INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Explorar performance com Apollo Federation

LEANDRO DANIEL OLIVEIRA QUEIRÓS
Junho de 2023

POLITÉCNICO
DO PORTO

Exploring performance with Apollo Federation

Leandro Queirós

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, June 30, 2023

Statement of Integrity

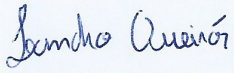
I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, June 2023



Leandro Queirós

Abstract

The growing tendency in cloud-hosted computing and availability supported a shift in software architecture to better take advantage of such technological advancements. As Monolithic Architecture started evolving and maturing, businesses grew their dependency on software solutions which motivated the shift into Microservice Architecture.
The same shift is comparable with the evolution of Monolithic GraphQL solutions which, through its growth and evolution, also required a way forward in solving some of its bottleneck issues. One of the alternatives, already chosen and proven by some enterprises, is GraphQL Federation. Due to its nobility, there is still a lack of knowledge and testing on the performance of GraphQL Federation architecture and what techniques such as caching strategies, batching and execution strategies impact it.
This thesis aims to answer this lack of knowledge by first contextualizing the different aspects of GraphQL and GraphQL Federation and investigating the available and documented enterprise scenarios to extract best practices and to better understand how to prepare such performance evaluation.
Next, multiple alternatives underwent the Analytic Hierarchy Process to choose the best way to develop a scenario to enable the performance analysis in a standard and structured way. Following this, the alternative base solutions were analysed and compared to determine the best fit for the current thesis. Functional and non-functional requirements were collected along with the rest of the design exercise to enhance the solution to be tested for performance.
Finally, after the required development and implementation work was documented, the solution was tested following the Goal Question Metric methodology and utilizing tools such as JMeter, Prometheus and Grafana to collect and visualize the performance data. It was possible to conclude that indeed different caching, batching and execution strategies have an impact on the GraphQL Federation solution. These impacts do shift between positive (improvements in performance) and negative (performance hindered by strategy) for the different tested strategies.

Keywords: GraphQL,Apollo Federation,Performance,GraphQL Federation

Resumo

A tendência de crescimento da computação cloud-hosted apoiou uma mudança na arquitetura do software para tirar maior proveito desses avanços tecnológicos. Com a evolução e amadurecimento das arquiteturas monolíticas, as empresas aumentaram sua dependência nas soluções software que motivou a mudança e adoção de arquiteturas de micro serviços. O mesmo se verificou com a evolução das soluções monolíticas GraphQL que, com o seu crescimento e evolução, também requeriam soluções para resolver algumas das suas novas complexidades. Uma das alternativas de resolução, já aplicado e provado na indústria, é o GraphQL Federation. Devido ao seu recente lançamento, ainda não existe um conhecimento sólido na performance de uma arquitetura de GraphQL Federation e que técnicas como estratégias de caching, batching e execution tem impacto sobre a mesma.
Esta tese tem como intuito responder a esta falha de conhecimento através de, primeiramente, contextualizar os diferentes aspetos de GraphQL e GraphQL Federations com a investigação de casos de aplicação na indústria, para a extração de boas práticas e compreender o necessário ao desenvolvimento de uma avaliação de performance.
De seguida, múltiplas alternativas foram sujeitas ao Analytic Hierarchy Process para escolher a melhor forma de desenvolver um cenário/solução necessária a uma análise de performance normalizada e estruturada. Com isto em mente, as duas soluções base foram analisadas e comparadas para determinar a mais adequada a esta tese. Requisitos funcionais e não-funcionais foram recolhidos, assim como todo o restante exercício de design necessário ao desenvolvimento da solução para testes de performance.
Finalmente, após a fase de desenvolvimento ser concluída e devidamente documentada, a solução foi testada seguindo a metodologia Goal Question Metric, e aplicando ferramentas como JMeter, Prometheus e Grafana para recolher e visualizar os dados de performance. Foi possível concluir que, de facto, as diferentes estratégias de caching, batching e execution tem impacto numa solução GraphQL Federation. Tais impactos variam entre positivos (com melhorias em termos de performance) e negatives (performance afetada por estratégias) para as diferentes estratégias testadas.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

AHP	Analytic Hierarchy Process.
API	Application Programming Interface.
CD	Continuous Deployment.
CI	Continuous Integration.
DGS	Domain Graph Service.
GQM	Goal Question Metric.
JVM	Java Virtual Machine.
MSA	Microservice Architecture.
REST	Representational State Transfer.

Chapter 1

Introduction

This chapter comprehends a contextualization, the problem to be analysed and the main objectives of this document, as well as an overview of the document structure that will be followed throughout.

1.1 Context

Cloud-hosted software/computing has been a growing tendency in the software engineering community (Lauretis 2019). This increased tendency and availability of such services as Microsoft Azure (Azure 2022), AWS (Amazon 2022) and others brought several benefits such as rapid elasticity, the ability to scale freely and high reliability (Rashid and Chaturvedi 2019).
Due to the constant and growing need for effective solutions by organizations and their users/clients, the use of the correct architecture and technology stack to take advantage of the cloud computing benefits become increasingly critical. Although there are multiple alternative architectures, two of the most widespread are the Monolithic and Microservice architecture (Gos and Zabierowski 2020).
Unlike the traditional Monolithic Architecture approach where all business functions and services are centralized in a single responsibility solution, the Microservice Architecture (MSA) aims to decompose the business domain into small, consistent bounded contexts and then represent them as a set of loosely coupled, deployment independent and autonomous services (Blinowski, Ojdowska, and Przybylek 2022).
Despite the benefits of the Microservice Architecture, additional complexities that previously were not existent in a monolithic approach are now extremely relevant (Blinowski, Ojdowska, and Przybylek 2022; Gos and Zabierowski 2020; Rashid and Chaturvedi 2019).

1.2 Problem

One of these complexities is how to effectively provide responses to API requests when these responses require data from multiple microservices. Using the database-per-service pattern will mean that each microservice will have a different database which queries and transacts into (Naman and Kerthyayana Manuaba 2022).
As an alternative to traditional REST APIs, GraphQL uses high-level abstractions such as schemas, types, queries and mutations to effectively and dynamically respond to API requests (Brito and Valente 2020). Although benefits in development times and debatable performance improvements (Brito and Valente 2020; Naman and Kerthyayana Manuaba 2022), GraphQL APIs suffer from similar issues as REST APIs in a microservice architecture when information needs to be joined from multiple microservices APIs.

In the State of GraphQL, 2022 survey report it's possible to observe that performance is the second most popular pain point, only losing its first place to error handling. Despite this, error handling has been extensively documented by the Apollo Federation spec (Apollo GraphQL 2023a). Furthermore, GraphQL Federation is the second most known of the "Other Features" section of the survey, only being less known than GraphQL Subscriptions State of GraphQL 2022.
In order to provide a way forward in solving the cross-microservice API queries, GraphQL Federation offers an open architecture which allows the combination of GraphQL API (subgraphs) into a supergraph (Apollo GraphQL 2022d). Consequently, when querying the supergraph router/gateway it's possible to fetch data from all the underlying subgraphs (GraphQL APIs) (Apollo GraphQL 2022d).
Due to its nobility, there is still a lack of knowledge and testing on the performance of GraphQL Federation architecture. Moreover, performance issues using federation and some common packages have been reported (Apollo GraphQL 2022b).
Despite the fact that there are previous works done which strive to determine the impact of a solution migration to GraphQL Federation, these are focused on comparing an architecture without GraphQL Federation and one with GraphQL Federation implementation (Fontão 2022). Relevant concepts like the different caching strategies (Schrade 2021) and custom execution strategies (Helfer 2021) are still unexplored, especially with Federation 2.1, released in August 2022 which may make a difference between a performant architecture or a slow response time solution.

1.3 Research Question

The aim of this work is to assess what impacts GraphQL Federation performance by exploring caching and execution strategies, as well as other peculiarities if any, that can affect it. To achieve this, a set of sub-objectives needs to be achieved.
Firstly, there is the need to explore which are the different caching, batching and execution strategies available in GraphQL and GraphQL Federation, and if there is already some evident impact on software performance. Some enterprise approaches GraphQL in federated architectures or GraphQL Federation can be insightful in unveiling any performance flaws detected in production environments. Furthermore, they can provide some best practices or workarounds to overcome some of the faced challenges when adopting GraphQL Federation.
Finally, there is a need to design an approach to consider performance using GraphQL Federation while determining ways to collect data to measure performance and determine its effects on other quality attributes.
During the extent of this document, there is a specific research question that needs to be answered, in the most detailed and well-founded way possible, as a primary output of all the work developed and documented in this document/thesis.
Research Question: What is the impact on the performance of GraphQL Federation in a microservice architecture application when utilizing custom caching, batching and execution strategies?

1.4 Methodology

Besides the GraphQL Federation, there are more alternatives to solve the problem of managing multiple GraphQL services such as Schema Stitching (Stubailo 2017) or pre-federation

patterns (Apollo GraphQL 2022f), due to the high number of enterprise use cases, the Federation specification displayed as relevant to explore further (Hampton, Watson, and Wise 2020).
While the premise of stitching multiple GraphQL graphs to provide a unified graph for a client or external applications to use proposed by the Schema Stitching alternative can solve the issue of unifying the underlying sub-graphs, multiple new challenges can arise from such endeavour. These include the logic required to operate such a unified graph and allow communication between the multiple graphs and changes to sub-graphs implying changes in the stitching service which may cause issues on the integration and deployment of such services (Isquick 2021).
Thus the methodology that will be followed through this document to achieve the objectives documented in Section 1.3 will consist of an analysis of the GraphQL Federation specification, with a focus on performance aspects.
While GraphQL is a query language, GraphQL Federation or the Apollo Federation specification consists of an open architecture which aims to manage GraphQL services (sub-graphs) through the use of a supergraph which represents a combination of all its sub-graphs.

1.5 Code Repository

All the code developed in Chapter 6 and data collected for Chapter 7 is available on a public GitHub repository (Queirós 2023). This is an important step and choice for this thesis as it allows for the current process and methodology to be re-used and iterated upon by future readers without unnecessary redevelopment.

1.6 Document structure

The current document/report will be split into x chapters. Each chapter will have a set of individual sections which aim for logical organization and sequencing of the knowledge to allow a good level of readability and quick navigation to the desired information depending on the aim of the reader. The chapters and their contents will be the following:

- State of Art - Comprehends the description and documentation of the highest level of general development in the area aimed by the current document. This chapter can also include technological or approach alternatives. In this document's specific case, a contextualization from Monolithic to Microservices architecture will be included with each approach's benefits and drawbacks, leading to the specific technology of GraphQL and GraphQL Federation. Lastly, an overview of the quality attributes is presented to accurately define the performance quality attribute
- Value Analysis - Comprehends the theoretical contextualization and purpose of a value analysis as well as the identification of the opportunity which drove the inception of the current document. The Analytic Hierarchy Process (AHP) method is a structured and standardized methodology to determine the best alternative to carry out the necessary work to answer the research question in Section 1.3.
- Analysis and Design - This chapter focuses on the analysis of the possible base solution alternatives by investigating each solution's technologies, benefits and constraints. Next, all the functional and non-functional requirements were documented to attain a performance assessment-ready application. Finally, architectural views following the 4+1 Vew Model Kruchten 1995 are provided to ease the visualization of the final solution layout, components and their interactions.

- Implementation - Aims to document all the required implementation processes for this thesis. This includes all the changes from the development of the functional requirements to the multiple caching, batching and execution strategies.
- Evaluation and Experimentation - Focus on evaluating the Implementation output through the specification of assessment metrics, methodology and the results of such analysis. In this document-specific case, firstly the hypothesis is specified, which will drive the rest of the evaluation and validation process. Afterwards, the Goal Question Metric (GQM) method is utilized to support a structured and organized of measuring and evaluating the hypothesis through the definition of a goal, questions and metrics to assess such questions. Ultimately, all the test scenarios, methodology, tools and other processes used to effectively collect the data for the previously defined metrics are documented, leading up to the conclusions that were possible to extrapolate from such data and experiments.
- Conclusion - Provides closure to the current document. This includes an analysis of what was achieved, any challenges faced, future improvements and final considerations or reflections.

Chapter 2

Background

In the current chapter, the main focus will be a further contextualization required to effectively understand the problem at hand and the rest of the work developed throughout this document.
A brief introduction to Microservice Architecture (MSA) and Monolithic Architecture is recorded, consisting of a description of its peculiarities, benefits and challenges, as well as some of the patterns that are adapted to face these.

2.1 Monolithic Architecture

To introduce the Microservice Architecture there is a need to understand what the monolithic approach consists of. This architecture is described as monolithic as all pieces of functionality and business logic that make up the software solution will be centralized and encapsulated in a single component/application. Underlying modules cannot be executed independently, this will require a full application execution (Ponce, Marquez, and Astudillo 2019).
In a specific scenario of a web application solution, where usually there is three essential components: a front-end / client-side application, a back-end / server-side application and a relational or non-relational database to persist all information, the server-side application will be of monolithic nature, any changes to its processes, logic or communication parameters will require a build and deployment of a new version of the server-side application (Lewis and Fowler 2014). A representation of a monolithic architecture and its scaling feature can be observed in Figure 2.1.

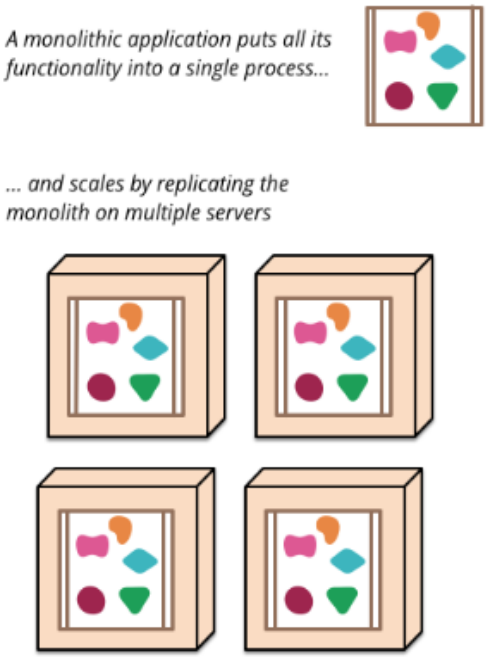


Figure 2.1: Monolithic Architecture (Lewis and Fowler 2014)

2.1.1 Advantages

The clearest of the advantages when opting for a Monolithic Architecture is its simplicity (MicroServiceIO 2018):

- Simple to develop: As all code and functionality sit under a single project/solution, it's quite a single task to retrieve/use/extend any current behaviour for a functionality update. IDE(s) are quite capable of making assertive suggestions when browsing the same project (JetBrains 2022).
- Simple to deploy: The deployment process involves the deployment of a single component, usually there is no need to handle deployed container communication as all logic and functionality are concentrated on a single one.
- Simple to scale: To scale a monolithic application means to deploy a new instance with the same code base or version. This simplicity must not be misunderstood as the best use of scalability resources.

2.1.2 Disadvantages

The simplicity provided by a Monolithic Architecture quickly fades away when the solution evolves and grows over the years implying a greater time to build and deploy, as well as increasingly powerful machines to run such software pieces. According to (Kalske, Mäkitalo, and Mikkonen 2018; MicroServiceIO 2018; Ponce, Marquez, and Astudillo 2019) other drawbacks can reveal as follows:

- Codebase size: As the amount of code and functionality grow, as is expected of a live software solution, other aspects such as compiling times, development time (directly and indirectly) and modularity take a toll as well.

- Deployment container overload: the larger and heavier the monolithic solution gets, the higher its computational requirements will be. Due to this, increasing costs can be expected to maintain the solution as responsive as possible.
- Scalability: Monolithic architectures can only be scaled over one dimension, which usually means duplicating or deploying a new instance with the same build or version. There is the impossibility to scale one of its components individually.
- Continuous deployment (CD): CD can be challenging as a change to any part of the monolithic solution will require the full solution to be rebuilt and redeployed which may carry high overhead times.
- Long commitment to a technology stack: By its nature, it's impossible to start a gradual transition to newly desired technologies to be adopted as part of the current technology stack. Workarounds need to be taken or a new solution from scratch needs to be developed to replace the old monolithic software.

2.2 Microservice Architecture

As one of the very first definitions of Microservice Architecture, James Lewis and Martin Fowler described it as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API". Furthermore. it was more recently described as "A microservice architecture decomposes a business domain into small, consistently bounded contexts implemented by autonomous, self-contained, loosely coupled, and independently deployable service" (Blinowski, Ojdowska, and Przybylek 2022).
This poses a change in the paradigm when compared to the more traditional Monolithic Architecture where large and high-complexity services/solutions were routine (Lauretis 2019). Microservice architecture makes use of its key features like bounded contexts, modularity and the Single Responsibility Principle (SRP) to decompose high-complexity applications into a subset of smaller components (microservices) which are easier to develop, manage and deploy in comparison with a single monolithic solution (Blinowski, Ojdowska, and Przybylek 2022).

2.2.1 Benefits

This type of architecture carries the following benefits according to (Blinowski, Ojdowska, and Przybylek 2022; Fowler 2019; Lauretis 2019):

- Maintainability: By breaking up complex logic or application functionality into a set of meaningful, independent and self-contained microservices contributes to high concurrency development across different microservices and a smaller size code base per service which eases the integration of new development team members and code understandability.
- Scalability: As each microservice is independent and self-contain, it means each of the solution microservice can be deployed in different machines as long as it does not present a challenge to its communication procedures. Functionality relevant to a set of microservices can be isolated and scaled differently to other solution parts depending on requirements.
- Repleacability: Due to the small size and independence of each of the microservice, if there is a need for complete re-development of a solution component, this is less harmful when using a microservice architecture when compared to a monolithic approach.

- Fault tolerance: Given a failure of a microservice, it will hardly threaten the whole solution's integrity. Microservices can be easily replaced or redeployed to work around these. Due to service size, redundant microservices can also be feasible to further strengthen the solution tolerance to faults.
- Technological freedom: Due to microservice's independency, programming language, frameworks and other technologies can be chosen freely. Due to high replaceability, microservices can be replaced by others to take advantage of certain technological advancements.

2.2.2 Challenges

Although the Microservice Architecture may seem a clear choice over a Monolithic Architecture for a new solution/application, this may depend on the scenario or problem its aims to solve.
When the solution scope and complexity are not high, the monolithic approach may make more sense as per its simplicity in developing and deploying. Microservice architecture may only bring more overhead and complexities that shade its benefits presented on Benefits in such a scenario.
Other challenges according to (Baraki et al. 2018; Blinowski, Ojdowska, and Przybylek 2022; Fowler 2019; Sampaio et al. 2017) are:

- Consistency: One clear example of such a challenge is data consistency. In a distributed system environment, the management of data consistency and reliability becomes an extremely complex task.
- Complexity: Effectively and efficiently splitting a business context into multiple bound contexts is not a trivial task, from both a monolithic to microservice migration or new solution perspective. Furthermore, there might be a need for a mature operations team and cooperation between multiple teams to ensure the multiple microservices that are constantly being added, replaced or redeploy, are being managed correctly.
- Distributed services: The development of distributed services is harder when compared to a single monolithic application. Microservices should be designed as dependent and fault tolerant as possible to make sure that the concurrent development and deployment do not become an actual drawback.

2.2.3 Patterns

In order to workaround or fully manage the challenge presented on Challenges and to reduce the complexity of a Microservice Architecture approach a set of agree patterns should be followed (Baraki et al. 2018).

API Gateway

In a Microservice architecture, answering end-user requests may require the aggregation of information across multiple microservice APIs. The API Gateway pattern aims to solve this by becoming the solution single entry point becoming responsible for the correct routing of the requests and its results aggregation (MicroserviceIO 2018a).
The API Gateway presents as another abstraction layer between the client application and the underlying microservice environment which further increments its independency (Baraki et al. 2018). In other words, instead of adapting and extending each and every microservice to fit client application needs, this responsibility can shift into the API Gateway layer/service which centralizes this processing and aggregation logic without impacting underlying microservices.

