

Next, for each solution, the necessary changes for each defined question are presented, and an analysis of the results obtained will be presented at the end of the section.

Initial Solution

In the initial solution, to add a new microservice, it is needed to make the following changes:

Microservice:

- Create a new application with the new microservice.
- Create controller, repository, and model classes.
- Create an API Response class, which has the logic for how requests and error messages are constructed.

Gateway:

- Create a folder for the new microservice.
- Create a file with the microservice schema.
- Create another file with the resolvers implementation (mutations and queries).
- In the config.ts file, it is necessary to define the URL of the new microservice.
- In the main.ts file, define the new component created.

Depending on the complexity of the microservice or the logic needed to implement it, it could be necessary to add more classes, but it was decided to simulate adding a microservice with more or less the same complexity and, of course, using the same logic as those that already exist.

Moving on to the situation where a new operation is added, in this case, a mutation, the steps for this to happen would be very similar to adding a new microservice, but instead of having a new application and schema, it would only be necessary to change the ones that exist. So the steps would be as follows:

Gateway:

- Add a new mutation to the schema file (e.g., edit customer).
- Add a resolver for the new mutation in the operations file and the necessary logic to call the microservice.

Microservice:

- Add in the controller class a new function that will receive the new request and process the necessary logic.
- Add the function to update the customer data in the repository class.

GraphQL Federation Solution

Moving on now to the evaluation of the final solution, referring to adding a new microservice, for the solution with federation, when it is necessary to add a microservice, it is needed to follow these steps:

Microservice:

- Create a new application with the microservice.
- Create the schema in the resources/schema folder.
- Create the data fetcher class with the implementation of the GraphQL operations defined in the schema (queries and mutations).
- Create the service class that will have the domain logic.
- Create the repository class to perform the accesses to the database.
- If necessary create the data loader class if this new entity is referred to in an external type (in another entity), as in the microservice accounts.
- Then, it just needs to run the command that puts the new schema in Apollo Studio.

As mentioned for the initial solution, it may be necessary to create more classes depending on the logic to implement.

As was done for the initial solution, it now follows for the case where a microservice is changed to add a new operation. The steps are as follows:

Microservice:

- Add the new mutation to the file that has the schema.
- Create a function for the mutation ("@DGSMutation").

- Create the service that will contain the update logic and call the repository.
- Create the repository class function that will update the data in the database.
- Run the command that updates the schema in Apollo Studio.

Evaluation

Comparing the analyses of how a new microservice has to be implemented in each of the solutions, it is possible to see that it is always necessary to change at least two components in the initial solution. It is needed to change the gateway where it is essential to add the new schema, the resolvers, and the URL of the new microservice. And also, it is needed to add the new application with the microservice logic.

For the final solution (with federation), it is only necessary to change a single component, which is the microservice that is created, and there is no need to change components that were already made.

These analyses show that there is less dependency between application components when using federation which means that the elements of the system are practically independent. Therefore, it's possible to say that the solution with federation has great modifiability because it is possible to change the system without negatively impacting its quality. This is because if there was a problem with the new code, the functions developed previously would continue to work, as there were no changes to what was already implemented. On the other hand, in the initial solution, new code is added in a component that is used by all microservices, which, if for some reason an error occurs in this code, it can make the system completely inoperable.

Now analyzing the situation of adding a new operation in the two solutions, it was possible to perceive, as in the previous analysis, that for the initial solution, it was necessary to change the gateway, and the customer microservice.

In the gateway, it was necessary to change two files already created and already used for other operations. In the microservice, it was only required to add the functions in the respective classes that help to support the update of the customer. In the final solution, it was necessary to make changes to the microservice, creating the functions to support the new operation, adding the mutation to the schema, and then running the command to update the schema in Apollo Studio. However, this step can be included in the deployment of the solution and is no longer done manually.

This second analysis shows that the system becomes more modular by using federation because the components have become so independent that a change in one feature has almost no impact on the other components.

Another point that must be considered in the analysis is the difficulty of developing the code in both solutions. From a more general point of view, it is possible to say that it may be more challenging to make changes to the initial solution because it is necessary to have a more comprehensive knowledge of the solution, and it is also required to know more than one technology since it is needed to change at least two components, the gateway (developed in javascript) and also the microservice (developed in Java), taking into account that the same technologies were maintained.

In terms of code implementation, looking only at the microservice because it is changed in both solutions, it is possible to see that the difficulty is practically the same because the use of federation in the analyzed cases does not increase the implementation logic in the microservices at all, it is only

done a little differently. While in the initial solution, the controller class defines the REST requests that will be received and also the business logic, directly calling the repository to make requests to the database and then returning the necessary information, in the solution with federation, the logic that the controller of the initial solution had was divided into two classes to have more separation of responsibilities. There is now the data fetcher class that will receive GraphQL requests, basically using the same logic that the controller uses to receive requests in REST, then there is the service class that will perform the business logic that was also in the controller in the initial solution and finally the service class calls the repository (same in both solutions).

So with federation, it becomes simpler to develop code because it is only necessary to change the microservice, but in terms of the difficulty of implementing this code in the microservice or terms of lines, the two solutions are practically the same. The only situation in which it can become more challenging to change or create a microservice when using federation is when it's necessary to have an entity to be referred to in another entity because it is necessary to implement a class called data loader that will have the required logic to map these fields in the external entity.

From the analyzes carried out, it was possible to conclude that the solution that uses GraphQL federation presents an improvement in maintainability when compared to the initial solution because, as explained in the previous analysis, the final solution is more modular and simple to make changes and this happens because with federation each subgraph (microservice) is designed to be completely independent of the other components. From the previous analyses, there is only a particular situation that may be a little more difficult to make a change in the solution with federation, which is when it is necessary to add logic referring to mapping fields of external entities, but after realizing how this logic is done it becomes something easy to repeat becoming simpler as you get more experience with federation, so despite this problem, with the other advantages presented above it can conclude that the solution with GraphQL federation can have better maintainability.