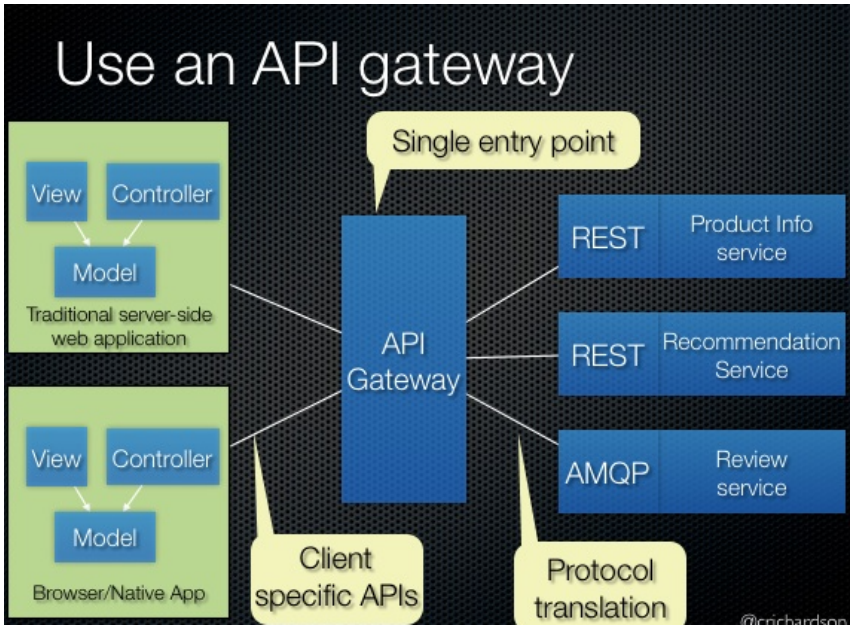


Figure 2.2: API Gateway Pattern (MicroserviceIO 2018a)

The main advantages identified by (Baraki et al. 2018; Fowler 2021; MicroserviceIO 2018a; Valdivia et al. 2020) for the API Gateway pattern are the following:

- Request Efficiency: As the API Gateway becomes the solution single entry point, the flow of information required to answer a specific request can be controlled and optimized to reduce the number of intra-microservice environment requests.
- Backward Compatibility: Changes to microservice interfaces and behaviour changes are isolated from external applications via API Gateway.
- Market-centric Architecture: Services can be easily modified. The extra abstraction layer (API Gateway) enables the solution to effectively answer distinct client requirements without sacrificing architecture evolution and maintainability.
- Ease of Extension: As all logic for external client application request handling is centralized on the API Gateway, the task to extend or adapt existing functionality/request carries less overhead when compared to this task being carried out in every single solution microservice.

Despite this, and as expected from any approach, API Gateways carry the following challenges:

- SPOF - Single Point of Failure: As a single point of entry for the entire solution, a failure to the API Gateway integrity can present as a complete system failure for external applications.
- Potential system bottleneck: Once more, as a single point of entry for the software solution, if not designed correctly and efficiently, it can become a bottleneck driving solution responsiveness.
- API Reuse: The reuse of the same API for different stakeholders needs to be carefully tracked. Functionality changes or extensions need to be applied taking into account these dependencies.
- Scalability: As the number of microservices grows, a single point of entry may no longer satisfy the software solution requirements and solutions need to be designed to handle this growth.

BFF - Backends for Frontends

As an alternative or variation of the API Gateway pattern, the BFF aims to create a specific back-end for each client type. Instead of a single point of entry as per API Gateway standards, a BFF API is developed per client type. This enables the solution to overcome some of the potential bottlenecks and SPOF drawbacks introduced by an API Gateway approach while still maintaining its benefits.
An example of such a pattern is the introduction of different APIs, each for a client-specific requirement, some can target browser application needs while others aim at mobile applications. A representation of such an example can be observed on 2.3

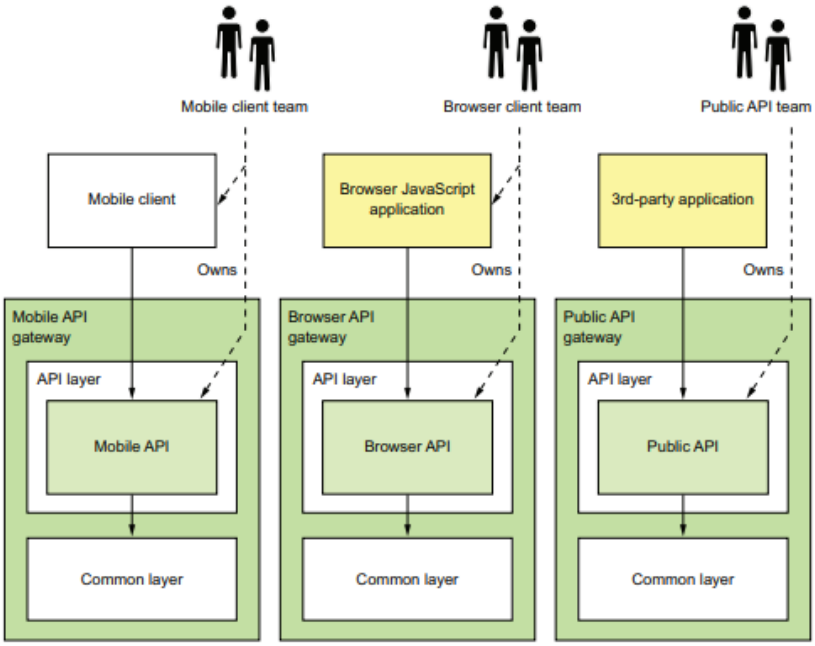


Figure 2.3: Backends for Frontends (Richardson 2018)

The main advantages identified by (Bhayani 2022; Richardson 2018; Valdivia et al. 2020) are the following:

- Backward Compatibility: Changes to microservice interfaces and behaviour changes are isolated from external applications via each of the BFF APIs.
- Market-centric Architecture: Services can be easily modified. The extra abstraction layers (BFF APIs) enable the solution to effectively answer distinct client requirements without sacrificing architecture evolution and maintainability. Furthermore, each BFF can adopt the most adequate technology to answer its requirements.
- Observability: As each BFF API represents a different set of requirements and functionality. Understanding each API output will give an accurate representation of the status of internal processes relevant to each BFF.
- Scability: Each BFF can be scaled independently from others depending on specific needs.
- Complexity: The solution entry point is no longer through a single point as in an API Gateway approach. This means that different business logic and complexity can now be split across BFFs. Furthermore, size and startup time can be reduced resulting from this decentralization.

Despite this, the BFF approach still carries some drawbacks:

- API Reuse: The reuse of the same BFF for different stakeholders needs to be carefully tracked. Functionality changes or extensions need to be applied taking into account these dependencies.

Database-per-service

Another important aspect and design of a Microservice Architecture is how data is stored and managed. Whereas a Shared Database Server Pattern centralizes all Microservices data on a single database, the Database-per-service pattern specifies a private database per microservice. An example can be observed on 2.4.

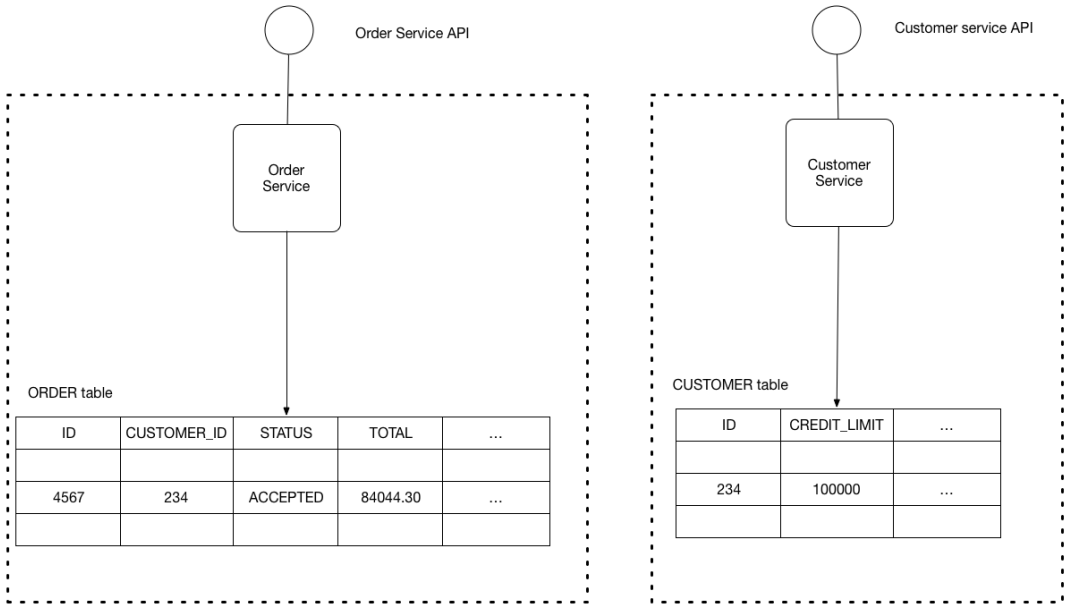


Figure 2.4: Database-per-service Pattern (MicroserviceIO 2018b)

By decentralizing the data of the whole architecture, according to (Baraki et al. 2018; Richardson 2018) some of the befits of the such pattern are the following:

- Loose coupling: To ensure microservices are as independent as possible, each should be independent on a data layer as well. If each of the services has its own database, this is ensured.
- Database technology: As each database is service-private, the database technology that better suits its needs can be adopted without constraining other microservices.
- Scalability: If system requirements demand the scaling of a given microservice, in a database-per-service approach, the database can be scaled on a service-by-service basis.

Despite this, by using this pattern in a microservice architecture, a number of new challenges and complexities arise:

- Data consistency: As data is now split between each of the microservice domains, there is a need to ensure data is consistent across the whole solution. The SAGA Pattern provides a way forward to overcoming such drawbacks by specifying a business transaction that spans multiple services as a saga, triggering a set of events on other services to ensure data consistency across the whole software solution. (MicroserviceIO 2018b; Richardson 2018)

- Infrastructure management: The number of individual data instances increases due to the growth of the microservice architecture thus requiring more resources to manage and maintain.
- Cross-service data aggregation: The implementation of the logic or queries to join databases across multiple databases is more complex than a single database approach.

2.3 GraphQL

On 2011, Facebook started building mobile native apps to meet their increasing mobile user base and overcome mobile browser limitations. Therefore there was a need to efficiently and effectively retrieve data that fit the mobile native apps needs. Consequently, Facebook developed and started the adoption of GraphQL 1.0 in 2012, and Facebook for iOS 5.0 used GraphQL as the API choice for the news feed (Jones 2022; Meta 2015).
GraphQL, unlike Java and Python, is a data-querying language. Declarative in its nature, it "provides significant performance and quality-of-life improvements over a REST API" (Apollo GraphQL 2022k). To achieve this, GraphQL exploits its queries to be utilized and sent to a server application, where, after being interpreted, a response is fed back as a JSON back to a client application. Other GraphQL properties comprehend (Meta 2015):

- Data specification: Responses follow the same structure as the GraphQL query which caused them. This not only allows some level of predictability of the response structure but also to solve issues on under-fetching or over-fetching by carefully tailoring the GraphQL query to client application requirements.
- Hierarchical: GraphQL hierarchical structure follows objects and its field types or nested types relationships, promoting more efficient usage of APIs as it avoids server-level data aggregation operations for client application-specific needs.
- Strongly typed: GraphQL is strongly typed, which means each query makes use of data types which can describe a set of available fields. This enables descriptive error messages.
- Data storage independent: The underlying database technology utilized in the server application is irrelevant to GraphQL, which makes use of existing code and simply specifies the data structure to be used for client-server communication.
- Introspection: A GraphQL server allows for supported data types queries.
- Backwards compatibility: Data specification is solely determined by the client GraphQL query, therefore additions or deprecation of specific server-side fields can be managed without significant disruptions.

2.3.1 Types

As a crucial unit of any GraphQL schema, types can be of six different kinds named type definitions - Scalar, Object, Enum, Interface, Union and Input (GraphQL 2019b)
Scalar and Enum types can be understood as the leaves of the GraphQL hierarchy tree, while Object types sit at the intermediate level by making use of Scalar, Enum and even other Object types.
Interface and Union are specified as abstract types. An Interface type defines a list of fields. Whenever another Interface or Object type implements an Interface, it guarantees that the defined list of fields is implemented as well. Union defines a list of possible types. Whenever a Union is claimed by other types, it guarantees that one of the possible types from the defined list will be returned.

Last but not least, Input types are used to differentiate what can be used as input to arguments and as output by fields. Scalar and Enum can be used as Input or Output types.

2.3.2 Fields

Object types can use other types through their fields. For example, on 2.5, the Person Type has three fields in total, each with a different type. The field "name" and "picture" are simple Scalar types but "picture" is of "Url" which is another Object type.

Example № 43

```
type Person {
  name: String
  age: Int
  picture: Url
}
```

Figure 2.5: Example Object Type - Person (GraphQL 2019b)

Fragments represent also comprehend a set of fields and are extremely useful to reduce schema or query duplicated text by introducing reusable fragments to represent commonly used concepts (GraphQL 2019b).

2.3.3 Operation types

GraphQL supports three types of operations - Query, Mutation and Subscription. Each of these has a different purpose to either support read-only data fetch, write operations or even event-based fetches.

Query

The query is a read-only fetch operation which takes full advantage of GraphQL declarative nature by providing full control of what is fetched from the server application. Moreover, it's the server's responsibility to then join and prepare the required information according to the Query specification (Apollo GraphQL 2022k).
For example, on 2.6 the query "hero" returns a "Character" type which has the fields "name" and "appearsIn". The query done by the client application can choose to either retrieve both fields or the character name only depending on what is specified on the GraphQL query.

```
type Query {
  hero(id: Int): Character
}

type Character {
  name: String!
  appearsIn: [String]
}
```

Figure 2.6: Example Query Definition (GraphQL 2022f)

Mutation

Mutations are used conventionally to represent and realize write operations on server applications. Just like the Queries, Mutations allow for flexibility on what information is feedback to the client (GraphQL 2022e).
Furthermore, multiple fields are also supported but executed sequentially in a Mutation operation while parallelly in a Query. Thus removing the opportunity for race conditions to exist for server application resources (GraphQL 2019b).

Subscription

Finally, GraphQL supports Subscription operations. Subscriptions can be described as "long-lasting operations that can change their result over time" (Apollo GraphQL 2022j).
GraphQL Subscriptions can maintain an active connection to the server allowing for low-latency, real-time updates to the client application. These should not be used as the widespread solution for server-client synchronization and applied to specific scenarios where requirements demand such an approach. Server-client synchronization can make use of recurrent polling that retrieves information in configurable time spans adequate to the level of synchronization required. An example Subscription definition can be observed on 2.7

```
type Subscription {
  commentAdded(postID: ID!): Comment
}
```

Figure 2.7: Example Subscription Definition (Apollo GraphQL 2022j)

2.3.4 Advanced Properties

After going over the GraphQL various operation types and schema elements such as types and fields, some other properties are introduced at the beginning of Section 2.3.

Execution Strategies

An Execution strategy specifies the methodology or plan utilized to execute GraphQL operations. Strategies are subject to three characteristics (Roksela, Konieczny, and Zielinski 2020):

- Parallelism: Strategies can be split between a serial or parallel approach.
- Synchronicity: Strategy categorization depends upon a synchronous or asynchronous approach to operation fields execution
- Data access management: Batching or Caching can be exploited as alternatives to data fetching.

Some GraphQL implementations such as "graphql-java", which corresponds to Java language implementation, offer a set of built-in strategies (Java 2022):

- AsyncExecutionStrategy: Allows for fields to be fetched asynchronously, although query field order is maintained as provided by the client application regardless of field execution and fetching completion.
- AsyncSerialExecutionStrategy: GraphQL spec states that Mutation type operations must be executed sequentially to avoid race conditions for server resources. This strategy is ideal for this operation type as it assures that fields are executed and fetched according to the original field order.
- SubscriptionExecutionStrategy: As its name suggests, ideal and default for Subscription operation types making use of reactive-streams APIs and "Publisher"/"Subscriber" interfaces (Streams 2022).

Caching

Unlike a REST API, where client applications can use HTTP caching, as any other endpoint-based API, in a GraphQL scenario there is no concept of URLs to extrapolate a global identifier which allows for globally identifying each request. This presents as a challenge in the provisioning of a caching solution in GraphQL (GraphQL 2022a).
Globally unique identifiers are presented as a solution for such issues. This pattern states that a field, for example, "id", should be reversed as a globally unique identifier to enable a caching solution around using these identifiers. Optionally, tools like the "Global Object Identification" can reuse such Globally unique identifiers concepts by aligning its standardized logic to expose object identifiers (GraphQL 2022b). Concerns regarding API compatibility as the reserved field may be used differently by other APIs. Situationally, this may justify an extra identifier field which will store other APIs globally unique identifiers (GraphQL 2022a).

Batching

Alternatively to the caching approach, assuming a naive GraphQL service, a high number of requests can cause data to be loaded from the database constantly which diminishes the service efficiency and responsiveness. This can be solved by opting for a batching approach to your request resolution. In this technique, requests are grouped for a short period of time, after which they are combined and submitted as a single request to a defined underlying service or database via a data loader (GraphQL 2022c). Facebook created its own DataLoader, not only to be utilized in this scenario but also expectation and hopeful thought that their implementation can be used as a reference for other language-specific data loaders (Facebook 2023).

Introspection

GraphQL services can be developed in such a way that Introspection is supported. This comprehends the ability to query the GraphQL service for any type, type interfaces, enumerations and even system documentation. Type names that have the prefix "__" are a part of the introspection system (GraphQL 2022d). An example of type introspection can be found on 2.8, where the type "Droid", its fields and field types are exposed via the introspection system.

Figure 2.8: Example Subscription Definition (GraphQL 2022d)

2.3.5 Best practices

GraphQL specification can be silent or not take a stance on some common API challenges and approaches. This is intentional in its nature and does not correlate to a lack of solutions for these problems within GraphQL. The main reasoning for this is that these suggestions or opinions, about optimal approaches, are understood as out of the GraphQL specification scope (GraphQL 2022c). Despite this, a set of commonly used best practices can be listed as follows:

- HTTP: Being the most common protocol for server-client communication when opting for a GraphQL implementation. Unlike REST APIs, GraphQL APIs operate under a single endpoint which is usually "/graphql", therefore any queries or other types of requests should be directed at this specific endpoint. A detailed specification of "GraphQL over HTTP" is currently under development (GraphQL 2023), adoption of such specification is not mandatory but recommended to standardize HTTP communication within GraphQL implementations, thus maximizing interoperability (GraphQL 2022g).
- JSON: Although not required by the GraphQL specification, JSON is usually utilised as the response format. Due to its popularity, widespread use by client and API developers, readability and easy compression through the use of GZIP, JSON comprises an excellent choice for a GraphQL implementation. By observing some schemas, queries, types and other GraphQL concepts it's noticeable the inspiration took from the JSON syntax GraphQL 2019a, 2022c.
- Versioning: Similarly to a REST API, a GraphQL API can be versioned. Despite this, GraphQL offers a set of tools to enable API continuous development. API Versioning is usually related to the API being outdated or no longer able to support client requirements, most commonly, non-GraphQL APIs suffer from issues from its data fetching being fairly static and any changes requiring an API redeployment. However, GraphQL

data fetching capabilities are fairly dynamic, allowing for new type definitions or even for the client to change its queries depending on what information is required from a given GraphQL service (GraphQL 2022c).

- Nullability: Null values usage is something commonly used or recognized in type systems. In a GraphQL-type system, every field is nullable by default. This allows for a better failure tolerance as any issue regarding network connection, database access or asynchronous processing can result in Null values. Extra development and thought processes need to go into the usage of such values as sometimes "Null" may not be the best value for a field that suffered a failure. For example, in a specific scenario where a list is expected, an empty list is returned as a default for any failure event, instead of a "Null", which may be more adequate (GraphQL 2022c).
- Pagination: Although GraphQL type system supports lists of values, pagination of such lists is understood as within API designer authority (GraphQL 2022c). Tools like the Relay project (Relay 2020a,b), allows GraphQL clients to "consistently handle pagination best practices with support for related metadata via a GraphQL server". Furthermore, the specification advises this new pattern of pagination to be referred to as "Connections" and to expose them in a standardized fashionRelay 2020b.
- Server-side Batching and Caching: As explained on 2.3.4 and 2.3.4, some kind of caching solution or batching of client requests is recommended. Disregarding these will impact GraphQL service performance and efficiency, more noticeable as the number of requests scales by its GraphQL clients.

2.3.6 Trade-offs

As with any other technology, approach or even architecture, GraphQL has a set of drawbacks that should be carefully weighed against its advantages. Only by doing this exercise, a developer or any system stakeholder will be able to determine if GraphQL is the right way forward. Some of the drawbacks identified by (Apollo GraphQL 2022k) when using GraphQL are the following:

- Change management: As GraphQL implementation represents a change in how data is represented, interacted and retrieved, this implies a new learning curve in the organization teams and API stakeholders. Furthermore, if a monolithic GraphQL server is being used to manage all organization data fetching needs, a considerable amount of collaboration across the organization developers is required to share and maintain such schema.
- Slow operations potential: The GraphQL Schema defines how the client may query the GraphQl server, and the latter defines how to fetch the data from the underlying data sources. Therefore, there is a need to be proactive and detect any client queries which may cause high usage operations to be carried on the GraphQL server and possibly overload it. Implementation of some type of tracing is recommended to have a clear view of server performance. Apollo Tracing (Apollo GraphQL 2022c) offers some capability for performance tracing for GraphQL servers using programming languages such as Node.JS, Ruby, Java and others (Walraven 2017)
- Incompatibility with web browser caching: Web browser automatic caching is based upon URLs. As GraphQL uses a single endpoint as the point of access to the GraphQL server, URL-based caching is no longer viable for a GraphQL implementation.

Chapter 3

State of art

Throughout this Chapter, there is deep dive into the document target technologies, GraphQL and GraphQL Federation, aiming to establish an important contextualization and identify any challenges or drawbacks that might drive the evolution from one to another. Some of the critical focus points include caching, batching, execution strategies and a set of particular enterprise's use cases (Netflix, Adobe and Walmart) to observe, extract and identify some best practices or architectural decisions that may be important for a performant GraphQL Federation architecture.
Lastly, an overview of the quality attributes is documented, particularly aiming at the performance quality attribute.

3.1 GraphQL Federation

As mentioned on (2.3.6), GraphQL still carries a number of drawbacks. These become increasingly perceived as each solution matures and grows in complexity over its lifetime. As discussed in chapter 3.1.2, some patterns are commonly applied to a non-federated and unconsolidated GraphQL architecture to get around some of these challenges.
The transition from GraphQL to GraphQL Federation is pretty similar to Monolithic to Microservice Architecture. Like the Monolithic Architecture, GraphQL standalone services are easier to develop as the whole solution GraphQL types, fields and objects can be defined in a single, centralized schema. This is especially adequate when the business requirements, processes, data structures or domain are not too complex and the solution does not require a massive request-to-response capability.
As the solution evolves, the business grows or the development team increases, the monolithic approach turns into a drawback or a bottleneck in terms of maintainability, scalability and other aspects as previously specified in chapter 2.1.2. This may justify a Microservice architecture approach using now multiple GraphQL services, but, as discussed in chapter 2.2 this comes with a set of new complexities.
Some of these challenges correlate almost directly with the identified drawbacks of a solution containing a set of GraphQL services. Data consistency, higher development complexity, and higher coordination requirements between service owners and others can be appeased with the consolidation of the multiple Graph and through solution federation to GraphQL Federation (Apollo GraphQL 2022e).

3.1.1 Architecture

The GraphQL Federation architecture describes a supergraph, which, in turn, communicates and aggregates other GraphQL APIs (subgraphs). An example of such architecture can be found in Figure 3.1.

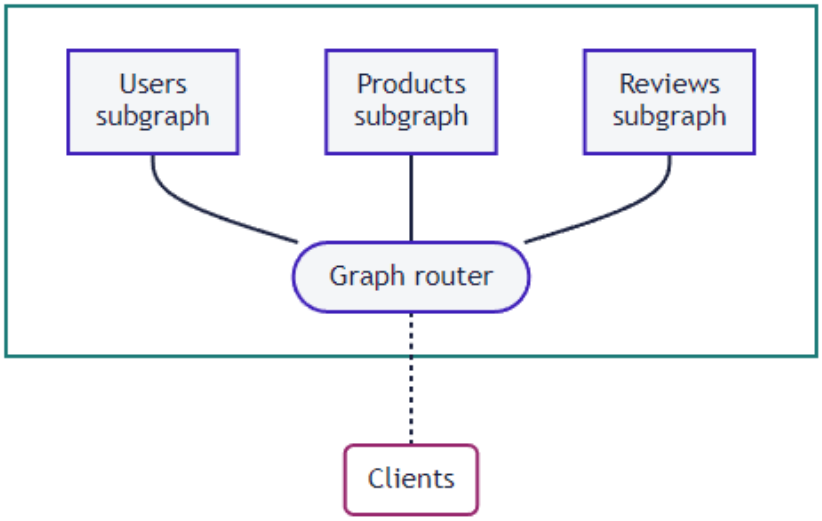


Figure 3.1: Apollo Federation Architecture (Apollo GraphQL 2022d)

The Graph router or gateway is the entry point for the supergraph which routes the requests received by any client application through the subgraphs. From a client application standpoint, this architecture behaves as a single GraphQL server, allowing for a single request to retrieve information from multiple subgraphs (Apollo GraphQL 2022d). This allows for shared ownership of the supergraph, and, even if a solution only holds a GraphQL API currently, Apollo Federation will be a useful tool to handle your GraphQL APIs as the solution grows. On 3.1.2, other alternative patterns and methodologies are explored and analysed.
Some other advantages of using Apollo Federation include (Apollo GraphQL 2022d):

- API Unification: Apollo federation architecture allows for the client application to interact with the full solution functionality through the Graph router. Alternatively, clients will be required to use multiple endpoints to interact with different system functionality if the Apollo Federation layer is not present. Furthermore, to improve client experience, some level of standardization needs to be introduced to each of the GraphQL APIs.
- Avoid Monolithic GraphQL: Despite its simplicity, a monolithic GraphQL server as the single access point for all back-end services may not be able to scale according to organization, functionality and user needs.
- Separation of concerns: In (Stünkel et al. 2020) Apollo Federation is described as intrusive. This is due to the fact that each of the subgraphs and its schemas has to be further developed with Apollo Federation DSL new annotations so that can integrate seamlessly with the supergraph. Despite this, Apollo Federation allows for the solution to adhere to the separation of concerns pattern when developing the individual GraphQL services which compose the supergraph.

Entities

As a core element and building block of an Apollo federation graph, entities can be defined in a specific subgraph and further utilized or extended by others. While defining a new entity, it's important to use the "key" directive to define what is the primary key, which can be either a single or a set of fields and their types.
Entity extension may be meaningful for subgraphs other than the entity originating one to include essential or related information to it. Furthermore, it is necessary to understand that

the originating subgraph entity has no knowledge of such extensions (Hampton, Watson, and Wise 2020), which may imply proper documentation and communication of such extensions to ensure these cross-graph dependencies do not become a future problem.

3.1.2 Unconsolidated Patterns

GraphQL was often adopted in architectures where client applications query a single GraphQL server, which in turn aggregates data from the underlying back-end services (Apollo GraphQL 2022f). As this architecture evolves, new GraphQL servers can be created as a result of new client application needs or different organization teams or departments starting the adoption of GraphQL for their services. Apollo documents a set of four patterns for managing new GraphQL implementations that make adaptations to the starter architecture, prior to GraphQL consolidation.
Despite these, a common shortcoming identified by Apollo is the inconsistency resulting from such patterns (Apollo GraphQL 2022f). Data fetching consistency from both the client point of view and how common entities are represented are key to improving the efficiency and understandability of the solution GraphQl-based architecture.

Client-only GraphQL

With a great focus on client-side architecture, this pattern comprehends the implementation of GraphQL API by client teams within its own application context. The main driver for this decision is to improve the client development experience by introducing this new abstraction layer between the client and the upstream REST or any other kind of services required. The implication of such an approach is that the client application is now responsible for the data aggregation, and all the requests it requires from said upstream services, thus incurring performance costs. An example of such pattern architecture can be observed on 3.2

Figure 3.2: Example Client-only Architecture - GraphQL (Apollo GraphQL 2022f)

Backend for Frontend

GraphQL can be used as the implementation language for the Backend for Frontends (BFF) pattern.
As explained on 2.3, BFF services describe a new layer of abstraction between the client and backend applications that is specific to that client's requirements and experience. In each of these new abstraction layers, client-focused services would be implemented in GraphQL taking advantage of its data-fetching capabilities.

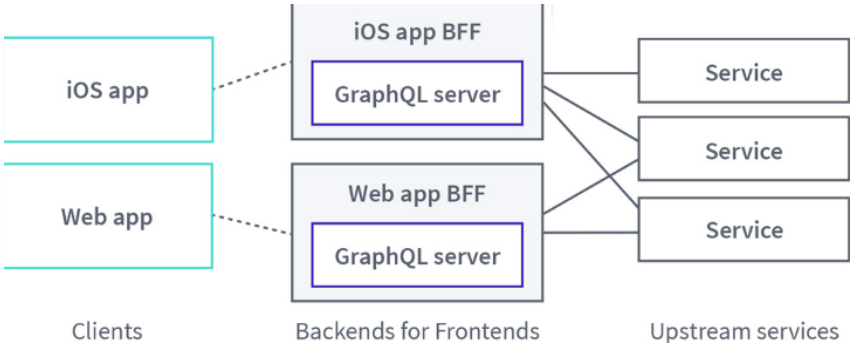


Figure 3.3: Example Backend for FrontEnd Architecture - GraphQL (Apollo GraphQL 2022f)

Despite the fact that this seems like an optimal combination between GraphQL and BFF, building and maintaining a BFF service for each client or set of client needs has some implications. One of these is the duplication effort across the development teams, which develops each BFF as an isolated unit from others.
Alternatively, with similar clients, the reuse of the same BFF service across may be appealing, but, as each of these clients' needs and requirements evolves, the complexity and size of the underlying schema increase without control, as there is no clear owner of such shared BFF service.

Monolithic Architecture

The monolithic pattern is described as having two forms (Apollo GraphQL 2022f).
First of which is characterized by the maintenance and operation of a single GraphQL server and codebase that is shared by multiple organization teams and used by a set of clients. Despite the codebase being organized, just like the BFF pattern, there is no clear ownership of this GraphQL component.
The second form is characterized by the maintenance and operation of a single GraphQL server, but, this time around by a single organization team. This team have clear ownership of the monolithic GraphQL and determines a set of standards to be followed to drive its adoption.

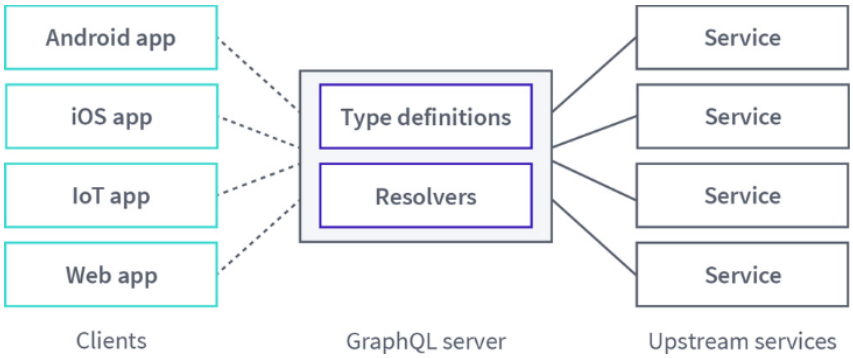


Figure 3.4: Example of Monolithic Pattern - GraphQL (Apollo GraphQL 2022f)

Although such a pattern can help set the stage for the effective consolidation of an organization's GraphQL efforts, there is a set of drawbacks for both forms.

Taking into consideration the first form, where the monolithic server is managed by multiple teams, its shared ownership model implies some challenges. Namely, managing and adapting the service to serve multiple client needs may be problematic, and require workarounds for some scenarios, Furthermore, the standardization of such service is complex as the graph is extended and adapted on a client-to-client basis.
Lastly, considering an organization's single team managing the monolithic GraphQL service, the responsibility of maintaining such a solution can become unbearable for the team responsible. Challenges like building and evolving GraphQL schema for new client needs without compromising existing client functionality and data fetching capabilities, can prove unfathomably complex and require too many resources and time to be actioned upon.

Overlapping Graphs

This pattern is characterized by the existence of multiple graphs within the same enterprise. This can be a result of the deliberate choice of each GraphQL API being managed independently or a consequence of the gradual adoption of GraphQL by the organization (Apollo GraphQL 2022f). This pattern takes on similar drawbacks as the BFF approach, namely the duplication of effort across multiple development teams. Additionally, by deliberately creating a new Graph per GraphQL API, standardization across the multiple APIs may not be managed correctly, creating a less interactive experience for the clients, which no longer interact with a unified and normalized set of APIs.

3.1.3 Consolidation Decision Matrix

Apollo Consolidation Decision Matrix is an artefact recommended by Apollo for practitioners to reliably measure GraphQL adoption and evolution to a federated approach (Hampton, Watson, and Wise 2020).

Table 3.1: Consolidation Decision Matrix (Hampton, Watson, and Wise 2020)

Concern	Criterion	Yes/No	Remediation/Guidance
Consensus	Are multiple teams contributing to your graph?	Yes/No	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?	Yes/No	If teams are collaborating well together, consider the potential switching cost of diving teams or adding new teams.
Delivery	Is there a measurable slowdown or downward the trend in GraphQL service change delivery?	Yes/No	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?	Yes/No	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?	Yes/No	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.

Table 3.2: Consolidation Decision Matrix (Hampton, Watson, and Wise 2020)

Concern	Criterion	Yes/No	Remediation/Guidance
Consensus	Does your GraphQL governance process have a reasonably robust education component to on-board new teams?	Yes/No	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?	Yes/No	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?	Yes/No	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?	Yes/No	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?	Yes/No	Ensure that the requirements for separating concerns have a performance and optimization budget.

Several conclusions can be extracted depending on the answers provided for each concern and criterion pair Hampton, Watson, and Wise 2020:

- If the answers are all "yes", the output of such a matrix is a go-ahead for the successful implementation of a federated architecture.
- If any of the answers is a "no" or not quite agreed upon, extra work needs to be done to ensure these can be turned into "yes" or close monitorization to ensure these do not present as showstoppers for a federated architecture transition.

3.1.4 Federated schema best practices

As a result of GraphQL Federation widespread use and documented approaches (Sub-Section 3.2), challenges and lessons learned, a set of best practices emerged. A number of these best practices can be found even in non-federated/unconsolidated patterns as documented on 3.1.2 but still are relevant when designing graphs within a federated architecture (Hampton, Watson, and Wise 2020).
As documented on 3.1.1, Entities represent an essential part of any graph within a federated architecture. Therefore, carefully planning which entities are meaningful for the solution, and how these are referenced and extended by other graphs is essential in ensuring low coupling between the different subgraphs and a clear separation of concerns. As each federated architecture is a living, growing and evolving software solution, this exercise will be ongoing throughout the solution life cycle to ensure service/subgraph boundaries are maintained and respected.
Some other best practices identified are the following (Hampton, Watson, and Wise 2020):

- Demand-oriented, abstract schema design: Schema should be designed aiming for the client's needs and not for an individual client. Tightly coupled services/subgraphs and implementation details should be avoided following Agility principles (Schmidt and DeBergalis 2022). Furthermore, schema data to be exposed and treated should correlate to the client's processes and use cases to avoid exposing extra information for which the client has no use and may reveal implementation details.
- Schema Expressiveness: Proper schema documentation, standardizing naming and formatting conventions, and designing fields relevant to specific use cases contributes may be critical for graph maintainability. Such best practices can be translated or correlated to a standardized and consistent API which is of high relevance as documented on 3.1.2 through the analysis of each pattern's shortcomings.
- Nullability: Although all fields are nullable by default, some fields may be non-nullable from a business or functionality standpoint (using "" on the schema definition), meaning that client applications expect these to contain values when querying through the solution. Making these nullability decisions is incredibly critical, as non-null fields can lead to extra challenges when maintaining and evolving the schema. Furthermore, minimizing nullable arguments and input fields is highly advised as transitioning from a nullable input field to a non-nullable state, or vice-versa can introduce breaking changes depending upon any directly or indirectly dependent business processes and logic (Hampton, Watson, and Wise 2020).

3.1.5 Performance

As explored previously in Section 2.3.4, some concepts such as Caching, Batching and Execution Strategies can drive the GraphQL performance while dealing with client queries.

These can be equally applied to each of the subgraph GraphQL servers, but there is a need to understand what GraphQL Federation offers in these areas.

Caching

Caching can be customized in GraphQL Federation through the use of cache hints. To be able to use cache hints the required definition must be included in the target subgraph schema as depicted in Figure 3.5.

GraphQL

Copy

```graphql
enum CacheControlScope {
  PUBLIC
  PRIVATE
}

directive @cacheControl(
  maxAge: Int
  scope: CacheControlScope
  inheritMaxAge: Boolean
) on FIELD_DEFINITION | OBJECT | INTERFACE | UNION
```

Figure 3.5: Cache Hints Definition (Apollo GraphQL 2022h)

These cache hints can be included at multiple levels and not exclusively in the actual GraphQL schema. It is possible to dynamically define the cache hints at the subgraph resolver level, which combined with the subgraph cache hints will be sent to the gateway level (Apollo GraphQL 2022i). Once at the GraphQL Federation gateway level, the overall cache hint for a response will be calculated. The calculation is done based on "the most restrictive settings among all of the responses received from the subgraphs involved in query plan execution" (Apollo GraphQL 2022h).

Query Plans

Another important concept of the GraphQL Federation architecture and query resolution is its query plans. Query plans detail how to populate each field of data for a set operation, it constitutes a blueprint for dividing an incoming operation into a set of others, each resolved by an individual sub-graph within the federated architecture. An example of a query plan structure can be observed in Figure 3.6 (Apollo GraphQL 2022g).

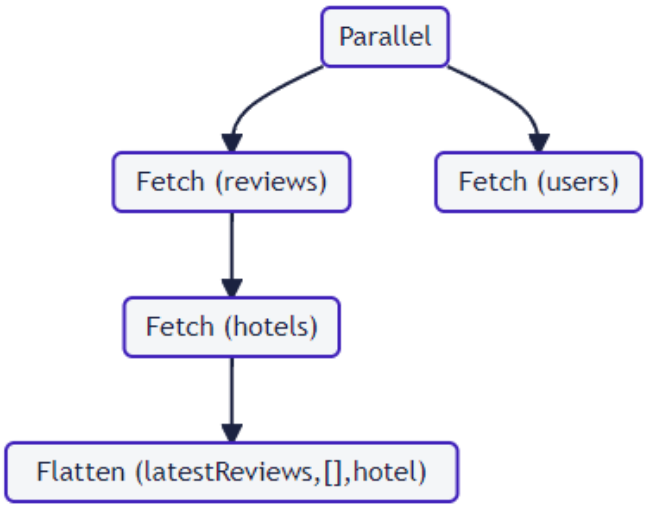


Figure 3.6: Query Plan Structure Example (Apollo GraphQL 2022g)

As perceived from the example, a query plan consists of a hierarchy of nodes. The top-level node will always be a QueryPlan node, after which each node can assume one of the following types:

- Fetch: Describes a specific operation on to a specific service or subgraph.
- Parallel: This type can be used to specify that the node's immediate children can be executed in a parallel fashion.
- Sequence: This type can be used to specify that the node's immediate children have been executed in a sequential fashion.
- Flatten: Always within a parent Sequence node, it specifies that the data from its Fetch node will be merged with the rest of the Sequence node's data.

3.2 Enterprise Approaches

Through the development of this section, a subset of enterprise case studies will be described that utilised a federated approach to GraphQL consolidation with the help of different Apollo platform pieces (Apollo GraphQL 2022f).

3.2.1 Netflix

Netflix's use case is one of the most known success cases of the GraphQL Federation. Being known for its loosely coupled and highly scalable microservice architecture, which allows for independently evolving and scaling services, Netflix offers a unified API aggregation layer to reduce the complexity of cross-service use cases.
So that both the UI developer's simplicity of a single conceptual API and the back-end developer's decoupling and resilience offered by the API layer could be met, Netflix developed a federated GraphQL platform to support the API layer. Besides, such an approach allowed for increased consistency and development efficiency while marginal losses were detected on the scalability and operability fronts (Shikhare 2020).
Netflix Studio Ecosystem scope comprehends the whole process from the TV show or movie pitch to when it's widely available through the Netflix platform. Such space architecture

started suffering from the growing complexity of the data and its relationships, which were addressed by the products teams by the following architectural patterns and drawbacks:

- Single-use aggregation layers: Due to the loose coupling that is natural in a microservice architecture, there was a duplication of effort by different teams developing a common purpose data-fetching code and aggregation layers.
- Materialized views: Through the use of materialized views (Microsoft 2020), teams were able to extract data from other services to match their specific system needs and format. Although no performance issues were raised from such an approach, data consistency thrived being the "top support issue in Studio Engineering in 2018" (Shikhare 2020).

Studio API

To overcome such challenges, the Netflix team started building a carefully organized graph API "Studio API". Studio API relied on GraphQL as its API technology, and through the unified abstraction layer on top of the data and its relationship, it allowed for easier access to cross-service shared data. Data consistency issues were also reduced, as each field on GraphQL is resolved by a single piece of data fetching code, also known as "Resolvers" (GraphQL 2020). The Studio API architecture is depicted in Figure 3.7.