An important point to mention about the analysis is that when checking the necessary changes in the final solution for the microservice, it becomes clear that more classes need to be created or changed than in the initial solution, which may indicate that it is more difficult to maintain, but this happens because when developed it was decided that the federation should be implemented applying good practices. That's why more classes were created for separation of responsibilities, so each class has only a single responsibility, making the code more organized and easier to maintain and understand. But this has nothing to do with the use of federation but with the developer's concern about applying good practices.

Chapter 8

Conclusion

This section summarizes what was done, difficulties found, future work, and contributions.

8.1 Summary

This document presents all the steps taken to evaluate the impact of GraphQL federation in terms of performance and maintainability following the methodology defined in section 1.4. For this methodology, some key points were defined, and then for each one, the results were presented and how they were carried out:

- **Problem identification and motivation:** the problem of GraphQL with microservices was analyzed (section 1.2), and it was noticed that recently a new approach (GraphQL federation) appeared and started to be applied by some companies, but currently, there are still few studies on this approach, and those that exist focus on more in how to use, than in realizing what benefits it can bring in the development of solutions. Therefore, initially, there was a focus on understanding how to apply this approach and then verifying the best way to evaluate the impact in terms of quality attributes.

  However, all this knowledge helped build the background and state of the art (sections 2 and 3), which give a great context to the technologies used. The initial description of microservices and GraphQL helped to understand the main principles of each. Subsequently, state of the art was built where some approaches to the use of GraphQL with microservices are explained, where it was possible to observe the problem of old approaches and how GraphQL federation came to help some companies to overcome some issues, being also in this section explained how to implement GraphQL federation.

  One of the main difficulties encountered during the thesis was the lack of scientific papers on the implementation of GraphQL with microservices and its impact on the quality attributes, so it was necessary to explore some gray literature.

- **Define the objectives for a solution:** After an exhaustive study of what currently exists regarding the defined problem, a specific definition of the objective was made and the necessary steps to be taken to reach the main objective, which is **"What is the impact of GraphQL federation on maintainability and performance?"** (section 1.3).

- **Design and development:** Due to the time, it was decided not to develop a project from scratch but to use an application that already uses GraphQL without federation and later apply this approach. Time limitation was one of the most important factors to take into account in the part of the project selection process, as a project with a limited number of microservices had

to be chosen to be able to complete the project in the defined time. One of this dissertation's significant difficulties was finding a project that suited what was desired and fulfilled the defined characteristics (section 5.1) in a short time.

In the design section 5, the architecture of the two solutions (initial and with GraphQL federation) was explained, namely the components that constitute them and the design level changes necessary to apply federation. To document the information, diagrams were used, each followed by an explanation so that everything represented is understandable.

The main design changes were the gateway no longer implementing logic (passing this to the microservices resolvers), instead just forwarding requests. Microservices moved from REST to GraphQL and added schemas, resolvers, and data loaders to handle federation and all the logic associated with GraphQL. Finally, a new component was added, Apollo Studio, which is essential for managing all the subschemas and superschema that the gateway exposes to client applications.

In the implementation (section 6), an explanation of the code developed in the initial solution was carried out to get an idea of what was already implemented and the changes that would be necessary to start using GraphQL federation. For the solution with GraphQL federation, a more exhaustive explanation of the implementation was made, talking about the good practices implemented, showing code excerpts to make the main features necessary to implement federation more noticeable, and at the same time demonstrating that the previously defined design was respected.

A difficulty was that the author of this dissertation had never worked with GraphQL and also because an approach as recent as GraphQL federation is used, so it took some time to learn how GraphQL worked and later how to apply federation. However, due to the time limit, this knowledge has to be acquired as quickly as possible. Another problem was implementing federation in Java using Netflix's DGS because the existing documentation is not very detailed, and there are almost no examples of implementations.

- **Demonstration:** this point refers to the final solution with GraphQL federation and the final comparison made between the two solutions where it is possible to perceive the impact of GraphQL federation on each quality attribute. Tests were also carried out on the final solution to demonstrate that the newly implemented code did not influence the results obtained.
- **Evaluation:** to carry out the evaluation of the solutions (section 7), the GQM method was used, which helped to quantitatively measure the metrics defined in sections 7.1.1 and 7.1.2 for each of the defined quality attributes. These metrics were measured using the PRTG Network and SonarQube tools that, after some research, were realized that they could assess performance and maintainability, respectively. To measure maintainability, it was necessary to import each of the projects into the SonarQube tool, which automatically measured maintainability, both of which showed good maintainability. As for the performance in the PRTG Network tool, it was necessary to define sensors for each existing GraphQL request and then analyze the average response time presented in the tool.

  For maintainability, it was also necessary to conduct a qualitative analysis of the effort needed to change each of the solutions (adding a microservice or adding a new operation), and the dependencies that each solution has between the components were analyzed. This analysis showed which of the solutions was more modular or more modifiable, thus making it possible to verify maintainability.

Due to time constraints, it was impossible to carry out a more intensive analysis, for example, using a more complex solution (with more microservices), which could influence the results obtained.

- **Communication:** so that all the knowledge acquired can be consulted, this dissertation was developed, and it is also possible to consult the solution with GraphQL federation in a public repository on Github.

In conclusion, with the results obtained from the performance analysis, it was impossible to conclude that GraphQL federation impacts this quality attribute. The only situation where there was a significant variation in the response time was when a more complex request was made (more data returned, and the information came from the three microservices) to the solution with GraphQL federation, which presented better performance. But despite this, it is not possible to reach any conclusion because it only happened in that request, requiring a more complex analysis to verify this situation. Regarding the study of the impact of maintainability in quantitative terms, it was not possible to reach any conclusion, both solutions presented good results, in qualitative terms, it was possible to conclude that the GraphQL federation has an impact on the maintainability of the solution, showing better maintainability as is proven in section 7.3.