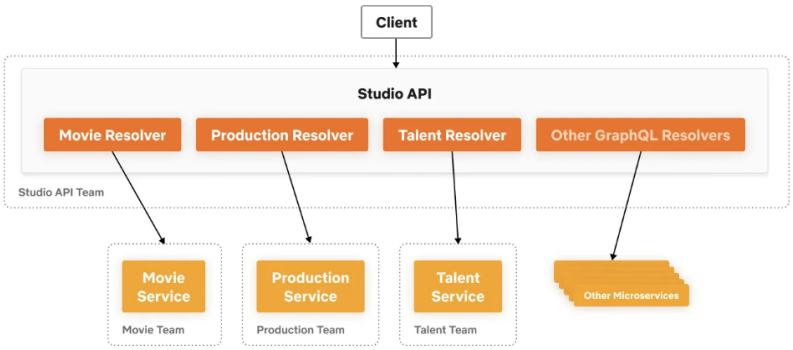


Figure 3.7: Studio API Architecture - Netflix (Shikhare 2020)

Despite being a success across Netflix's product teams, Studio API suffered from a few drawbacks:

- Disconnected from the domain expertise: The Studio API was focused on the development of the API and overcoming past challenges but disconnected from the domain expertise and product needs which took a toll on the schema's health.
- Growing the API: Integrating the new back-end into the Studio API was a manual process disregarding any rapid evolution premises promised by a microservice architecture.

Studio Edge

To maintain a unified GraphQL schema while decentralising the implementation of the resolvers to each domain expert team and the release of the GraphQL Federation Specification (Apollo GraphQL 2022a) by Apollo in early 2019, Netflix's new iteration over Studio API, "Studio Edge", materialized using GraphQL Federation as one of the critical evolution points. The architecture of Studio Edge can be observed in Figure 3.8.

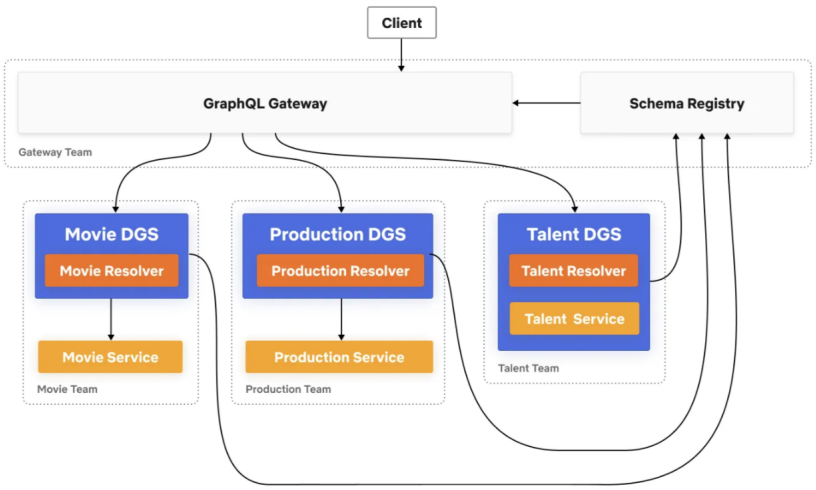


Figure 3.8: Studio Edge Architecture - Netflix (Shikhare 2020)

The following architectural components have been highlighted by Netflix:

- Domain Graph Service (DGS): Each DGS corresponds to a standalone GraphQL service where developers define their own federated GraphQL Schema. The DGS is owned and maintained by the team that holds the domain expertise and is responsible for that subsection of the overall API.
- Schema Registry: Stores all schemas and schema changes from every DGS. The registry exposes CRUD APIs for schemas, which are used by other developer tools or CI/CD pipelines. Responsible for all the schema validation requirements and the aggregation into the single unified schema to be provided to the GraphQL Gateway.
- GraphQL Gateway: Responsible for serving GraphQL queries to the consumers, composing and executing the query plan by querying the necessary DGS services.

Implementation Details

Netflix implementation also comprehends three main business logic components that "power the core of a federated API architecture" (Shikhare 2020):

- Schema Composition: This process describes the aggregation of all DGS schemas into a single unified schema. Whenever a new schema is pushed by a DGS it's assessed regarding its validity as an individual schema, backwards compatibility and compatibility with other DGS schemas before being checked into the Schema Registry for further use.
- Query Planning and Execution: The query plan is the result of the combination of the information from the unified schema, federation configuration and client query. It aims to split the client query into multiple sub-queries that are sent for the relevant underlying DGS. Executions can be either parallel or in sequence depending on the compiled query plan. Netflix also specifies that the Query Planning and Execution "adds a 10ms overhead in the worst case. This includes the compute for building the query plan, as well as the deserialization of DGS responses and the serialization of merged gateway response" as a note on the performance of such component (Shikhare 2020).

- Entity Resolver: By making use of GraphQL Federation key directive, and by each DGS exposing its federated types the GraphQL Gateway can join federated types across multiple DGS asynchronously seamlessly.

More information regarding the Netflix use case can be found in the Netflix blog posts, both part one and part two (Shikhare 2020).

3.2.2 StockX

StockX shares 9 of the lessons learned from the usage of GraphQL Federation throughout the year. Although there is an understanding of the extra challenges inherited from GraphQL Federation, as with any other distributed system, StockX's overall experience is positive (Schrade 2020). Some of the most relevant points from this particular enterprise use case are the following:

- Schema: StockX gives a great deal of importance to the iteration approach to schema building and how important schema documentation can be to increase the API understandably and usability.
- Performance: GraphQL Federation brought a great performance improvement to StockX. Payload sizes were reduced up to 7 times which allowed for many other metric improvements in mobile devices. The shift to GraphQL allowed StockX to unify its caching strategy, and therefore reduce the requests to the backend services.
- Scalability: To handle large traffic spikes, StockX made use of GraphQL Federation architecture possibility to scale individual services to avoid having to scale the whole architecture, which would incur extra costs onto StockX.
- Development: StockX also reported an increase in their development speed, as now each team can independently transverse a specific part of the federated graph without requesting permission from the team that owns it, due to the graphical nature of GraphQL schemas, and all the documentation developed as part of the previously mentioned point.

Finally, StockX describes the major workload of such GraphQL Federation migration/transition and how it revealed a good choice and a success, as perceived from the above points and others in the StockX blog post (Schrade 2020).

3.2.3 Adobe

Aside from the benefits of external consumption, employing GraphQL at Adobe has given the UI engineering teams a solution to deal with the problems of the increasingly complex world of distributed systems.

As previously seen in Section 2.2 breaking services down into smaller, logically concern-separated and independent services make up for a great set of advantages as experienced by Adobe (Bremner 2021). Despite this, extra layers, more communication channels and other techniques had to be applied to get access to such benefits.

GraphQL has been flagged as "a key component for the Adobe Experience Platform user experience engineering team: one that allows us to embrace the advantages of SOA and helps us to navigate the complexities of microservice architecture" (Bremner 2021).

Advantages from GraphQL on Adobe Experience Platform

Several advantages have been detailed by the author of (Bremner 2021) in Adobe:

- Easy learning curve

- IDE integration capabilities
- Improvements to the UI engineering team agility and velocity
- Client application performance
- Cross-service discoverability, standards, semantics, and validation

Furthermore, great performance on the operations front has been documented - Zero downtime and outages, low CPU usage and low memory usage. Finally, it allowed for a greater level of team collaboration across the organization with over 40 internal contributors, 10-30 commits on a weekly basis and over 40 API endpoints integrated and in current use.

GraphQL Challenges

Despite the advantages inherited by GraphQL usage, several challenges have been identified (Bremner 2021):

- One Graph: As multiple Adobe teams adopted GraphQL, each at a different adoption stage, some objects and types were identified as useful across multiple teams. Therefore a requirement to "converge", unify or consolidate the multiple graphs into a single unified graph materialized. Several options were analysed to achieve this, but, ultimately, Adobe opted for GraphQL Federation which allowed for each team to carry its own work, while a single gateway understood and carried out any of the federated queries between the multiple team instances.
- Graphs: Shift in methodology to graph-based APIs. Data can now be queried without structure constraints and there is no need for multiple alternative ways to retrieve a set of data, as this can be achieved by a single GraphQL query.
- Versioning: As discussed on Section 2.3.5, Versioning is no longer required in a GraphQL architecture. GraphQL allows for several dynamic procedures which remove the necessity of invasive versioning and version deployments. Adobe took to heart the GraphQL specified best practices by avoiding versioning themselves. Instead, they relied on schema "depreciation markers" and a comprehensive logging system to support the identification and resolution of any outdated functionality or field.

3.3 Performance Efficiency

As the work documented through this document understands the analysis of one of the quality attributes, this topic is critical to contextualize its standards and definitions.
ISO 25010 is the foundation of a product quality evaluation system. The quality model, as depicted in Figure 3.9, determines which quality attributes will be the aim of an evaluation.

Figure 3.9: Product Quality Model (ISO 25000 2011)

Performance efficiency also described as "performance relative to the amount of resources used under stated conditions" is the characteristic that will be focused on throughout this thesis and it's composed of three sub-characteristics (ISO 2011; ISO 25000 2011):

- Time behaviour: How the system or product meets its requirements when analysing features like response time, processing time and throughput.
- Resource utilization: Degree to which the system or product utilizes its resources when performing its usual functionality.
- Capacity: How the system max capability compares to its requirements.

Chapter 4

Value analysis

Value analysis refers to a systematic process of analysis and evaluation which aims to increase the value of a given item, product or service while not further restraining it of any other specific qualities. The value will be analysed via the Innovation process using the New Concept Development (NCD) model, leading to the idea genesis and selection stages. To support the selection of the best alternative, the Analytic Hierarchy Process (AHP) will be applied through a set of defined criteria.

4.1 Innovation Process

Now, more than ever, there is a need to effectively manage an enterprise's resources, products and clients driven by continuous technological advancements, and ever-growing consumer needs/desires. As a way forward for this management challenge, and to ensure the development of a valuable and long-lasting product, the multistage method understood as the innovation process was designed.
This process can be divided into three sequential steps: Fuzzy Front End, New Product Development, and finally Commercialization as observed on 4.1:

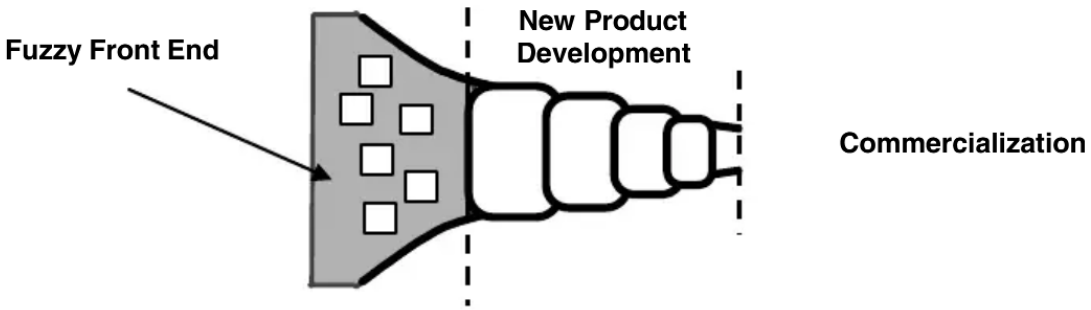


Figure 4.1: Innovation Process (Koen et al. 2001)

The Fuzzy Front End (FFE), also known as the front end of innovation, takes place at the beginning of the innovation process. It represents a stage of informal, unorganized and unstructured ideas preceding the New Product Development (NCD). Despite this, it constitutes one of the greatest opportunities to enhance innovation as a whole.
Next, the New Product Development (NPD) represents a stage where the structure, organization and scope mature, removing some of the chaos present in the previous stage. Furthermore, aspects like the commercialization dates, budgeting and revenue expectations can now be estimated or predicted with a higher level of certainty.

Last but not least, the Commercialization phase represents a culmination of all the work and effort applied in the previously mentioned stages. Here, the product is now ready to be commercialized as seen fit.

4.2 The NCD Model

The New Concept Development (NCD) model consists of a theoretical construct to address the lack of standardization and set best practices for the Fuzzy Front End (FFE) stage this model was created to provide further insight and common language Koen et al. 2001.

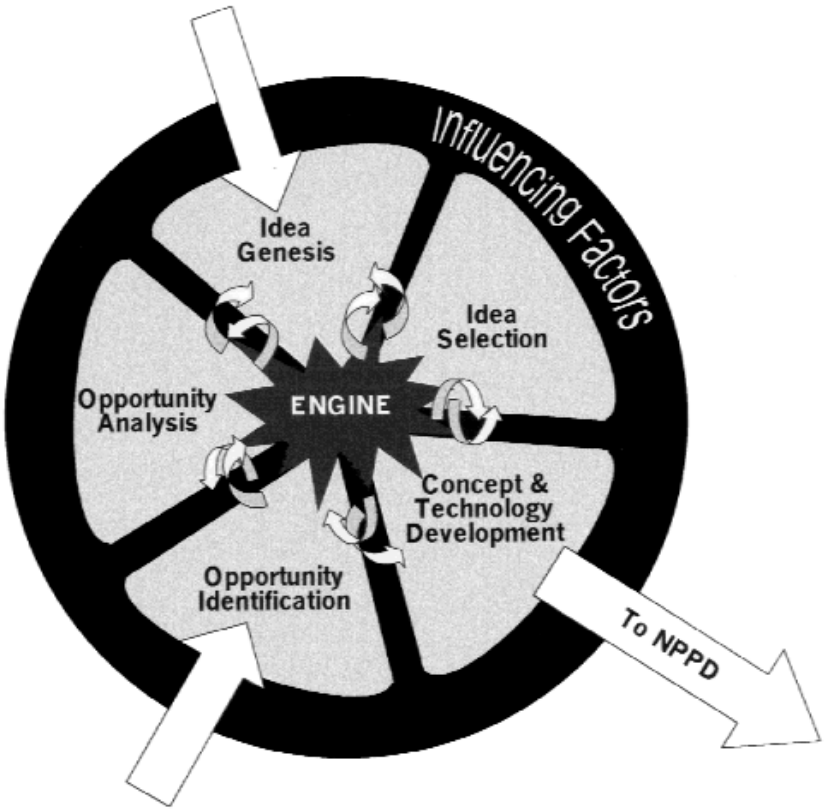


Figure 4.2: New Concept Development model (NCD) (Koen et al. 2001)

As depicted in figure 4.2, the NCD model consists of three main parts:

- Inner front-end elements (Wheel): This includes the following front-end elements - Idea Genesis, Idea Selection, Concept & Technology Development, Opportunity Identification and Opportunity Analysis
- Engine (Center): Fueled by the leadership and culture of the organization it will empower and drive the five front-end elements wheel.
- Influencing factors (Environment): It represents the exterior environment where aspects like business strategies, competitive factors and technological capabilities and maturity sit. The whole innovation process including the Fuzzy Front End (FFE) and New Product Development (NPD) is influenced by such an environment.

4.2.1 Opportunity Identification

Opportunity Identification is the first of the front-end elements to be specified, where the company or organization identifies the opportunities that can be followed through. This phase is typically aligned with the business goals. Technological advancements or breakthroughs are highly looked after, which may justify the reallocation of the organization's resources to better take advantage of these.
Specifically to the current document, the growth of GraphQL adoption as the API technologies chosen by developers and the growth/evolution of existing GraphQL architectures start exhibiting the drawbacks identified in 2.3.6, even by following best practices displayed on 2.3.5. Apollo Federation was identified as an opportunity to move past these drawbacks, but there are concerns related to other quality aspects such as performance due to its nobility.
To support GraphQL traction and increasing adoption since its inception, through Stack Overflow Insights (Overflow 2022a) it's possible to search for a specific tag trend over the years by attending tagged Stack Overflow questions. While searching for the specific tag "GraphQL", the evolution of its usage is easily observed, going from a 0.12% of Stack Overflow questions quota per month at the start of 2018 to 0.30% at the beginning of 2022. An extract of this trend chart can be observed in figure 4.3.

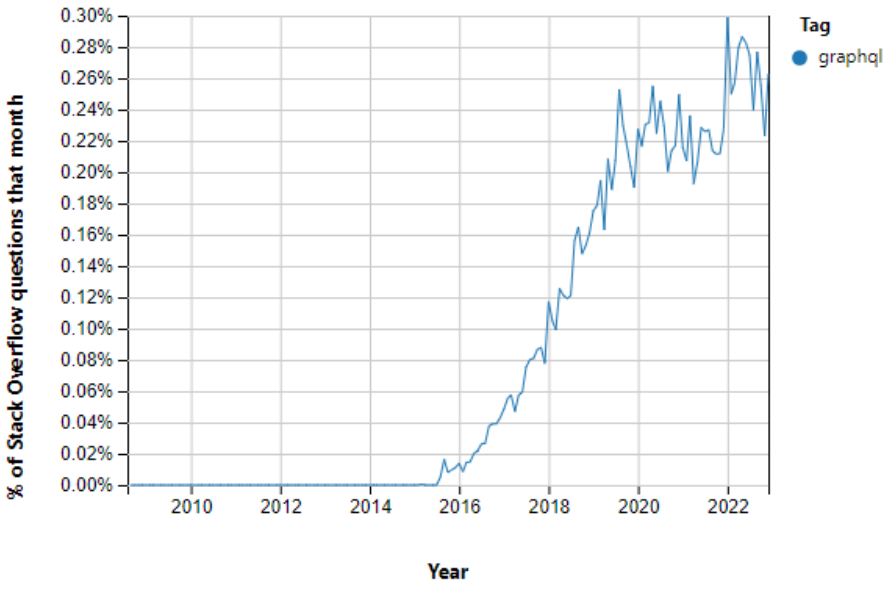


Figure 4.3: Stack Overflow Insights - Trends - GraphQL (Overflow 2022b)

4.2.2 Opportunity Analysis

Opportunity Analysis describes the stage where the opportunity identified in the previous stage is further studied and researched so that it can be converted into a business or technological opportunity. Multiple efforts can be developed by the organization to achieve such knowledge that will allow for further opportunity development, namely focus groups, market studies and scientific experiments.
In the specific case of this document, there is a need to understand how GraphQL adoption correlates to Apollo Federation growth.
The State of GraphQL 2022 community report (State of GraphQL 2022) aims to address questions such as "when do you need GraphQL exactly?" and any others that may come up

when attempting to adopt this technology within an organization's API stack. The survey reported that 47.9% of the participants use GraphQL for their API that are intended for website or app exposure. Furthermore, a range of client applications are listed, from which "Browsers" lead with 62.3% followed by "Native Mobile Apps" (24.2%), "Other Servers" (16.3%) and finally "Native Desktop Apps" (5.9%). Apollo Federation, also known as GraphQL federation, is also identified in the "Other Features" section, displaying that only 29.4% of the participants are not aware of this technology, while 46.9% are aware but do not actively use it and 23.9% use it currently, as displayed on figure 4.4. Finally, Apollo Server and Apollo Client have been identified as the only high-usage and high-retention libraries, which may ease and favour a transition to Apollo Federation and its architecture, as both are owned by the same organization which may mean cross-organization agreements and relations can be preserved and evolved. Additionally, a certain level of consistency or support can be expected between both technologies.

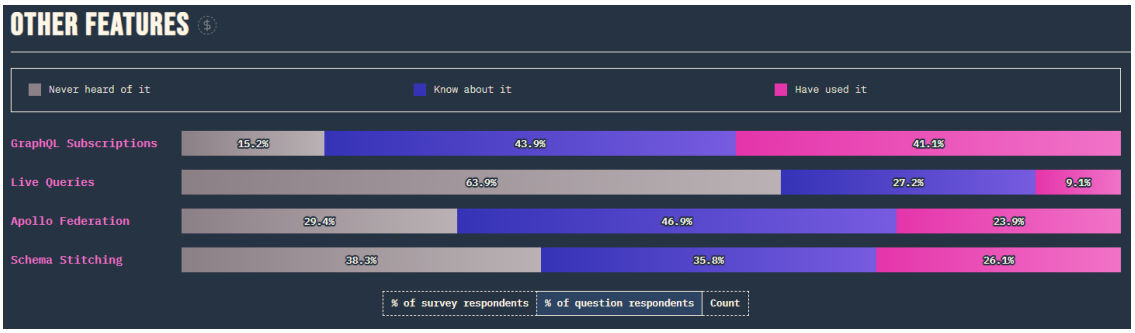


Figure 4.4: State of GraphQL 2022 - Other Features (State of GraphQL 2022)

This growth is further supported by Google Trends in which the growth of the "GraphQL" search term is evident. Google also identifies "Federation" as the top 6 related topics when this search term is utilized as depicted in figure 4.5 (Trends 2022).