8.2 Future Work

The use of GraphQL federation is very recent, so there are still some points that can be explored. Regarding this dissertation, it would be essential to analyze the performance and maintainability of a more complex project (more microservices) and a project with different types of query complexity. That's why the author suggested adding new features (new microservices) in the future and also trying to increase the complexity of the queries to understand how this can mainly influence the solution's performance.

Another interesting point to develop in the future would be to migrate a corporate solution (a company that wanted to change its current solution) to GraphQL federation and evaluate the benefits, challenges, and disadvantages. With this experience, it could also be necessary to explore other attributes of qualities depending on how the migration experience went and what the company considers critical to analyze, possibly security.

The studies used specific technologies (Java, Spring boot, Javascript, DGS framework, Apollo-gateway, and Apollo-server), so to be more particular about the results, the influence on the quality attributes of the technologies that were used should be discarded. Therefore, the same analysis should be carried out using different technologies and verify that the results obtained were similar to those presented in this document.

Finally, this dissertation helped clarify some questions about federation, how to implement it and what impacts it can have in terms of performance and maintainability, but there are still many points to be explored because it is such a recent approach.

8.3 Contributions

This document was developed in the context of a master's thesis at ISEP to study the impact of using GraphQL with microservices on quality attributes. But as explained earlier, when starting to explore the topic, it was realized that a new approach, still little explored, has recently appeared, GraphQL federation, so it was decided to focus this dissertation on the impact that this approach has on quality attributes.

As GraphQL federation is a recent approach, this document describes how to implement it using Java and Javascript languages and how to use the Apollo Studio tool to manage schemas, no longer have to implement this logic in projects. Also, in a very concise way, it is explained how it can perform tests in java to test a GraphQL implementation.

The focus of the document is on evaluating the impact of GraphQL federation on the attributes of quality, maintainability, and performance, and it was possible to conclude that for this complexity, there was no impact on performance, but there was an improvement in the maintainability of the solution with federation.

The source code is available on GitHub, fully open source, for testing, queries, improving what has been implemented, or even to continue the work developed.

Bibliography

[1] *What is an API? - API Beginner's Guide - AWS*. url: https://aws.amazon.com/pt/what-is/api/.

[2] M. I. Landeiro and I. Azevedo. "Analysis of GraphQL performance: a case study". In: *Software Engineering for Agile Application Development*. Vol. IGI Global. 2020, pp. 109–140.

[3] John Anderson and Nate Ross. *GraphQL: Making Sense of Enterprise Microservices for the UI*. url: https://medium.com/adobetech/graphql-making-sense-of-enterprise-microservices-for-the-ui-46fc8f5a5301.

[4] Antonio Quiña-Mera, José María García, Antonio Ruiz-Cortés, et al. "Quality in use evaluation of a GraphQL implementation". In: 2021. url: https://www.researchgate.net/publication/352800033.

[5] *Introduction to GraphQL | GraphQL*. url: https://graphql.org/learn/.

[6] Dane Avilla. *Beyond REST: Rapid Development With GraphQL Microservices | Netflix TechBlog*. url: https://netflixtechblog.com/beyond-rest-1b76f7c20ef6.

[7] Shanshan Li, He Zhang, Zijia Jia, et al. *Understanding and addressing quality attributes of microservices architecture: A Systematic literature review*. Mar. 2021. doi: 10.1016/j.infsof.2020.106449.

[8] Netflix. *How Netflix Scales its API with GraphQL Federation*. 2020. url: https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2.

[9] Netflix. *Scaling Netflix's API via GraphQL Federation (#2)*. url: https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-2-bbe71aaec44a.

[10] Alec Aivazis. *A Guide to GraphQL Schema Federation, Part 1*. url: https://alec.aivazis.com/blog/schema-federation/getting-started.

[11] Ville Touronen Helsinki, Jussi Kangasharju, Matti Luukkainen, et al. *Microservice architecture patterns with GraphQL*. Tech. rep. 2019.

[12] João Fontão. *JoaoFontao/GRAPHQL_FEDERATION*. 2022. url: https://github.com/JoaoFontao/GRAPHQL_FEDERATION.

[13] Jan vom Brocke, Alan Hevner, and Alexander Maedche. *Introduction to Design Science Research*. Sept. 2020. doi: 10.1007/978-3-030-46781-4{_}1.

[14] Nabor C. Mendonca, Craig Box, Costin Manolache, et al. "The Monolith Strikes Back: Why Istio Migrated from Microservices to a Monolithic Architecture". In: *IEEE Software* 38.5 (Sept. 2021), pp. 17–22. doi: 10.1109/MS.2021.3080335.

[15] Ossi Puustinen. "GRAPHQL FOR BUILDING MICRO-SERVICES". PhD thesis. Tampere University, Apr. 2020.

[16] *GraphQL Best Practices | GraphQL*. url: https://graphql.org/learn/best-practices/.

[17] Matthias Biehl. *GraphQL API Design*. Second. Vol. 5. Apr. 2018. isbn: 978-1979717526. url: https://play.google.com/books/reader?id=7j64DwAAQBAJ&pg=GBS.PA76&hl=pt_PT.

[18] *Schemas and Types | GraphQL*. url: https://graphql.org/learn/schema/.

[19] Muhammad Haris, Kheir Eddine Farfar, Markus Stocker, et al. "Federating Scholarly Infrastructures with GraphQL". In: (Sept. 2021). url: http://arxiv.org/abs/2109.05857.

[20] Microsoft Docs. *Microservice architecture style - Azure Application Architecture Guide*. url: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[21] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Architectural patterns for microservices: A systematic mapping study". In: *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*. Vol. 2018-January. SciTePress, 2018, pp. 221–232. isbn: 9789897582950. doi: 10.5220/0006798302210232.

[22] *Istio ingress controller as an API gateway · Banzai Cloud*. url: https://banzaicloud.com/blog/backyards-api-gateway/.

[23] Sam Newman. *Building Microservices*. Ed. by Mike Loukides and Brian MacDonald. First. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., Feb. 2015. url: http://safaribooksonline.com.

[24] *Backend for Frontend (BFF) Pattern: The Dos and Don'ts of the BFF Pattern | AKF Partners*. url: https://akfpartners.com/growth-blog/backend-for-frontend.

[25] AWS. *O que são microsserviços?* url: https://aws.amazon.com/pt/microservices/.

[26] Microsoft Docs. *Microservice architecture style - Azure Application Architecture Guide | Microsoft Docs*. url: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[27] Vibhu Singhal. *GraphQL in a Micro Services Architecture · WaveMaker Docs*. June 2020. url: https://docs.wavemaker.com/learn/blog/2020/06/11/graphql-microservice-architecture.

[28] Sri Lakshmi Vadlamani, Benjamin Emdon, Joshua Arts, et al. "Can GraphQL Replace REST? A Study of Their Efficiency and Viability". In: *Proceedings - 2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice, SER and IP 2021*. Institute of Electrical and Electronics Engineers Inc., June 2021, pp. 10–17. isbn: 9781665444767. doi: 10.1109/SER-IP52554.2021.00009.

[29] *Moving toward GraphQL consolidation - Federation - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/enterprise-guide/graphql-consolidation/.

[30] *Introduction to Apollo Federation - Federation - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/.

[31] *Netflix/dgs-framework: GraphQL for Java with Spring Boot made easy.* url: https://github.com/netflix/dgs-framework/.

[32] *Home - DGS Framework*. url: https://netflix.github.io/dgs/.

[33] *AWS AppSync | Managed GraphQL APIs | Amazon Web Services*. url: https://aws.amazon.com/appsync/?nc1=h_ls.

[34] *What is AWS AppSync*. url: https://serverless-stack.com/chapters/what-is-aws-appsync.html#what-is-aws-appsync.

[35] *Introduction to Apollo Server - Apollo Server - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/apollo-server/.

[36] *Fastify, Fast and low overhead web framework, for Node.js*. url: https://www.fastify.io/.

[37] *mercurius-js/mercurius: Implement GraphQL servers and gateways with Fastify*. url: https://github.com/mercurius-js/mercurius#quick-start.

[38] *RetailMeNot GraphQL Federation | by Kartik Kumar Gujarati | Ziff Media Engineering*. url: https://engineering.ziffmedia.com/retailmenot-graphql-federation-3bb36dac5aaa.

[39] Naga Malepati. *Federated GraphQL @ Walmart. This post talks about our motivation... | by Nidhi Sadanand | Walmart Global Tech Blog | Medium*. url: https://medium.com/walmartglobaltech/federated-graphql-walmart-bfc85c2553de.

[40] Andy Roberts. *Schema Services: Transitioning Towards a Federated Architecture - Apollo GraphQL Blog*. Nov. 2020. url: https://www.apollographql.com/blog/backend/architecture/schema-services-transitioning-towards-a-federated-graphql-architecture/.

[41] *ISO 25010*. url: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[42] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. "Promises and challenges of microservices: an exploratory study". In: *Empirical Software Engineering* 26.4 (July 2021). issn: 15737616. doi: 10.1007/s10664-020-09910-y.

[43] Kenneth Lysons and Brian Farrington. *Procurement and Supply Chain Management*. 9th ed. Pearson, 2016. isbn: 9781292086149.

[44] P. Belliveau, A. Griffin, and S. Somermeyer. *The PDMA toolbook for new product development.* Vol. 1. New York: N.Y.: John wiley & Sons, 2002, pp. 2–21.

[45] Peter A Koen, Greg M Ajamian, Scott Boyce, et al. *FuzzyFrontEnd: Effective Methods, Tools, and Techniques*. Tech. rep.

[46] Peter Koen, Greg Ajamian, Robert Burkart, et al. *PROVIDING CLARITY AND A COMMON LANGUAGE TO THE "FUZZY FRONT END"*. Tech. rep. 2001.

[47] Peter A Koen. *Understanding the Front End: A Common Language and Structured Picture*. Tech. rep. Stevens Institute of Technology, May 2004. url: www.frontendinnovation.com.

[48] IBM Market Development & Insight. "Microservices in the enterprise, 2021: Real benefits, worth the challenges". In: ().

[49] Matt Asay. *How GraphQL turned web development on its head | InfoWorld*. June 2020. url: https://www.infoworld.com/article/3545951/how-graphql-turned-web-development-on-its-head.html.

[50] *Stack Overflow Trends*. url: https://insights.stackoverflow.com/trends?tags=graphql.

[51] Geoff Lancaster. *Value-based marketing and its usefulness to customers*. Nov. 1999.

[52] R. Woodruff. "Customer value: The next source for competitive advantage." In: *Journal of the Academy of Marketing Science* (1997), pp. 139–153.

[53] V. Zeithaml. "Consumer Perceptions of Price, Quality, and Value: A Means-End Model and Synthesis of Evidence." In: *Journal of Marketing* (1988), pp. 2–22.

[54] John Borza. *FAST Diagrams: The Foundation for Creating Effective Function Models General Dynamics Land Systems*. Tech. rep. 2011.

[55] São Paulo. *Aplicação das técnicas de Metodologia do Valor no processo administrativo de montagem de equipamentos*. Tech. rep. 2008.

[56] *malik-aliyev-94/payday: Kubernetes Microservices with spring boot, flask, NodeJS GraphQL, MySQL and MongoDB*. url: https://github.com/malik-aliyev-94/payday.

[57] *Design Principles | Object Oriented Design*. url: https://www.oodesign.com/design-principles.

[58] Craig Larman. *Applying UML and Patterns*. Second Edition.