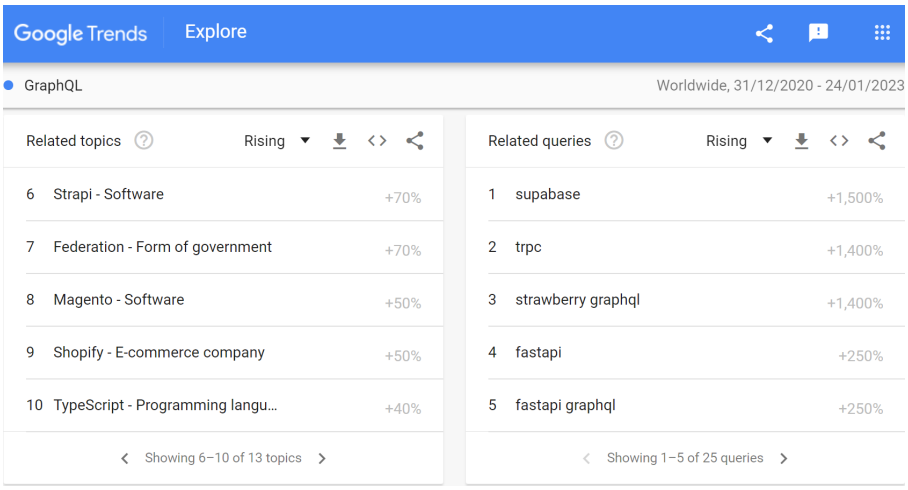


Figure 4.5: Google Trends - GraphQL (Trends 2022)

Furthermore, the multiple organization case studies documented in section 3.2 display that GraphQL Federation is indeed a reliable solution when addressing the use of GraphQL in more complex and mature architectures fit for a production environment. Batching, caching and

execution strategies documented in chapters 2.3.4, 2.3.4 and 2.3.4 respectively represent critical points that be utilized to improve the solution performance.

4.2.3 Idea Genesis

This present stage comprehends the development and maturation of the opportunity into a concrete idea or set of ideas. Each idea may be the result of a set of iterations over itself in which it was enriched, combined, torn apart and built upon until it satisfies all the involved stakeholders Koen et al. 2001.
In this particular thesis, the three main ideas identified attending to the objectives identified in the section 1.3 and research question in 1.3 are the following:

1. Use an existing GraphQL solution using well-known programming languages such as Java or JavaScript with multiple services and transition it to Apollo Federation for performance assessment.
2. Use an existing Apollo Federation solution with multiple services (sub-graphs) for performance assessment.
3. Develop a brand new solution using Apollo Federation, with multiple services (sub-graphs) for performance assessment.

4.2.4 Analytic Hierarchy Process (AHP)

As the idea generation was complete in the previous stage documented in 4.2.3, where three ideas prevailed, there is a need to effectively research, evaluate and select the idea which is perceived to have the most value. To achieve this, the Analytic Hierarchy Process was applied as it allows for both qualitative and quantitative criteria in the evaluation process. As the method name suggests, AHP relies on splitting the selection process into multiple hierarchical levels with the intent to ease its understanding and assessment (Nicola 2022).
As the first step of the AHP method, a hierarchy decision tree needs to be developed where the criteria to be applied to the previous set of ideas will be specified. The criteria selected for the current document exercise are the following:

- Time: The Time will be used as a measure of the development time required to fulfil the idea.
- Complexity: The Complexity criteria will qualify how complex the idea is to develop and deploy to allow for its use.
- Relevancy: The Relevancy criteria will evaluate how applicable the idea is to the performance assessment exercise.

The alternatives to be used and assessed throughout the process are the following:

1. Use an existing GraphQL solution using well-known programming languages such as Java or JavaScript with multiple services and transition it to Apollo Federation for performance assessment.
2. Use an existing Apollo Federation solution with multiple services (sub-graphs) for performance assessment.
3. Develop a brand new solution using Apollo Federation, with multiple services (sub-graphs) for performance assessment.

The resulting hierarchical decision tree is depicted in Figure 4.6:

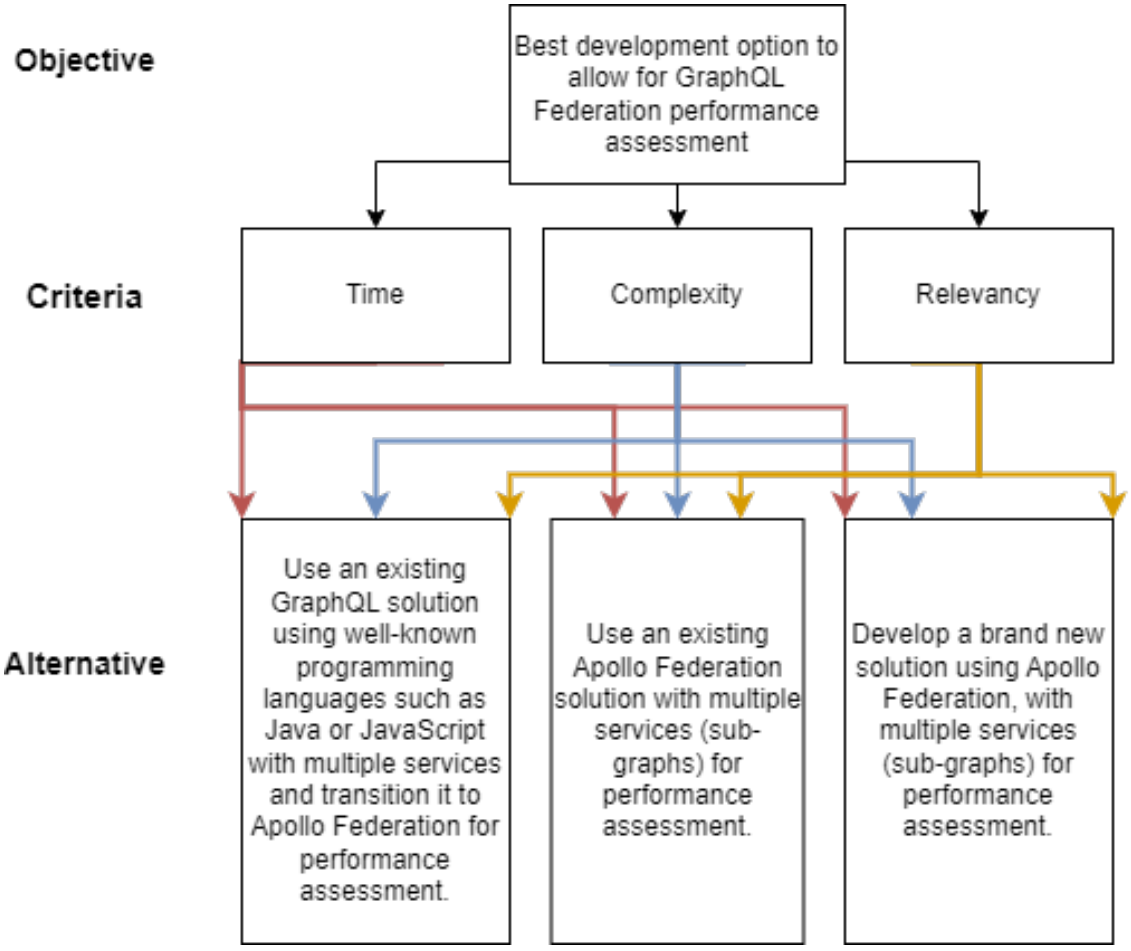


Figure 4.6: Hierarchical Decision Tree

Following the definition of a hierarchical decision tree, there is a need to determine which criteria prevail over each other. So this can be achieved, the matrix in Table 4.1 was developed, in which criteria are compared two by two to understand their relation and relevance to the idea selection process.
To keep matrix consistency the quantitative scale cannot exceed a total of nine levels (Nicola 2022). Therefore Saati's Nine Point scale was adopted to ensure the AHP method developed for this thesis follows some kind of standardization and best practices. This scale is depicted in Figure 4.7.

Intensity of importance	*Meaning*
1	Equal importance
3	Weak importance of one over another
5	Essential important
7	Demonstrated importance
9	Absolute importance
2, 4, 6, 8	Intermediate values between the two adjacent judgement

Figure 4.7: Saati's Nine Point Scale (Saaty 1990)

Based upon this scale, the Criterion Comparison Matrix was developed (Table 4.1). This artefact can be read from the left to the right, where a value greater than 1 means a greater importance relationship, and, implicitly, a value lesser than 1 means a less important relationship. For example, the criteria "Time" was deemed more important than "Complexity" but less when compared to "Relevancy". Each of the values was developed taking into account the thesis priority or constraints, for example, time is an essential unit and highly constrained by the thesis deadlines, but the relevancy of the given alternative should be considered critical for this exercise.

Table 4.1: Criterion Comparison Matrix

	Time	**Complexity**	**Relevancy**
Time	1.00	3.00	0.50
Complexity	0.33	1.00	0.20
Relevancy	2.00	5.00	1.00

After the criterion comparison matrix is developed, some level of normalization is required so that important values are adjusted to the same scale/unit. The normalization is achieved by dividing individual importance (e.g. Time to Time comparison = 1.00) by the sum of all the values within the column it belongs to (e.g Time column = 3.33).
With the comparison matrix now normalized, it's possible to go further by determining the relative priority vector which allows the perception of criterion importance order. Each relative priority is calculated by computing the mean of each row's normalized values.
The normalized criterion comparison matrix and the relative priority vector are depicted in Table 4.2.

Table 4.2: Normalized Criterion Comparison Matrix

	Time	**Complexity**	**Relevancy**	**Relative Priority**
Time	0.30	0.33	0.29	0.3092
Complexity	0.10	0.11	0.12	0.1096
Relevancy	0.60	0.56	0.59	0.5813

From a quick look at the relative priority vector, for this specific AHP exercise, the criterion "Relevancy" is the most important (0.5813), followed by "Time" (0.3092) and finally the criterion "Complexity" (0.1096).
This is aligned with the document's purpose, as the idea to be followed need to be as relevant as possible to the performance assessment exercise. Despite this, time is a critical driver, as it is in all projects or developments, due to constraints to delivery dates and to allow for the completion of other activities. Last but not least, Complexity always needs to be considered as it has an effect on how efficiently the author can develop the required idea and can impact the desired outcome as a whole.

Consistency Ratio

The next step of the AHP method comprehends the calculation of the Consistency Ratio (CR) which allows for the determination of how consistent the previously calculated values are. To determine if the values are consistent the Consistency Ratio (CR) must assume a value lesser than 0.1 (Nicola 2022).
Firstly, λ_{max} needs to be determined by using the following formula (Nicola 2022):

$$Ax = \lambda_{max} x \tag{4.1}$$

Here "A" refers to the criterion comparison matrix depicted in Table 4.1 and "x" the relative priority vector depicted as the last column in Table 4.2. With this context into account, the value for λ_{max} was calculated to be 3.0076 which is the mean of the values observed in the following Table 4.3.

Table 4.3: Consistency Comparison Matrix

Criterion	Value
Time	3.00
Complexity	2.82
Relevancy	3.15

Once computed λ_{max} value, it's possible to calculate the value for the Consistency Index (CI) through the following equation (Nicola 2022):

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{4.2}$$

Here "n" represents the number of criteria/criterion used throughout the AHP method. Assuming the number of criteria (n = 3) and the previously calculated value of λ_{max} (3.0076), a Consistency Index of 0.0038 was achieved.
Finally, to calculate the Consistency Ratio (CR) and whether the AHP process is reliable the following equation can be applied (Nicola 2022):

$$CR = \frac{CI}{RI} \tag{4.3}$$

Here "CI" represents the Consistency Index previously calculated through the equation 4.2 and "RI" the Random Index. The Random Index can be extracted from Figure 4.8, which represents the Random Indexes for n-order square matrices. In this specific case, as n is 3, 0.58 will be selected as the Random Index for CR calculation.

Matrix sizes (n)	1	2	3	4	5
RI Value	0	0	0.58	0.9	1.12

Figure 4.8: Random Indexes for n order square matrices (Jesus França et al. 2020)

As the value of Consistency Ratio (CR) was computed to be 0.0065, and, therefore lesser than 0.1, we can verify the consistency of the values used so far on the AHP method.

Alternative Comparison

After the consistency is assured by the calculations done on 4.2.4, the method can proceed to the development of each criteria comparison matrix with the alternatives defined on 4.2.4. The comparison matrices can be observed in Tables 4.4, 4.5 and 4.6 for Time, Complexity and Relevancy respectively. Each of the values defined corresponds to the estimated time, complexity and relevancy from the investigation and research work carried out in Chapter 2 and Chapter 3.

Table 4.4: Comparison Matrix Time - Alternatives

Time	A1	A2	A3
A1	1.00	0.59	3.00
A2	2.00	1.00	7.00
A3	0.33	0.14	1.00

Table 4.5: Comparison Matrix Complexity - Alternatives

Complexity	A1	A2	A3
A1	1.00	2.00	0.50
A2	0.50	1.00	0.25
A3	2.00	4.00	1.00

Table 4.6: Comparison Matrix Relevancy - Alternatives

Relevancy	A1	A2	A3
A1	1.00	0.50	3.00
A2	2.00	1.00	5.00
A3	0.33	0.20	1.00

Applying the same transformation as on Table 4.2 and determining the relative priority vector for each, the normalized Tables 4.7, 4.8 and 4.9 in the same order as before are achieved.

Table 4.7: Normalized Comparison Matrix Time - Alternatives

Time	A1	A2	A3	Alternative Local Priority
A1	0.30	0.30	0.27	0.29
A2	0.60	0.61	0.64	0.62
A3	0.10	0.09	0.09	0.09

Table 4.8: Normalized Comparison Matrix Complexity - Alternatives

Complexity	A1	A2	A3	Alternative Local Priority
A1	0.29	0.29	0.29	0.29
A2	0.14	0.14	0.14	0.14
A3	0.57	0.57	0.57	0.57

Table 4.9: Normalized Comparison Matrix Relevancy - Alternatives

Relevancy	A1	A2	A3	Alternative Local Priority
A1	0.30	0.29	0.33	0.31
A2	0.60	0.59	0.56	0.58
A3	0.10	0.12	0.11	0.11

Now that is possible to extract each criteria's local alternative priority, a new matrix multiplication is required to determine the composite priority. The multiple local alternatives will

be joined into a single matrix and multiplied by each criteria weight as calculated on 4.2. The result of such transformation is depicted in the column "Composite Priority" of Table 4.10.

Table 4.10: Composite Priority Computation from Local Alternative Priority

	Time	**Complexity**	**Relevancy**	**Composite Priority**
A1	0.29	0.29	0.31	0.30
A2	0.62	0.14	0.56	0.54
A3	0.09	0.57	0.11	0.15

From the analysis of each alternative composite priority, we can extrapolate the order of priority that is preferential, attending to the previously defined AHP objective. In this particular case, the second alternative seems to be optimal for the purpose, leading to a composite priority of 0.54. Secondly, the first alternative is the second most valuable alternative with a composite priority of 0.30.

4.2.5 Idea Selection

Even though a set of valuable ideas are output from the Idea Genesis stage, businesses must decide on which one to pursue, to achieve the most value possible. This process should be less rigorous than the New Product Development (NPD) stage to allow ideas to grow and advance, even if their value is still uncertain (Koen et al. 2001).
From the several ideas generated in "Idea Genesis", one has to be selected as the chosen or optimal idea to develop further. Attending to driving factors like time to develop, complexity and relevancy to the performance assessment exercise, the second idea was chosen. This choice is further reinforced by the Analytic Hierarchy Process (AHP) method outputs in section 4.2.4.
This alternative allows for less time to be dedicated to the development of the solution, complexity to be restricted to specific implementation details which may demand extra research and the scenarios required for the performance assessment exercise can be built upon the existing architecture. Despite this, if there is no applicable existing project/solution, there may be a need to pivot to searching for a GraphQL solution and transition whatever is necessary to allow for the performance assessment exercise, as documented on the first AHP alternative.

Chapter 5

Analysis and Design

After the multiple ideas were generated in Section 4.2.3, and after its further analysis by using the AHP technique, the idea to use an existing Apollo/GraphQL Federation solution was selected.
Following this selection, the current chapter aims to document the analysis required to find the optimal solution from a selection of possible candidates by determining advantages and pain points for each.
Finally, the design and all other planning points will be documented to provide an early look into the development of the extra functional and non-functional requirements for the performance evaluation. This will be extremely important to introduce/launch the actual development work which will be further documented in the implementation chapter.

5.1 Base solution

As described in Section 4.2.5, an existing GraphQL Federation solution needs to be selected to, after the required development, be utilized as the source of information for Chapter 7.
To search and research for such solutions, GitHub (Github 2023) was employed as it is widespread across developers and provides a huge amount of codebases that can be easily queried by tags (GitHub 2023). The following main tags/aspects were selected to lead the search:

- GraphQL Federation: As the desired base solutions should use GraphQL Federation, it's important to use this tag which would not only include GraphQL projects but also such projects which are using Apollo Federation spec and technology to solve its graph management.
- Java: Due to the author's experience with this programming language, it would be ideal if the underlying GraphQL services used in the federated solution used Java or GraphQL-Java (Java 2023).
- Verified: The selected solution should be from a trustworthy and quality source. This may be analysed by the number of forks, the number of bugs, update frequency and other factors that can help determine if the GitHub project is a verified and functional starting point.
- Documentation: Although this can be an extra point for the project verified status, documentation would be an extra plus when choosing the base project as it would drive a more efficient and accurate analysis.

Taking into account the mentioned aspects, two GitHub repositories stand out: Netflix's DGS example (Palacio 2022) and Apollo's Federation JVM Spring example (Kuc 2023a). These two solutions will be further analysed in sub-sections 5.1.1 and 5.1.2 flagging the advantages and pain points for each.

5.1.1 Solution A: Netflix's DGS Example

This GitHub repository represents the Netflix example for the use of the DGS (Domain Graph Service) framework for GraphQL Federation solutions. The DGS (Domain Graph Service) framework is an open-source framework used to support federated GraphQL services/API generation (Netflix 2023). This is achieved by the use of code annotations which will drive the schema generation (code-first approach). Finally, the framework also offers powerful features related to caching, error handling and data loaders (Netflix 2023).
Furthermore, the DGS framework has comprehensive and up-to-date documentation of its capabilities and features which eases the learning curve when adopting this framework as a new user (Netflix 2023).

Current Architecture

Currently, this example contains a GraphQL query for shows and another for reviews, but no mutations or subscriptions can be found. A representation of the currently available software components and their relations is depicted in Figure 5.1.

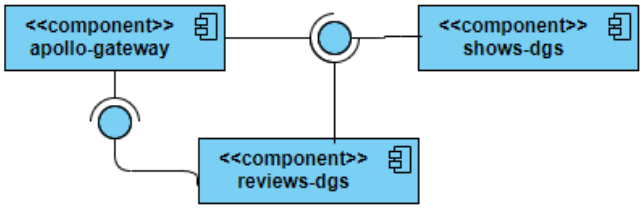


Figure 5.1: DGS Example - Component Diagram

Apollo Federation Compatibility

GraphQL Federation compatibility is a critical point for the base solution. As per DGS framework support, Apollo's Federation-Compatible subgraph libraries list (Apollo GraphQL 2023c) flags no issues and describes DGS (Domain Graph Service) framework as fully capable for Federation 1 and 2, as observed in Figure 5.2.