[59] Julieta Calabrese, Rocío Muñoz, Ariel Pasini, et al. *Assistant for the Evaluation of Software Product Quality Characteristics Proposed by ISO/IEC 25010 Based on GQM-Defined Metrics*. Tech. rep.

[60] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. *THE GOAL QUESTION METRIC APPROACH*. Tech. rep. Maryland: Institute for Advanced Computer Studies.

Appendix A

Maintainability Results

From figure A.1 to figure A.7, an overview of the results obtained for each component using the SonarQube tool is presented.

As previously mentioned (subsection 7.1.1), SonarQube has a scale from A to E to evaluate the maintainability of a project. The best value of the maintainability scale provided by SonarQube is A and the worst is E.

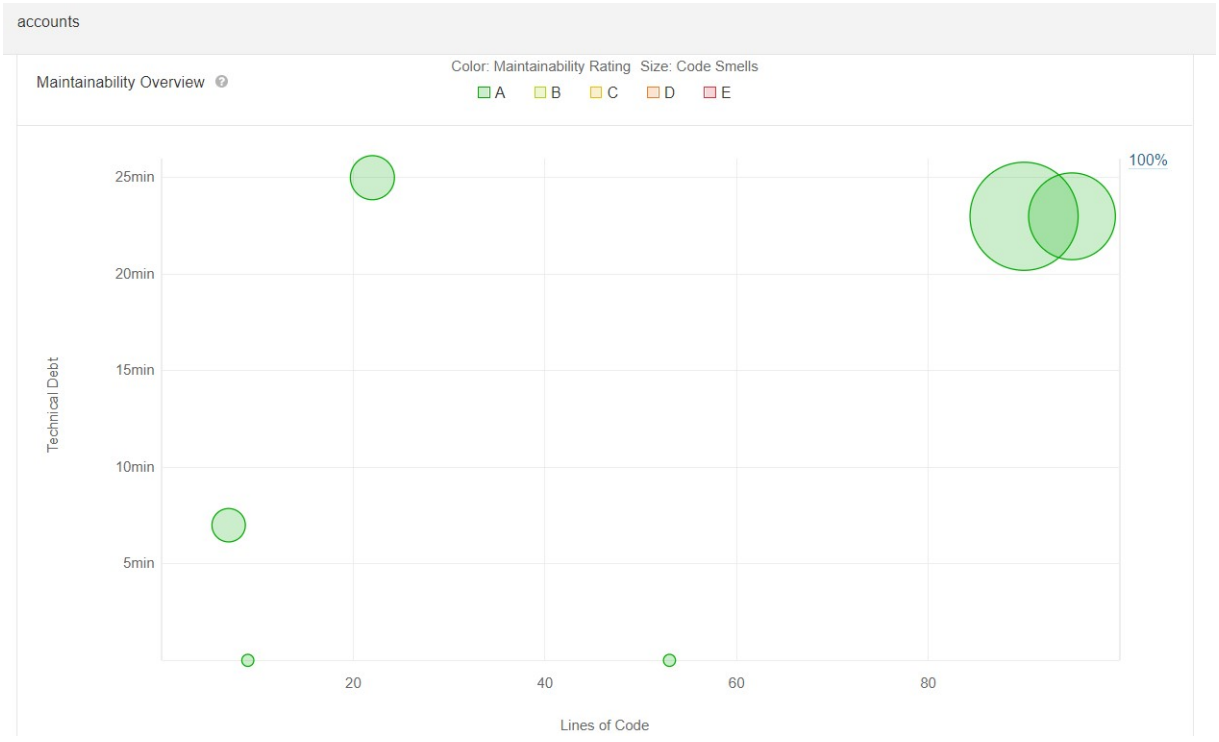

Figure A.1: SonarQube Overview Accounts Initial Solution

Figure A.2: SonarQube Overview Customer Initial Solution

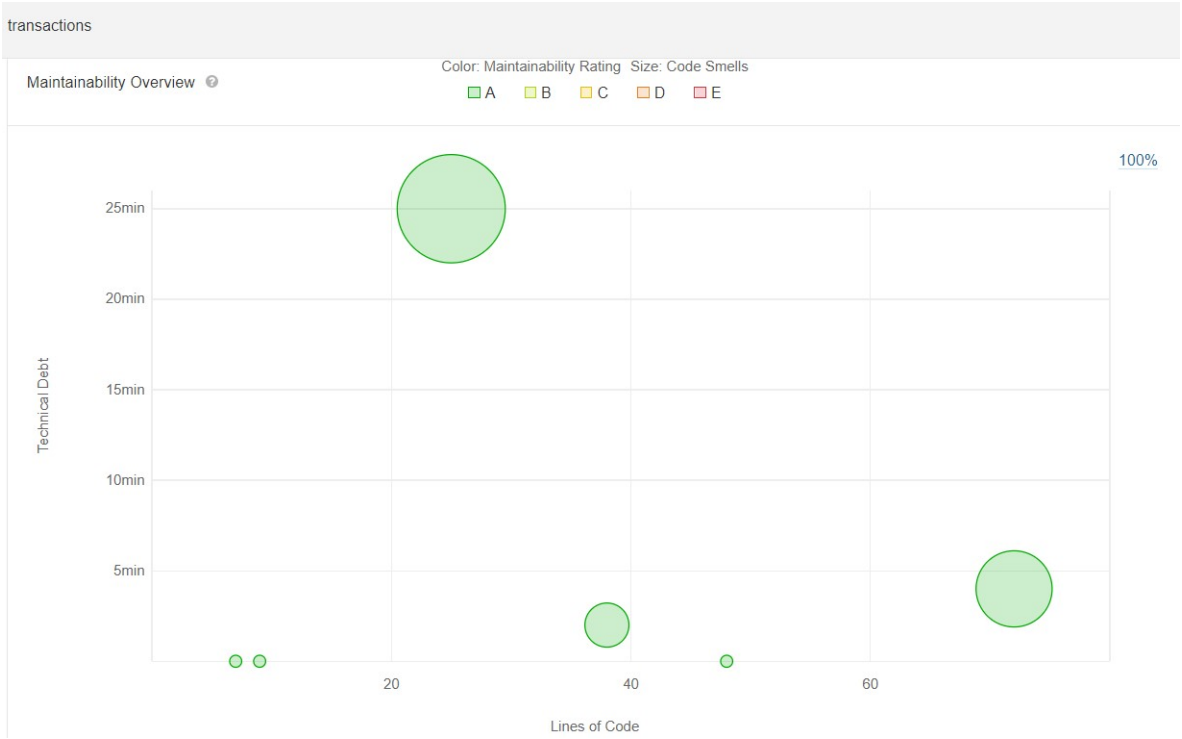

Figure A.3: SonarQube Overview Transaction Initial Solution

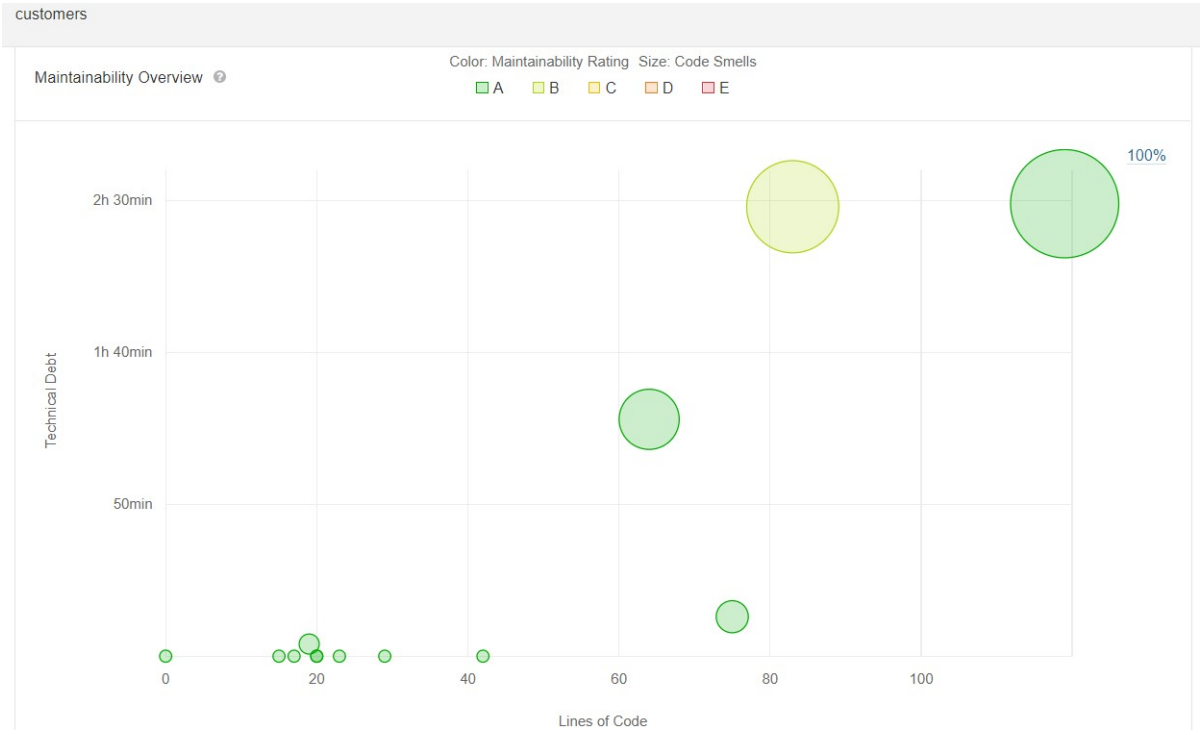

Figure A.4: SonarQube Overview Gateway Initial Solution

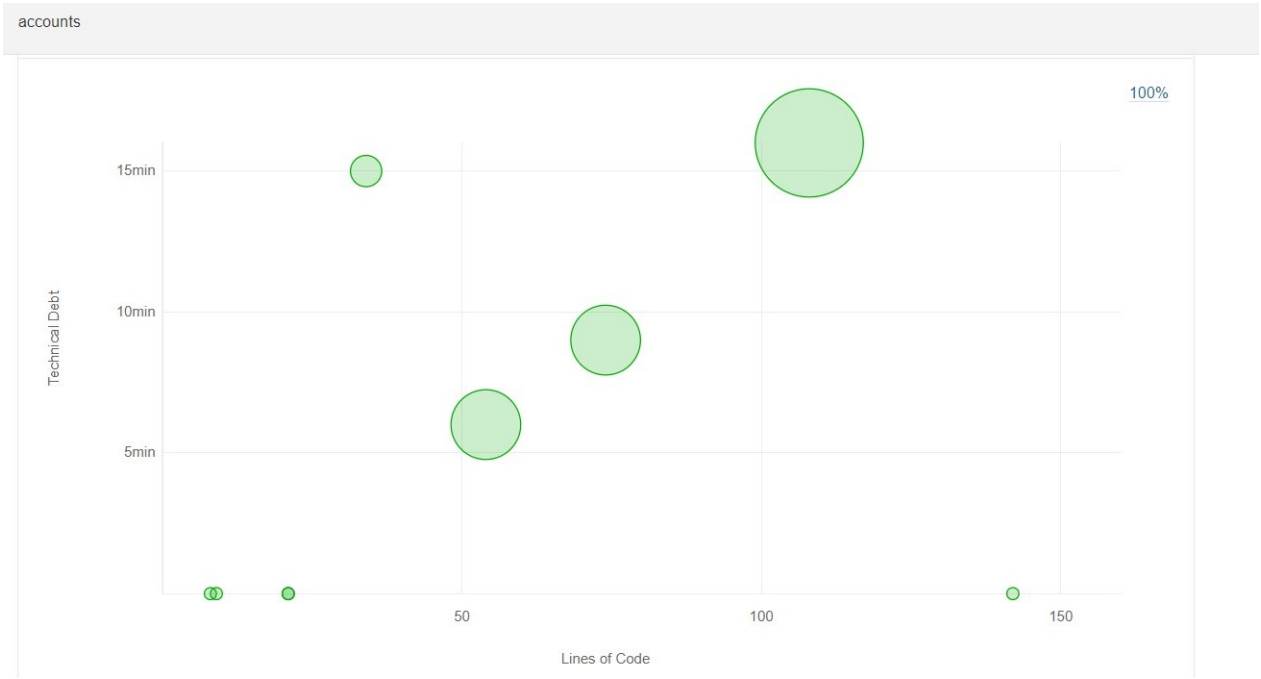

Figure A.5: SonarQube Overview Accounts Final Solution

Figure A.6: SonarQube Overview Transaction Final Solution

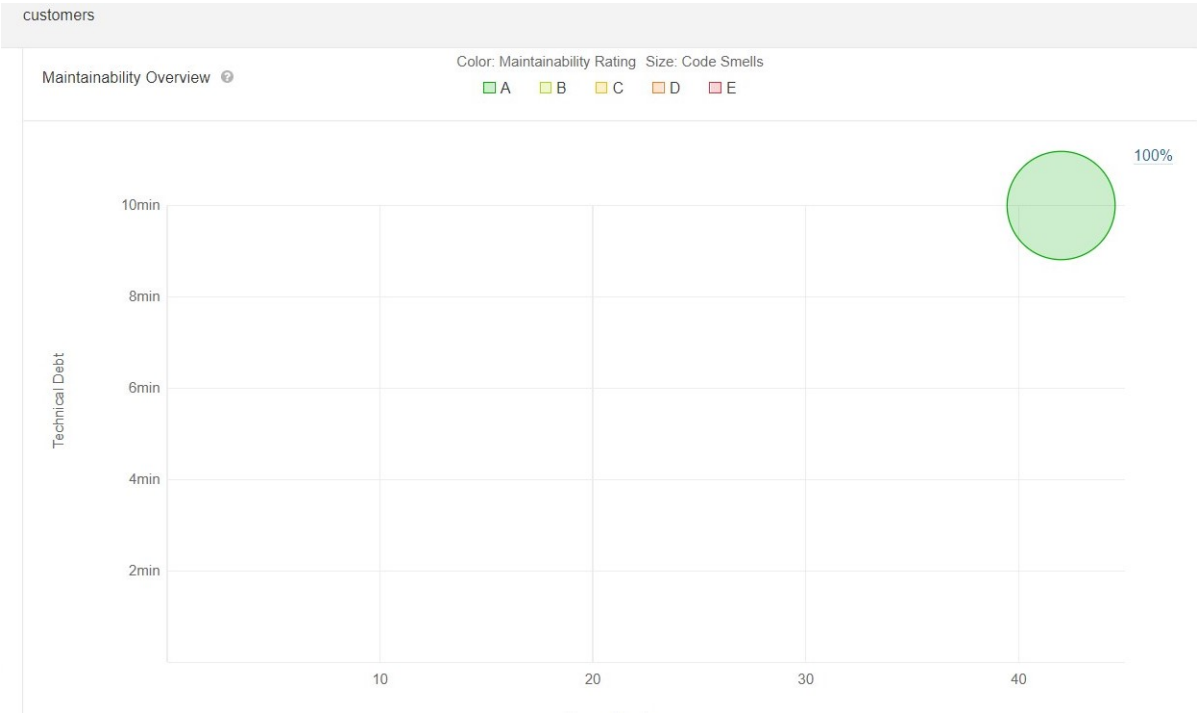

Figure A.7: SonarQube Overview Gateway Final Solution

From figure A.8 to figure A.15, the results obtained for each component using the SonarQube tool are presented in more detail. In the details, the debt ratio, rating, debt, and finally, the code smells found are presented