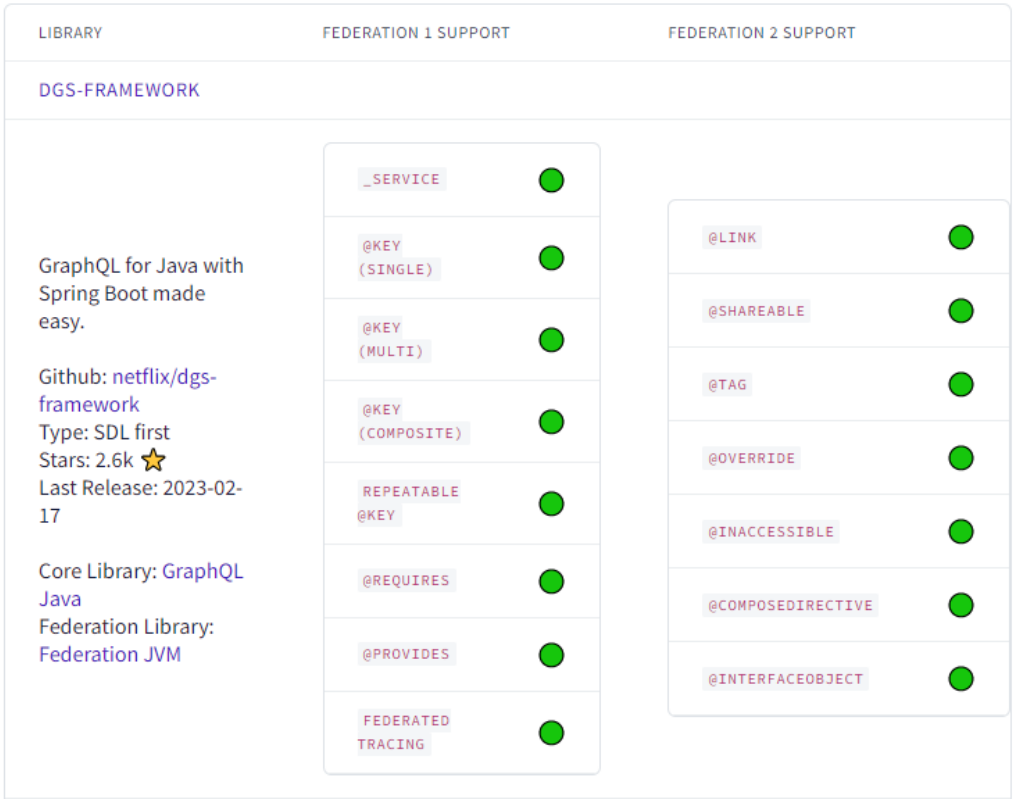


Figure 5.2: DGS Framework - Apollo Federation Compatibility (Apollo GraphQL 2023c)

5.1.2 Solution B: Apollo's Federation JVM Spring Example

This GitHub repository represents an example implementation of GraphQL Federation for JVM (Java Virtual Machine) using Spring (Kuc 2023a).
Federation JVM is built on top of GraphQL-Java but with new logic that makes it possible for the different sub-graphs to be Federation compatible. Additionally, some logic is specifically dedicated to providing access to common Federation types such as Scalar and Entities. Finally, some Federation-aware instrumentations are included such as cache control and tracing capabilities (Kuc 2023b).
Although Federation JVM is not a fully-fledged framework like DGS, the authors still provide some level of documentation and example projects to allow for its adoption for new projects (Kuc 2023a,b).

Current Architecture

This base solution only supports a product query, which also retrieves its list of reviews. No operations of mutation or subscription type are present currently but its need should be revised in a further design iteration if this is chosen as a base solution. A component diagram representing the currently available components and their relations is depicted in Figure 5.3.

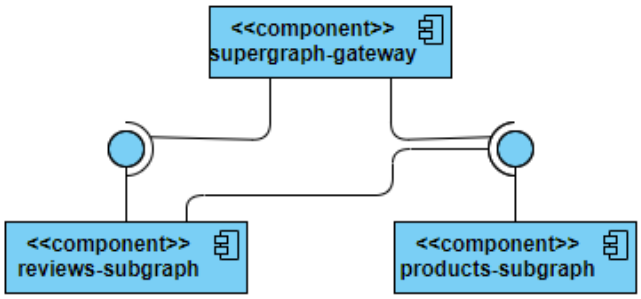


Figure 5.3: Federation JVM Example - Component Diagram

Apollo Federation Compatibility

As previously applied for Netflix's DGS example, the Federation JVM library needs to be analysed in regard to compatibility. Just like on Netflix's example, Apollo's Federation-Compatible subgraph libraries list (Apollo GraphQL 2023c) flags no issues and describes the Federation JVM library as fully capable for Federation 1 and 2, as observed in Figure 5.4.

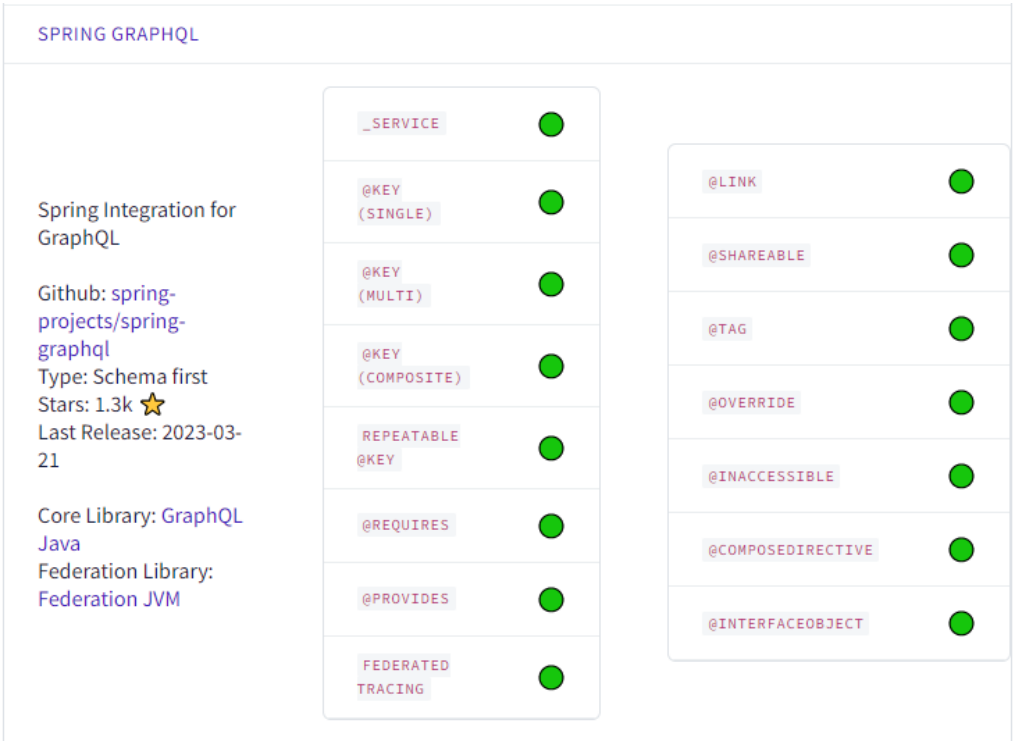


Figure 5.4: Federation JVM - Apollo Federation Compatibility (Apollo GraphQL 2023c)

5.1.3 Selected Solution

Now that both of the base solutions were investigated and analysed, a choice needs to be made regarding which is the most adequate for the thesis. A set of characteristics played into the decision-making process:

- Complexity: Regarding complexity, both base solutions are quite simple and do not contain too much logic regarding their domain. DGS is a fully-fledged framework and thus carries additional complexity when it comes to using its annotations. Although this would help with the code generation and other aspects of the development process, for the exercise of the performance assessment it is only expected that the performance test points (caching, batching and executions strategies) are developed, which would not take full advantage of DGS.
- Relevancy: Regarding relevancy, at first glance both solutions seem perfectly adequate, but after exploring DGS some concerns have been raised. The main of which is how much impact would the extra annotations and logic that runs on the background of the framework would impact the performance of the overall solution, thus tainting the performance assessment. For this reason, Federation JVM seems to be optimal regarding relevancy.
- Experience: Regarding experience, both base solutions are at a standstill. The author's experience with both solutions is tiny or close to none.
- Federation Compatibility: Both solutions are perfectly Federation compatible as it was perceived in Sections 5.1.1 and 5.1.2.
- Documentation: Even though Netflix's DGS framework has more extensive documentation regarding its capability and functionality, what is provided by Federation JVM is sufficient to allow developers to develop GraphQL Federation solutions utilizing Spring.

As perceived from each of the characteristics of the decision-making process, the Federation JVM example is more adequate for this thesis's objectives. The deal breaker was exactly on the relevancy characteristic where the unknown complexities or aspects of the DGS framework background processes or logic could taint the performance assessment exercise which is of critical importance for the thesis.

5.2 Design

This section will aim to describe all the planning and design done before the extension of the base solutions into the application that is going to be used for the performance assessment (Chapter 7).
The functional and non-functional requirements will be organized and documented to achieve the desired functionality level required for the performance assessment activities. Special focus will be aimed at the performance non-functional requirement, which is critical to the thesis.
Finally, the evolution between the base and final architecture will be presented allowing for a bird's eye view of what is going to be achieved.

5.2.1 Requirements

With the performance assessment exercise in mind, a set of functional requirements were identified which will extend the current base application functionality and capability, described in Table 5.1. These have been described taking into account the standard user story template which splits the definition into role, activity and business value (Zeaaraoui et al. 2013).

Table 5.1: Functional Requirements

ID	Description
UC1	As a Product Manager I want to be able to query all existing Products
UC2	As a Product Manager I want to be able to create a new Product
UC3	As a Product Manager I want to be able to search for a specific Product
UC4	As a System Administrator I want to be able to query all existing Users
UC5	As a System Administrator I want to be able to create a new User
UC6	As a System Administrator I want to be able to search for a specific User
UC7	As a Product Manager I want to be able to query all existing Reviews
UC8	As a User I want to be able to create a new Review
UC9	As a Product Manager I want to be able to search for a specific Review

Furthermore, the use case diagram present in Image 5.5 was developed to better illustrate all the use cases and users. This particular system view represents the "+1" from Philippe Kruchten's 4+1 View Model of Software Architecture, otherwise designated as the scenario view (Kruchten 1995).

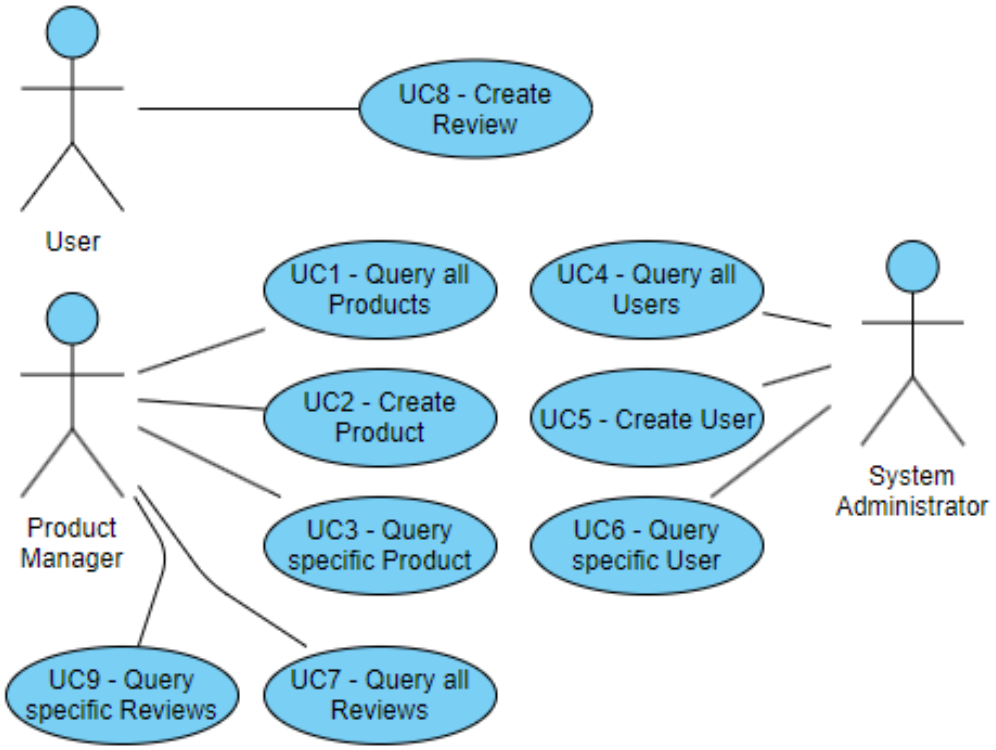


Figure 5.5: Use Case Diagram

The non-functional requirements were identified and defined through the ISO 25010 specification framework. As it was already introduced in Chapter 3, it is split into the following characteristics:

- Functional Suitability
- Performance Efficiency
- Compatibility

- Usability
- Reliability
- Security
- Maintainability
- Portability

Maintainability

Maintainability describes the level of effectiveness and efficiency at which a product can be modified, improved or corrected ISO 25000 2011.
In this specific case, it's important to allow for the solution to be used and evolved by others with the intent to allow for its evolution and give way for other performance assessment activities not necessarily conducted or in the scope of the current thesis.
Furthermore, the sub-characteristic of modularity will be critical to allow for changes to be made across the different subgraphs/GraphQL services without compromising the whole solution.

Reliability

Reliability describes the degree to which the product, solution or piece of software performs its functions under specific contexts and conditions (ISO 25000 2011).
For the performance assessment to be carried out correctly, it's important for the solution to behave nominally in any execution conditions required of the performance tests. These conditions may describe a high volume request rate to identify any performance divergences between caching, batching and execution strategies.

5.2.2 Constraints

Since the requirements are defined against the final solution, there is a need to also specify any constraints that affected its definition, development and execution.
As already mentioned and emphasised in Section 4.2.4 of Chapter 4, Time is of significance for the current thesis. There is an objective amount of availability by the author and due dates that need to be complied with, which drove some of the choices made regarding the initial solution to be extended upon as documented in Sections 4.2.5 and 5.1.3.
Another implicit constraint is the technologies that need to be used for the solution that will be used for the performance assessment. GraphQL and GraphQL Federation are mandatory which in turn limit the number of programming languages that actually have such implementations. Furthermore, selected technologies should not imply learning curves which could hinder the time which is available for development and the thesis as a whole.

5.2.3 Solution Architecture

As the base solution architecture can already be perceived as part of Section 5.1.2, throughout this section the final application architecture will be documented as what should result from the development work to be carried out.

Final Solution

When comparing to the base solutions, it's possible to understand that now a new GraphQL service has been added - Users. Furthermore, some changes will also be inherited in other

existing services regarding the requirements documented in Section 5.2.1, despite the fact they cannot be perceived in Image 5.6.

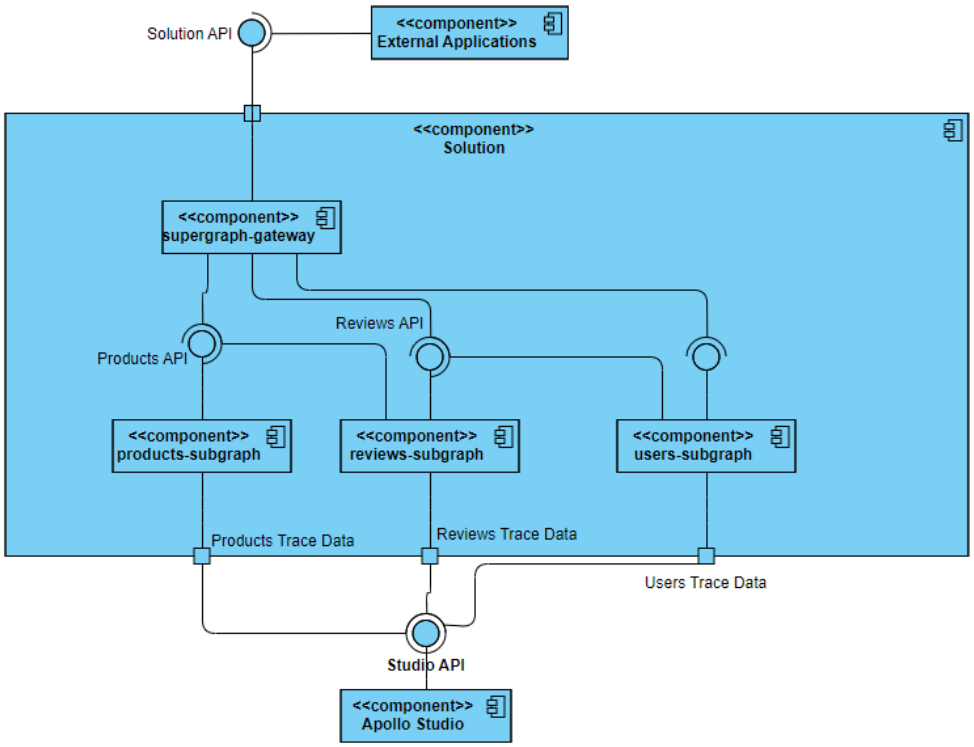


Figure 5.6: Solution Architecture - Logical View

To complement the logical view, the 4+1 View Mocal view is depicted in Image 5.7 (Kruchten 1995). In this specific case, the supergraph and each of the sub-graphs will be deployed in different containers as it can be perceived. More details regarding this Containerization can be found in Chapter 6.

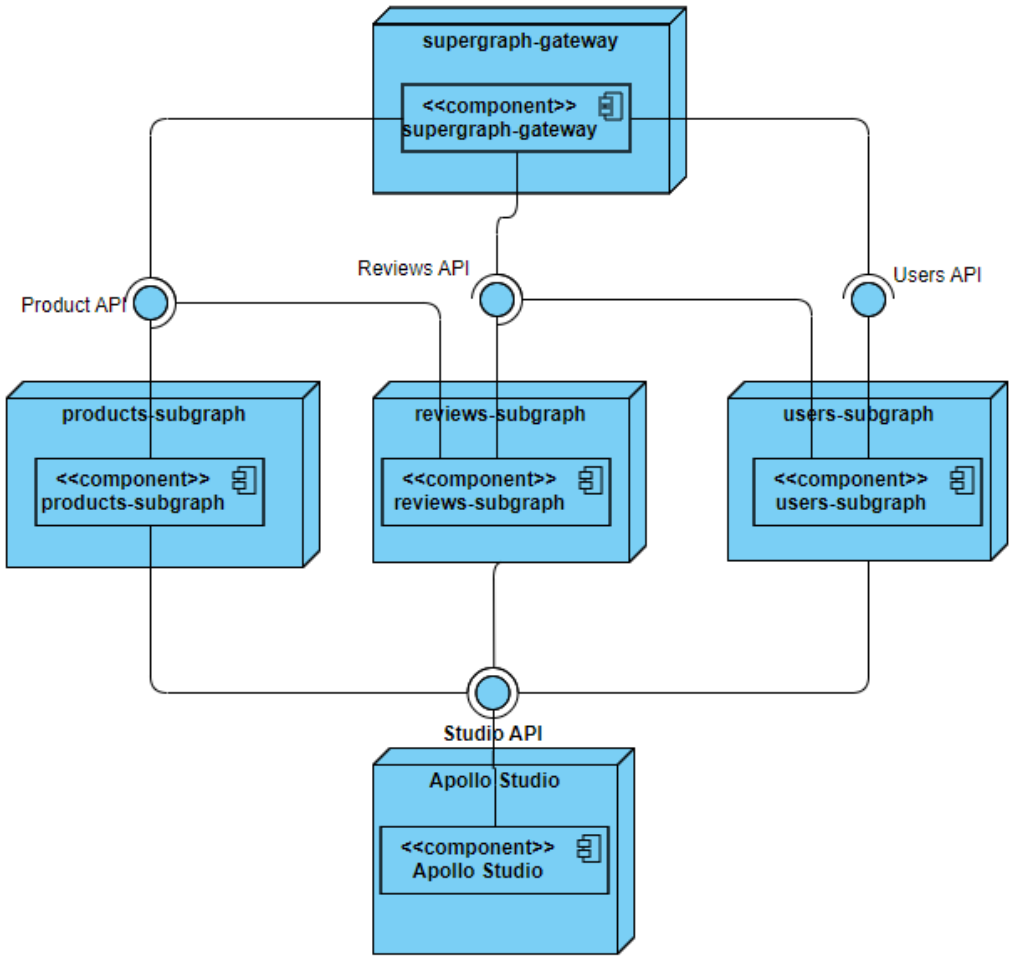


Figure 5.7: Solution Architecture - Physical View

Chapter 6

Implementation

This chapter mainly focuses on detailing the artefacts, functionality and any other documentation resulting from the implementation process. This chapter follows the design, blueprint or methodology described in Chapter 5. Firstly, the base solution is further documented regarding how it can be utilized and the current GraphQL schema and data management capabilities. Following this description, all the required developments are recorded and explained, leading up to a performance assessment-ready application.

6.1 Base Solution

This section aims to further detail the base solutions components, schema and database capabilities.
Firstly, it's critical to understand the base application, and how it works so the developments are easily identifiable and understandable. As detailed in Chapter 5, the repository in which the application is maintained already has a meaningful amount of documentation. Table 6.1 depicts each of the base solution components and how to run them.

Table 6.1: Base Solution - Run Components

Component	**API**	**Technology**	**Command**
products-subgraph	GraphQL	Java Spring Boot	gradle bootRun
reviews-subgraph	GraphQL	Java Spring Boot	gradle bootRun

Even though the above components are now running, after correctly executing the above commands and ensuring build integrity, the supergraph router will still not be available. To achieve this Rover was used to start the federation router (Apollo GraphQL 2023d). This is further detailed in Table 6.2.