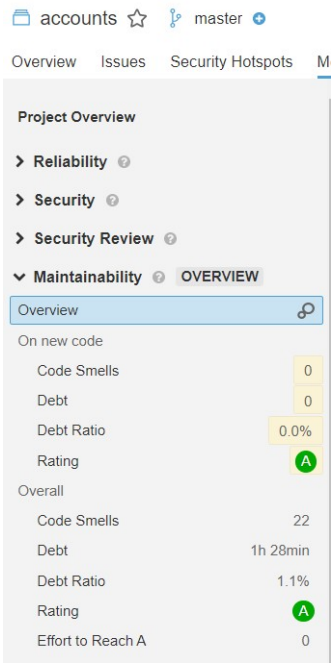

Figure A.8: Results Account Maintainability Initial Solution

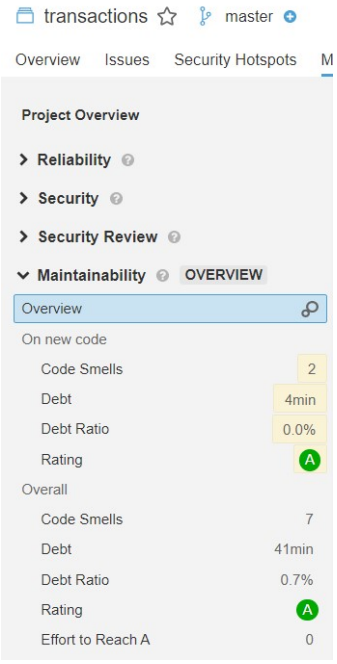

Figure A.9: Results Transaction Maintainability Initial Solution

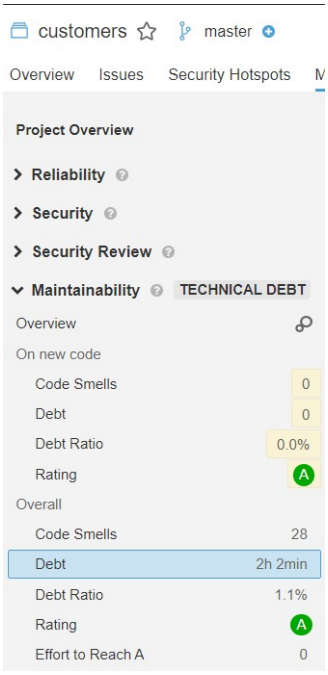

Figure A.10: Results Customer Maintainability Initial Solution

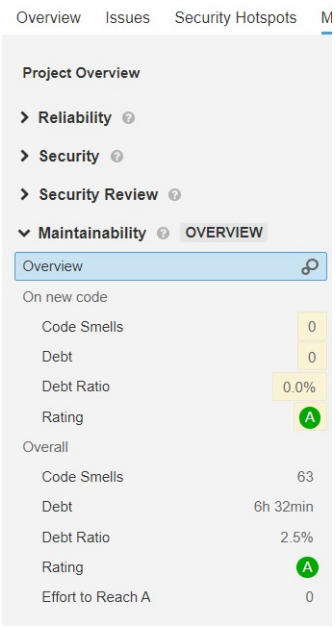

Figure A.11: Results Gateway Maintainability Initial Solution

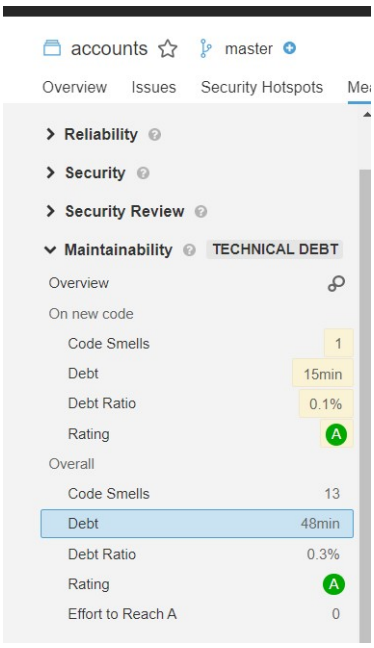

Figure A.12: Results Account Maintainability Final Solution

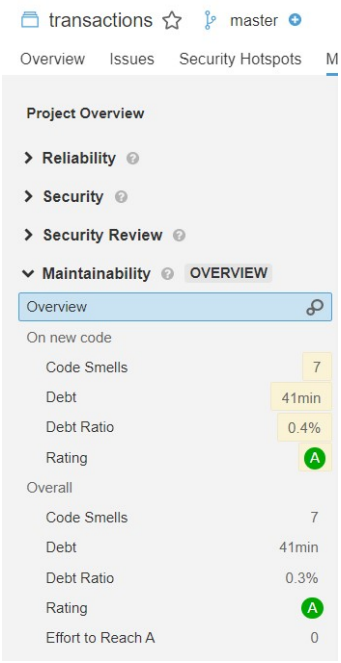

Figure A.13: Results Transaction Maintainability Final Solution

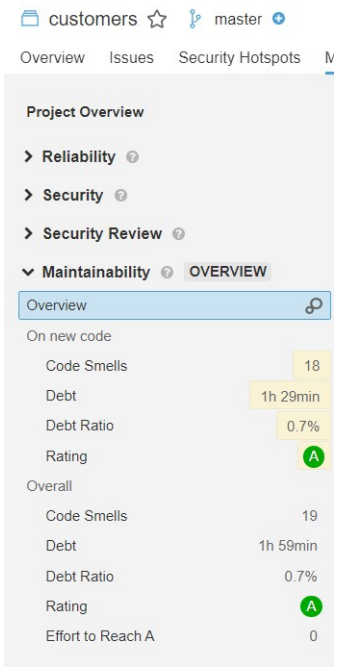

Figure A.14: Results Customer Maintainability Final Solution

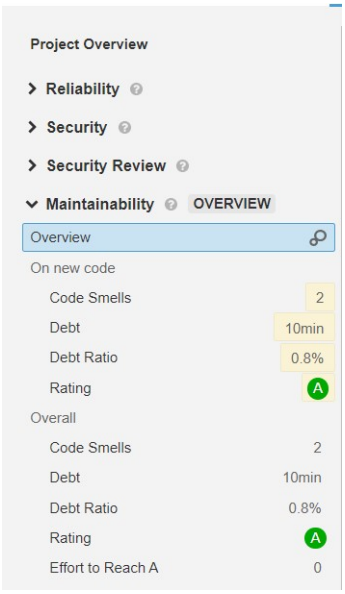

Figure A.15: Results Gateway Maintainability Final Solution

Appendix B

Performance Results

This annex presents the sensors created in the PRTG Network tool to evaluate the performance of the solutions.

Sensor	Probe Group Device	Status	Last Value	Message	Graph	Priority	Fav.
Initial Solution - create acco...	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	41 msec	OK	Loading time 41 msec	★★★☆☆	
Initial Solution - create trans...	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	34 msec	OK	Loading time 34 msec	★★★☆☆	
Initial Solution - disable acc...	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	28 msec	OK	Loading time 28 msec	★★★☆☆	
Initial Solution - get account...	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	53 msec	OK	Loading time 53 msec	★★★☆☆	
Initial Solution - get customer	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	37 msec	OK	Loading time 37 msec	★★★☆☆	
Initial Solution - get custom...	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	156 msec	OK	Loading time 156 msec	★★★☆☆	
Initial Solution - get transact...	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	44 msec	OK	Loading time 44 msec	★★★☆☆	
Initial Solution - login	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	106 msec	OK	Loading time 106 msec	★★★☆☆	
Initial Solution - Signup	Sonda local (Local Probe) » Servidores » DESKTOP-EU2E4F7	Up	37 msec	OK	Loading time 37 msec	★★★☆☆	

Figure B.1: Sensors Initial Solution

Figure B.2: Sensors Final Solution