Table 6.2: Base Solution - Run Federated Router

Component	**Command**
products-subgraph	rover dev –name products –schema <Schema Location> –url <URL>
reviews-subgraph	rover dev –name reviews –schema <Schema Location> –url <URL>

This will ensure both subgraphs are now added to the Federated Architecture and are accessible from the GraphQL Federation supergraph whose properties are detailed in the router YAML file displayed on 6.1.

```yaml
cors:
  # allow_credentials: true
  allow_any_origin: true

health_check:
  listen: 0.0.0.0:8088
  enabled: true

homepage:
  enabled: false

sandbox:
  enabled: true

supergraph:
  listen: 0.0.0.0:3000
  introspection: true
  tracing-enabled: true

include_subgraph_errors:
  all: true

#plugins:
#  experimental.expose_query_plan: true
```

Listing 6.1: Router (YAML).

Intending to solidify the understanding of the already documented architecture and components, Figure 6.1 details a UML Sequence Diagram with the different agents (both humans and software components), which are involved in the GraphQL query to obtain all recorded products.

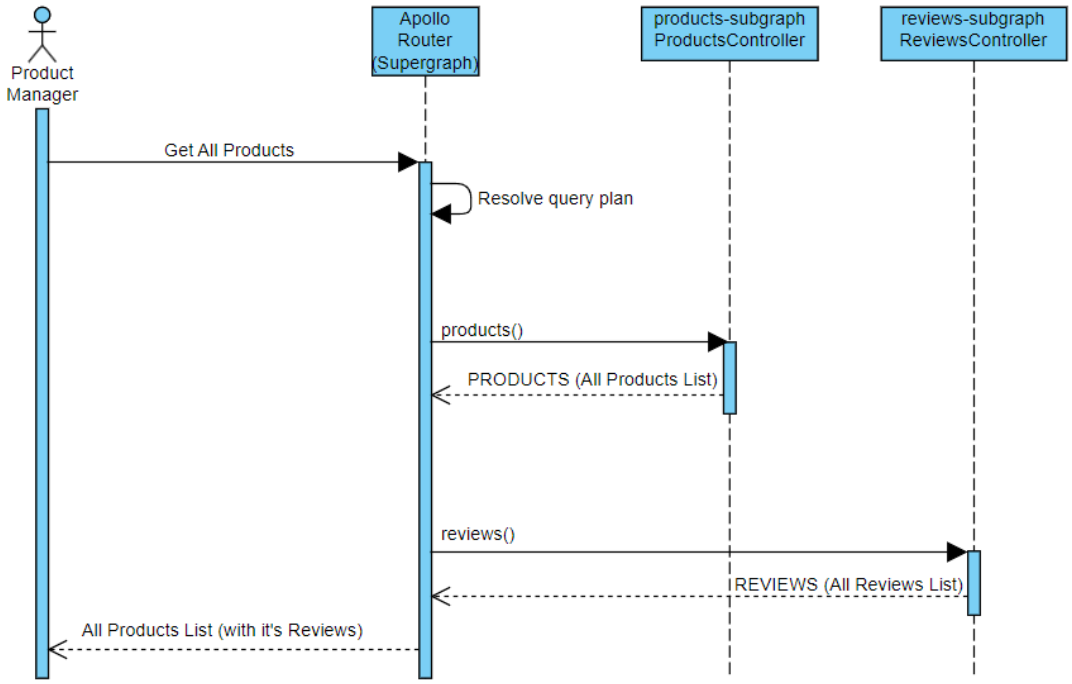


Figure 6.1: Sequence Diagram - All Products Query

As perceived, the solution entry point will be the supergraph which will resolve the received query and develop a query plan taking into account the current Apollo Schema Registry. After the Query Plan is developed, the requests for each of the individual subgraphs are triggered. In the wake of the response of each subgraph being received, the subgraph will aggregate all information taking into account the registered schemas in the Apollo Schema Registry (Apollo GraphQL 2022d).
The Query Plan developed by the supergraph from the current Apollo Schema Registry can be easily accessed through the Apollo Router front-end. On the Query or other GraphQL Operation section, it's possible to display the Query Plan. Figure 6.2 depicts the query plan which resulted from an All products query (used in Figure 6.1).

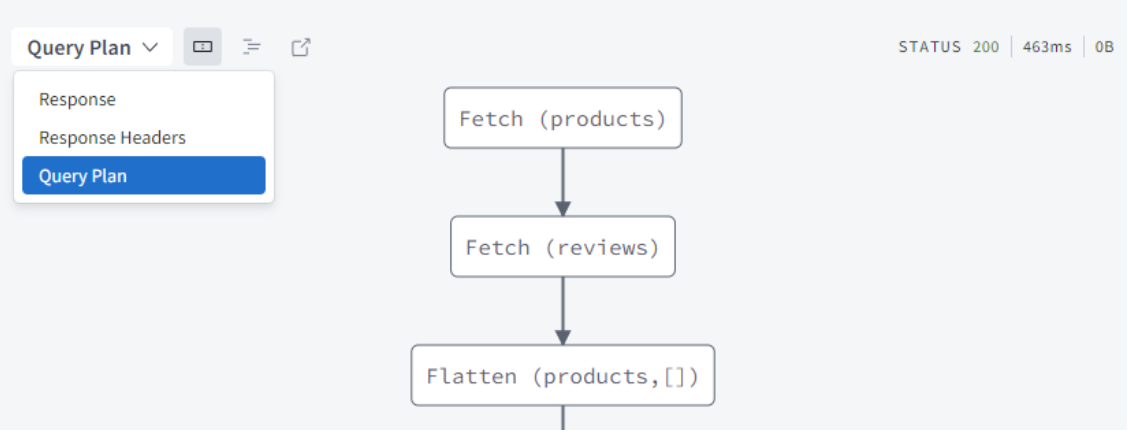


Figure 6.2: Apollo Router Query Plan - All Products Query

6.1.1 Schema

As part of the base solution, both product and review schemas only contain simple GraphQL Queries and the Product and Review types respectively. These will be joined when registered on the supergraph, resulting in a brand new supergraph schema (Apollo GraphQL 2023b).
If the supergraph router introspection is enabled as it is in the base solution (Listing 6.1), the Apollo Router front-end will provide visibility of the supergraph schema that is currently in use. The available supergraph schema as per the base solution state is depicted in Listing 6.2.

```
directive @defer(label: String, if: Boolean! = true) on FRAGMENT_SPREAD
    | INLINE_FRAGMENT

type Product {
  id: ID!
  name: String!
  description: String
  reviews: [Review!]!
}

type Query {
  product(id: ID!): Product
  products: [Product!]!
  review(id: ID!): Review
  reviews: [Review!]!
}

type Review {
  id: ID!
  text: String
  starRating: Int!
}
```

Listing 6.2: Supergraph Schema (Introspection).

6.1.2 Database

The base solution unfortunately does not support any database capabilities and the data is maintained in static, hard-coded lists. Furthermore, there is not a clear separation between the Controller and the Repository (Data) layer within the source code. An example of such hard-coded lists can be observed in Listing 6.3.

```
@Controller
public class ProductsController {

...

  private final Map<String, Product> PRODUCTS = Stream.of(
    new Product("1","Saturn V", "The Original Super Heavy-Lift Rocket!")
    ,
    new Product("2","Lunar Module"),
    new Product("3","Space Shuttle"),
    new Product("4","Falcon 9", "Reusable Medium-Lift Rocket"),
    new Product("5","Dragon", "Reusable Medium-Lift Rocket"),
    new Product("6","Starship", "Super Heavy-Lift Reusable Launch
    Vehicle")
  ).collect(Collectors.toMap(Product::id, product -> product));


...

}
```

Listing 6.3: Database Solution - Base Application.

Even though this is not impactful in the base solution scope and requirements, as mutations will be needed to support the functional requirements detailed in the design phase (Chapter

5) this implies the development of a more dynamic database solution which allows for both read and write operations.

6.2 Performance Assessment Solution Development

This section focuses on the documentation in detail of the required development to attend to the functional and non-functional requirements specified in Chapter 5. Furthermore, some examples are provided of how the implementation of some of the execution, caching and batching strategies was achieved.
Similar to Section 6.1, an overview of the final solution component is depicted in Table 6.3. The most perceivable change is the addition of the new users-subgraph which manages the users that review products and their details.

Table 6.3: Final Solution - Run Components

Component	API	Technology	Command
products-subgraph	GraphQL	Java Spring Boot	gradle bootRun
reviews-subgraph	GraphQL	Java Spring Boot	gradle bootRun
users-subgraph	GraphQL	Java Spring Boot	gradle bootRun

The developed user schema is depicted in Listing 6.4. The most notable aspect of the user schema is the extension of the Review type (defined on the review subgraph) through the use of the **Extends** directive thus adding a User to the Review schema type. Furthermore, it's also possible to observe the information that is recorded for each of the solution users, which in this case are their identifiers, names and ages.

```
type Query {
    user(id: ID!): User
    users: [User!]!
}

type Mutation{
    addUser(reviewId: String!,id: String!, age: Int!, name: String!) :
    User
}

type Review @key(fields: "id") @extends {
    id: ID! @external
    user: User
}

type User @key(fields: "id") {
    id: ID!
    name: String!
    age: Int
}
```

Listing 6.4: User Schema.

The same process as in Table 6.2 is used for the users-subgraph to make sure the schema is added to the Apollo Schema Registry and utilized by the solution supergraph router. Following the same process as explained in Section 6.1.1, this results in the final solution supergraph schema depicted in Listing 6.5.

```
directive @defer(label: String, if: Boolean! = true) on FRAGMENT_SPREAD
    | INLINE_FRAGMENT

enum CacheControlScope {
  PUBLIC
  PRIVATE
}

type Mutation {
  addProduct(id: String!, name: String!, text: String!): Product
  addReview(productId: String!, id: String!, starRating: Int!, text:
    String!): Review
  addUser(reviewId: String!, id: String!, age: Int!, name: String!):
    User
}

type Product {
  id: ID!
  name: String!
  description: String
  reviews: [Review!]!
}

type Query {
  product(id: ID!): Product
  products: [Product!]!
  review(id: ID!): Review
  reviews: [Review!]!
  user(id: ID!): User
  users: [User!]!
}

type Review {
  id: ID!
  text: String
  starRating: Int!
  user: User
}

type User {
  id: ID!
  name: String!
  age: Int
}
```

Listing 6.5: Supergraph Schema (Introspection).

Although it's not noticeable from such a high-level overview of the solution, patterns such as SRP (Single Responsibility Pattern) and Repository were adopted and implemented by splitting the Data Management and API Controller responsibilities with the creation of a new Repository layer in each and every subgraph. Furthermore, with the objective of increasing API responsiveness and enabling serial execution strategy tests, controller-level methods were converted to an asynchronous format. An example of such modification is depicted in Listing 6.6 for the GraphQL Queries pertaining to getting all reviews, and a review by its identifier.

```java
package com.example.reviews;

@AllArgsConstructor
@Controller
public class ReviewsController {


  @Autowired
  private final ReviewRepository repo;

  @QueryMapping
  public CompletableFuture<List<Review>> reviews() {
    return CompletableFuture.supplyAsync(() -> {
      return repo.findAll();
    });
  }

  @QueryMapping
  public CompletableFuture<Review> review(@Argument String id) {
    return CompletableFuture.supplyAsync(() -> {
      return repo.findReviewById(id);
    });
  }

}
```

Listing 6.6: Reviews Controller - Asynchronous Methods.

6.2.1 Functional

Succeeding the overview of the changes at a higher level, this sub-section aims to explain and details the required developments which are directly related to the functional requirements identified in the design phase (Chapter 5).
As the changes to each subgraph are similar in nature, and in an effort to reduce the extension of this sub-section, detailed examples will only be provided for the products-subgraph. All GraphQL mutations and queries can already be observed in Listing 6.5 within the respective type declaration.

Queries

With respect to the use cases documented in Table 5.1 which represent pure read operations, the base GraphQL queries for products and reviews were updated, and the user's GraphQL queries were developed.
Firstly, the schema files were updated to include new query declarations under the Query type declaration. An extract of the products-subgraph schema is detailed in Listing 6.7.

```
type Query {
    product(id: ID!): Product
    products: [Product!]!
}

type Mutation{
    addProduct(id: String!, name: String!, text: String!) : Product
}

type Product @key(fields: "id") {
    id: ID!
    name: String!
    description: String
}
```

Listing 6.7: Product Schema.

Now that the schema definition for the new queries is complete, new controller methods are required to capture the logic and behaviour to be followed to obtain such information. Through the use of the **QueryMapping** annotation, Spring quickly resolves each schema GraphQL Query to the respective controller method. Furthermore, the **Argument** annotation can be used to bind each of the specified schema query arguments at the controller level. Such developments example can be perceived in Listing 6.8.

```
package com.example.products;


@AllArgsConstructor
@Controller
public class ProductsController {


  @Autowired
  private final ProductRepository repo;


  @QueryMapping
  public CompletableFuture<Product> product(@Argument String id) {
    return CompletableFuture.supplyAsync(() -> {
      return repo.findProductById(id);
    });
  }

  @QueryMapping
  public CompletableFuture<List<Product>> products() {
    return CompletableFuture.supplyAsync(() -> {
      return repo.findAll();
    });
  }
}
```

Listing 6.8: Products Controller - Queries.

Finally, the correct development of these functionalities can be easily verified by some simple GraphQL queries to the supergraph, but, most importantly, its to be expected for these new queries to be available on the supergraph schema as shown in Listing 6.5.

Mutations

Attending to the use cases documented in Table 5.1 which represent write operations, new GraphQL Mutations were developed for each and every subgraph.
Similarly to GraphQL Queries, these developments started at the schema level. New Mutation type definitions were added with the required arguments and expected outcomes. An example for the products-subgraph is available on Listing 6.7.
In the wake of such definitions being complete, new controller methods were added to capture the desired behaviour for each of these GraphQL operations. Through the use of the **MutationMapping** mapping, Spring is able to resolve each of the defined schema mutations to the respective controller method. In the same way as GraphQL Queries, the **Argument** annotation can be used to identify any possible mutation arguments. An example of such developments is detailed in Listing 6.9 for the product-subgraph controller class.

```
1  package com.example.products;
2
3  @AllArgsConstructor
4  @Controller
5  public class ProductsController {
6
7    @Autowired
8    private final ProductRepository repo;
9
10   @MutationMapping
11   public CompletableFuture<Product> addProduct(@Argument String id,
       @Argument String name, @Argument String description){
12     return CompletableFuture.supplyAsync(() -> {
13       return repo.addProduct(id, name, description);
14     });
15   }
16 }
```

Listing 6.9: Products Controller - Mutations.

All the available solution mutations, arguments and expected results can be observed on the supergraph schema on Listing 6.5.

Database

As mentioned in Section 6.1, the data management approach chosen in the base solution consisted of static, hard-coded lists. With the intent of future-proofing the solution and making it more resilient to new database approaches, the Repository pattern was applied to all subgraphs. This consists in a new Repository layer which contains a repository abstraction and all its implementations. To include a new database approach, developers only need to add a new implementation of such abstraction with the specific considerations of that approach, denoted with the Spring annotation **Repository**. Listing 6.10 displays an example of such repository abstraction for the products-subgraph, which will then be used on the API controller as shown in Listing 6.8

```java
package com.example.products.repository;

...

public interface ProductRepository {

  public List<Product> findAll();

  public Product findProductById(String id);

  public Product addProduct(String id, String name, String description);

  public List<Product> findProductByIdBatch(List<String> id);

}
```

Listing 6.10: Product Repository.

6.2.2 Performance Strategies

Following the development of the solution functionality, there is some specific research and developments related to caching, batching and execution strategies and how to apply them to the current solution.

Execution Strategies

Starting with the executions strategies, Graphql Java enables the usage of three distinct strategies (Java 2022):

- AsyncExecutionStrategy
- AsyncSerialExecutionStrategy
- SubscriptionExecutionStrategy

As the performance assessment requires the override of standard GraphQL execution strategies to determine possible impacts, there is a need to understand how this can be achieved in a Spring GraphQL application.
The base solution included some GraphQL configuration as part of the GraphQLConfiguration classes (Configuration annotated class). An example of the base configuration is detailed on Listing 6.11.

```
1  package com.example.products;
2
3  @Configuration
4  public class GraphQLConfiguration {
5
6    @Bean
7    public GraphQlSourceBuilderCustomizer federationTransform() {
8      return builder -> {
9        builder.schemaFactory((registry, wiring)->
10         Federation.transform(registry, wiring)
11           .fetchEntities(env -> null)
12           .resolveEntityType(env -> null)
13           .build()
14       );
15     };
16   }
17 }
```

Listing 6.11: Base GraphQL Configuration.

To effectively change the execution strategy, the underlying GraphQL object needs to be accessed which allows for change of the different GraphQL Operations (Query, Mutation and Subscription) execution strategies. A specific example of how to change the execution strategy of GraphQL Queries for a specific application can be found on Listing 6.12, more specifically on Line 16 where the AsyncSerialExecutionStrategy is specified for GraphQL Queries.

```
1  package com.example.products;
2
3  @Configuration
4  public class GraphQLConfiguration {
5
6    @Bean
7    public GraphQlSourceBuilderCustomizer sourceBuilderCustomizer() {
8      return builder -> {
9        builder.schemaFactory((registry, wiring)->
10         Federation.transform(registry, wiring)
11           .fetchEntities(entityDataFetcher)
12           .resolveEntityType(new ClassNameTypeResolver())
13           .build()
14       );
15       builder.configureGraphQl(graphQlBuilder -> {
16           graphQlBuilder.queryExecutionStrategy(new
     AsyncSerialExecutionStrategy());
17           graphQlBuilder.preparsedDocumentProvider(preparsedCache);
18         }
19       );
20
21     };
22   }
23 }
```

Listing 6.12: GraphQL Configuration - Execution Strategy Example.

Caching Strategies

Succeding the execution strategies, caching strategies were explored. On this specific strategy point, two different points of customization were identified within GraphQL Java and

Apollo Federation:

- Server-side Caching
- Entity Level Cache Hints

Starting off with server-side caching, in a similar way to execution strategies, a caching engine can be assigned to a Spring GraphQL application through the change of the GraphQL configuration. To achieve this the PreparsedDocumentProvider can be configured to cache and reuse Document instances, thus avoiding the re-parsing and re-validation of GraphQL operations within GraphQL Java (Spring 2023d). Listing 6.13 depicts an example of how this was achieved utilizing the Caffeine caching library following the recommended configuration (Manes 2023; Spring 2023a).

```
package com.example.products;

...

@Configuration
public class GraphQLConfiguration {

  @Bean
  public GraphQlSourceBuilderCustomizer sourceBuilderCustomizer() {

...

    //Caching - Server-side
    Cache<String, PreparsedDocumentEntry> cache = Caffeine.newBuilder().
    maximumSize(10_000).build();
    PreparsedDocumentProvider preparsedCache = (executionInput,
    computeFunction) -> {
      Function<String, PreparsedDocumentEntry> mapCompute = key ->
    computeFunction.apply(executionInput);
      return cache.get(executionInput.getQuery(), mapCompute);
    };

    return builder -> {
      builder.schemaFactory((registry, wiring)->
        Federation.transform(registry, wiring)
          .fetchEntities(entityDataFetcher)
          .resolveEntityType(new ClassNameTypeResolver())
          .build()
      );
      builder.configureGraphQl(graphQlBuilder -> {
          graphQlBuilder.preparsedDocumentProvider(preparsedCache);
        }
      );

    };
  }
}
```

Listing 6.13: GraphQL Configuration - Caching Strategy Example.

Next up, as the Apollo Server recognizes the **cacheControl** directive, it's possible to define the application caching restrictions at the subgraph schema level, following Apollo's GraphQL configuration (Apollo GraphQL 2022i). An example of such schema Entity Level cache hints can be found at the product-subgraph schema level as defined in Listing 6.14.

```
type Product @key(fields: "id")@cacheControl(maxAge: 30){
    id: ID!
    name: String!
    description: String
}

enum CacheControlScope {
  PUBLIC
  PRIVATE
}

directive @cacheControl(
  maxAge: Int
  scope: CacheControlScope
  inheritMaxAge: Boolean
) on FIELD_DEFINITION | OBJECT | INTERFACE | UNION
```

Listing 6.14: Products - Entity Level Cache Hints.

Batching Strategies

Last but not least, batching strategies were also explored and developed for the performance assessment.
Within Spring GraphQL Java, it's possible to configure batching strategies through the use of the application BatchLoaderRegistry. Although there are multiple approaches to achieving method batching, the chosen one for the final solutions was the use of the **BatchMapping** annotation which replaces and overrides the use of the BatchLoaderRegistry (Spring 2023e). An example of such batch mapping controller method can be observed in Listing 6.15.

```
package com.example.products;

...

@AllArgsConstructor
@Controller
public class ProductsController {

  ...

  @BatchMapping(typeName = "String")
  public Mono<Map<String, Product>> productBatch(@Argument List<String>
    id) {
    return
      Mono.just(repo.findProductByIdBatch(id).stream()
        .collect(Collectors.toMap(Product::id, Function.identity())));
  }

  ...

}
```

Listing 6.15: Products Controller - Batching.

Chapter 7

Evaluation and Experimentation

The current chapter aims to document all the experimentation and evaluation processes and results. First, the hypothesis will be specified taking into account the problems and objectives identified in Chapter 1 and all the contextualization contained in Chapter 2.
Next, the Goal Question Metric (GQM) approach will be applied to organize and standardize the hypothesis assessment process through the definition of goals, its questions and metrics.
Finally, the data collection and interpretation are detailed, including the tools used, some of the data collected and what insights can be interpreted from it.

7.1 Hypothesis

As identified in the Introduction Chapter (Chapter 1), the main driver of the current document is determining the impact of caching, batching or execution strategies on GraphQL Federation performance.
After all the research done as part of Chapter 2, the hypothesis to be tested and evaluated it's whether the correct selection of caching, batching or execution strategies can affect, in a positive way, the overall performance of a GraphQL Federation Architecture.
Ideally, and to verify the truth of such a hypothesis as less ambiguously as possible, such performance measurement should aim for a high number of subgraph scenarios where performance change might be multiplied depending on the level of federated types shared across GraphQL services (high-complexity scenarios).

7.2 Methodology

7.2.1 Goal Question Metric (GQM)

The Goal Question Metric (GQM) approach aims for the "specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data" (Basili, Caldiera, and Rombach 1994). The measurement system consists of three different levels, also implied by the approach name:

- Goal: Also known as the "Conceptual Level", a goal is defined as aiming for an object, a product, a process or a resource. It is also contextualized to a specific environment, point of view or external elements.
- Question: Also known as the "Operation Level", a question describes how the assessment or evaluation of a specific goal may be performed. While materializing a question there is the attempt of identifying the object of the measurement.
- Metric: Also known as the "Quantitative Level", a metric is the lowest level in the GQM measurement system. Metrics are defined taking into account the Goal and

Question upper levels and can be used as data structuring concepts for the data collected throughout the tests or evaluation of the defined goal.

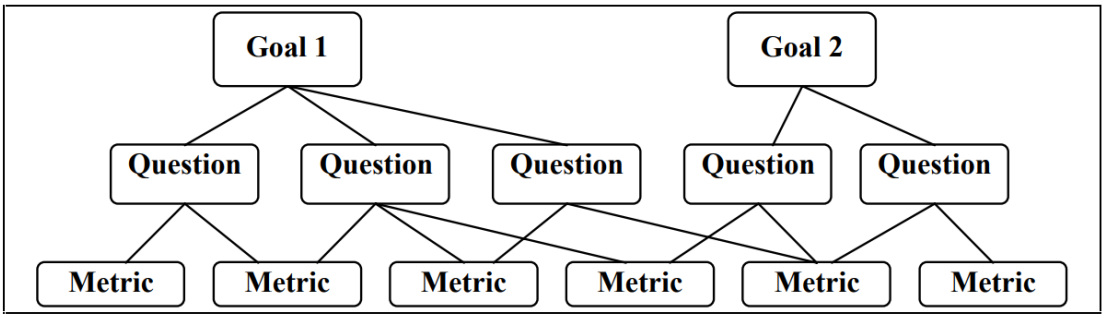


Figure 7.1: GQM Model (Basili, Caldiera, and Rombach 1994)

The hierarchical structure of the GQM Model can be observed in Figure 7.1. The model definition starts with the topmost level, the Goal. A goal is defined as "specifying the purpose of measurement, object to being measured, issue to be measured, and viewpoint from which the measure is taken" (Basili, Caldiera, and Rombach 1994). From the defined goal a set of questions are developed, further breaking down the goal/objective/problem into its significant parts. Finally, the group of metrics is defined as adequate to the goal and questions previously defined.
Evidentially, and as perceived in the GQM Model representation (Figure 7.1), a metric can be used to answer multiple questions under the same goal.

7.2.2 Performance

The performance will need to be tested and evaluated as the main focus of the hypothesis defined in Section 7.1. Taking into account the previous specification of the GQM approach, the GQM Model for the current performance evaluation will comprise the following sub-models.

Caching

When evaluating the impact of caching strategies the GQM Model will include the following details, also depicted in Figure 7.2:

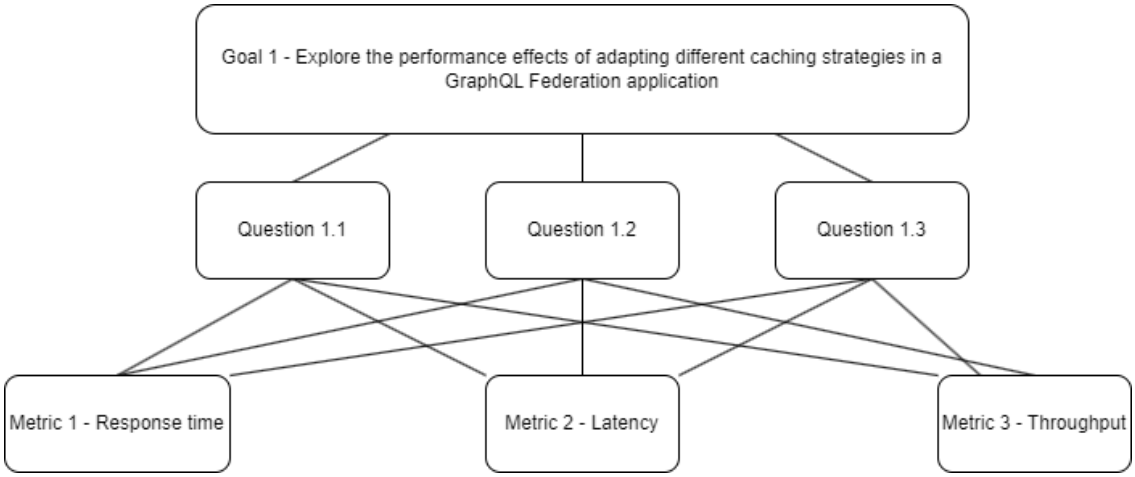


Figure 7.2: GQM Model Caching Goal

- Goal 1: Explore the performance effects of adapting different caching strategies in a GraphQL Federation application
- Questions:
  - Q 1.1: What are the effects on the performance of a GraphQL Federation application using standard caching strategies?
  - Q 1.2: What are the effects on the performance of a GraphQL Federation application using server-side caching?
  - Q 1.3: What are the effects on the performance of a GraphQL Federation application using entity and gateway level cache hints?
- Metrics: Response time, Latency and Throughput.

Batching

When evaluating the performance impact of different batching strategies the GQM Model will include the following details, also depicted in Figure 7.3:

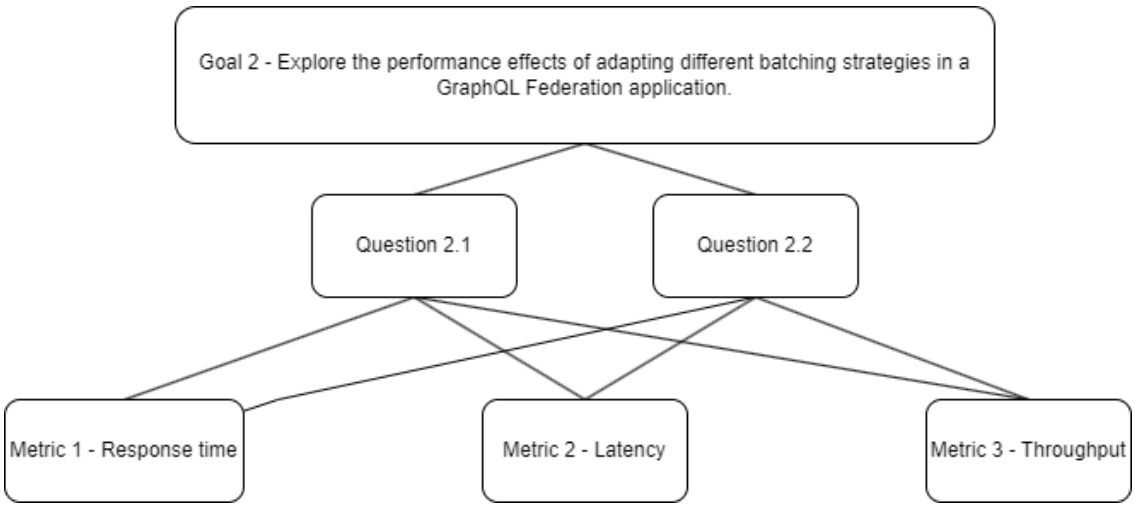


Figure 7.3: GQM Model Batching Goal

- Goal 2: Explore the performance effects of adapting different batching strategies in a GraphQL Federation application.

- Questions:
  - Q 2.1: What are the effects on the performance of a GraphQL Federation application using standard or no batching strategies?
  - Q 2.2: What are the effects on the performance of a GraphQL Federation application using request batching (Chapter 2.3.4)?
- Metrics: Response time, Latency and Throughput.

Execution Strategies

When evaluating the performance impact of different execution strategies the GQM Model will include the following details, also depicted in Figure 7.4:

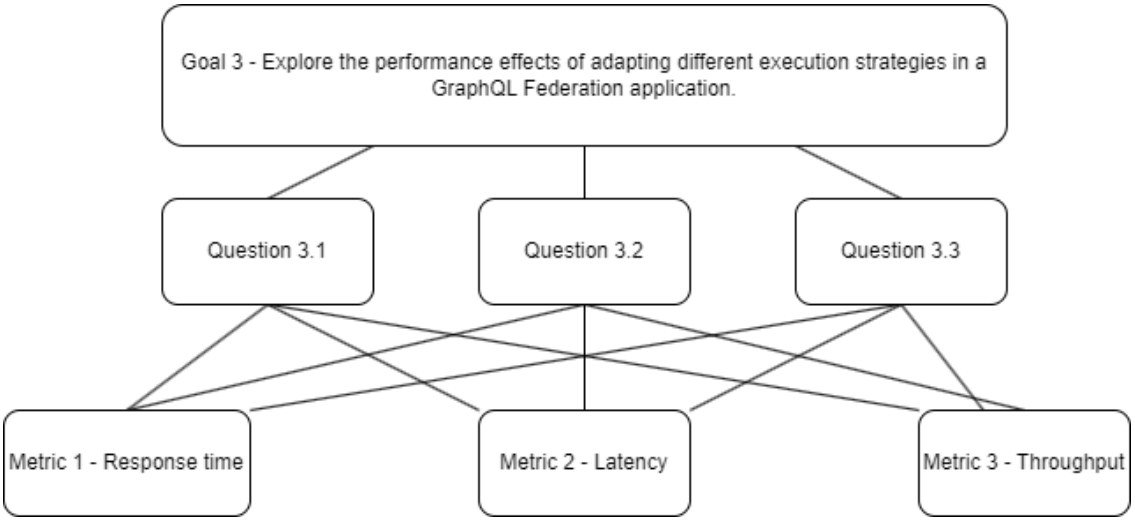


Figure 7.4: GQM Model Execution Strategies Goal

- Goal 3: Explore the performance effects of adapting different execution strategies in a GraphQL Federation application.
- Questions:
  - Q 3.1: What are the effects on the performance of a GraphQL Federation application using default execution strategies?
  - Q 3.2: What are the effects on the performance of a GraphQL Federation application using asynchronous execution strategies?
  - Q 3.3: What are the effects on the performance of a GraphQL Federation application using serial execution strategies?
- Metrics: Response time, Latency and Throughput.

7.2.3 Data Collection - Tools

Tools to obtain data for such metrics may vary but tools such as JMeter and Apollo Studio are well contextualized within the author's language preference for the test application, as well as for GraphQL Federation itself (Apollo Studio).
With these tools, each metric can be correctly populated and will enable the questions to be answered. Each of the metrics is recorded to categorize each scenario's performance.
For this GQM exercise, the tools that are going to be used are the following:

- JMeter: Supergraph Router metric collection.
- Prometheus: Subgraph monitoring that is going to be used as a support tool to JMeter.

- Grafana: Prometheus data format to allow for extra insights.

JMeter

JMeter is an open-source Java application which allows for performance and load testing of solutions (JMeter 2023). Due to its high portability, there were no specific imposed requirements on the solution resulting from the implementation phase (Chapter 6) which on its own represented a huge advantage as a data collection tool.
GraphQL operations are supported by JMeter by default which means that testing the supergraph queries is quite straightforward. Image 7.5 represents an example of the JMeter infrastructure required to test and collect information regarding the GraphQL Query to retrieve all products.

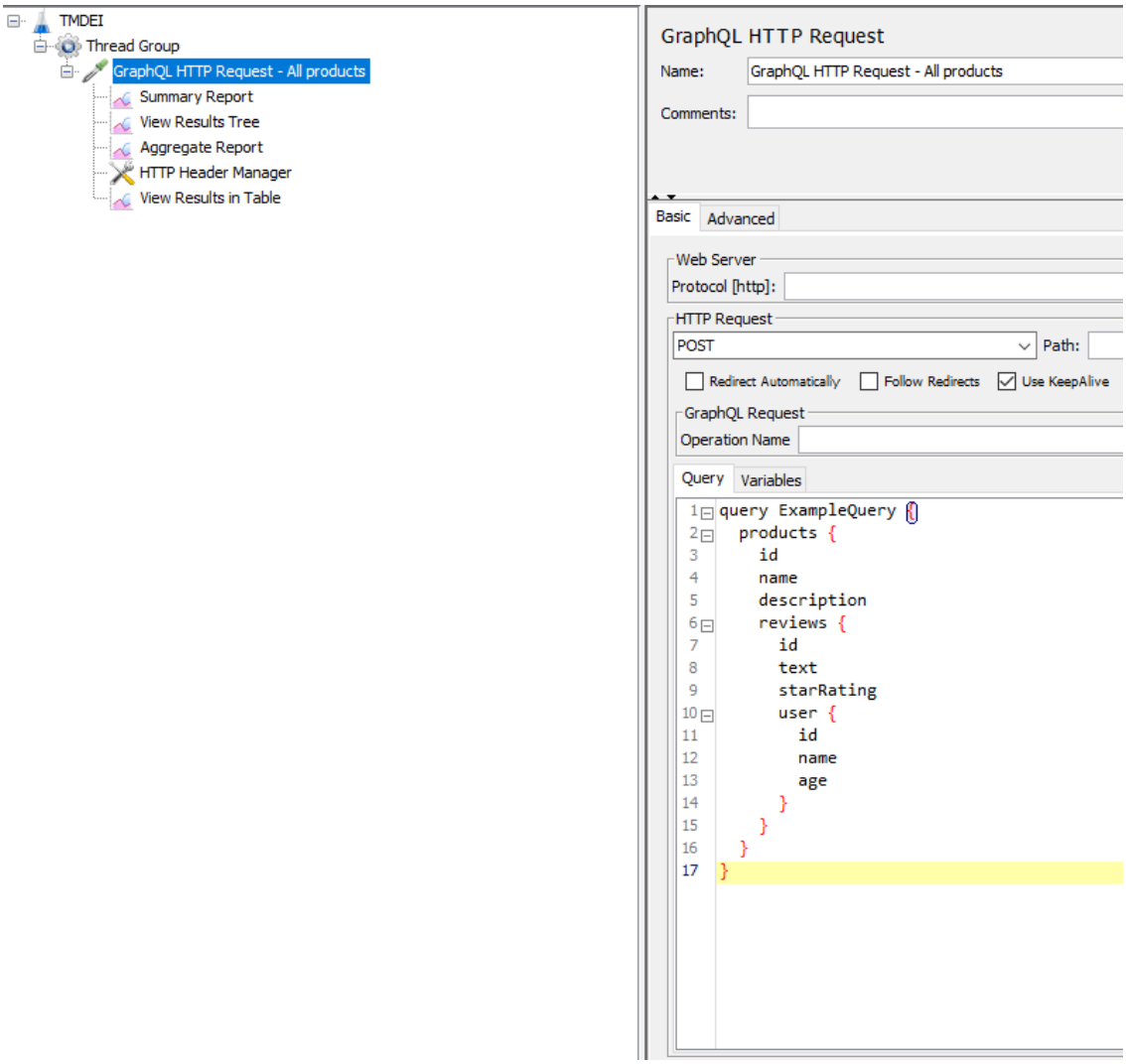


Figure 7.5: JMeter - Data Collection Example

Following this infrastructure, JMeter offers a set of Listeners, which can be used to collect the different metrics required for the GQM process and questions. The following listeners can be especially useful for the current thesis methodology:

- Summary Report: Besides the request error percentage and the size of sent and received data, the summary report collects information regarding the API throughput.

- Aggregate Report: Besides reporting about the error percentage and throughput like the summary report, the aggregate report provides information regarding the maximum and the minimum response times.
- View Results in Table: This type of Listener will be especially useful due to its latency reporting. As the name implies, all JMeter request information is organized in a table format, which allows for maximum and minimum latency queries.

Prometheus

Prometheus is an open-source system monitoring and alerting toolkit (Prometheus 2023). On its own Prometheus is already a pretty powerful tool, offering a variety of metrics and query capability to manage them, but when joined with Spring Boot Actuator (Spring 2023c) dependency management and Micrometer (Micrometer 2023) metrics facade, it allows for detailed level reporting regarding individual application instances (Spring 2023b).
Oppositely to JMeter, Prometheus does require some extra configuration on the solution to enable the Spring Boot Actuator endpoints for Prometheus. Listing 7.1 displays an example of the additions required for the products-subgraph application YAML to allow for Prometheus scrape export through the **/actuator/prometheus** endpoint.

```yaml
spring:
  graphql:
    graphiql:
      enabled: true

management:
  endpoints:
    web:
      exposure:
        include: health,info,prometheus,metrics
```

Listing 7.1: Prometheus Reporting - Spring Boot Application

Despite the fact that all the information is now available, the Prometheus query capabilities will only be available if the Prometheus server itself is installed. To configure the Prometheus server to consume the Spring Boot Actuator endpoint information, a configuration such as the one presented in Listing 7.2 will be required, specifying the new scrape configuration and URL to be used.

```yaml

...

scrape_configs:

  ...

  - job_name: "spring-actuator-products"
    metrics_path: 'actuator/prometheus'
    scrape_interval: 5s
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:8080"]

  ...
```

Listing 7.2: Prometheus Server Configuration

Grafana

Grafana is an open-source platform for monitoring and observability. Besides Prometheus, Grafana supports the visualization of metrics, logs and traces from data sources like Loki, Elasticsearch, InfluxDB, Postgres and more (Grafana 2023).
For the current thesis use case, Grafana will be used to consume the Prometheus data source information and monitor each individual subgraph throughout the performance assessment and measurement exercise. Similar to Prometheus, Grafana does require a new server to be set up and the Prometheus data source to be added as depicted by Image 7.6.

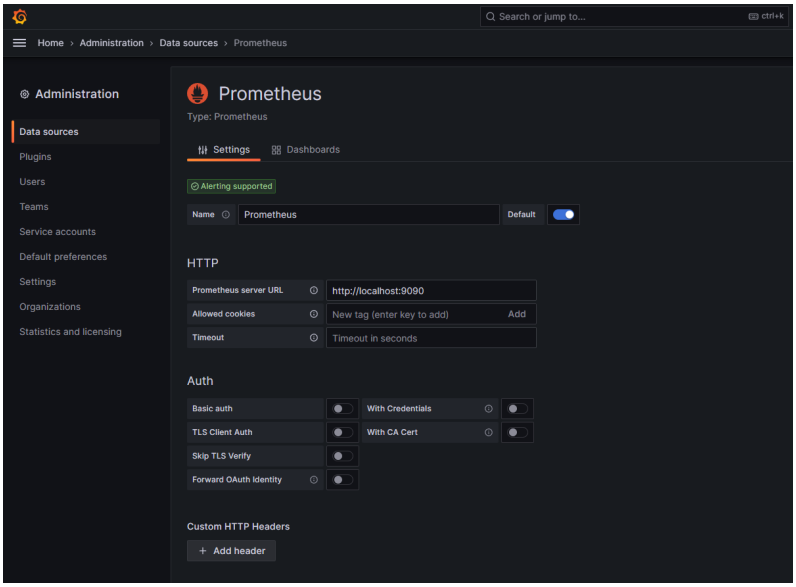


Figure 7.6: Grafana - Example Prometheus Data Source

As Grafana supports a comprehensive library containing a wide variety of pre-configured dashboards, a dashboard was chosen from this library to support the subgraphs monitoring. Image 7.7 depicts an example of the SpringBoot APM Dashboard for the products-subgraph, which is configured to receive Prometheus data and organize it in a wide variety of useful information through graphs, gauges and stats (prakarsh 2020).

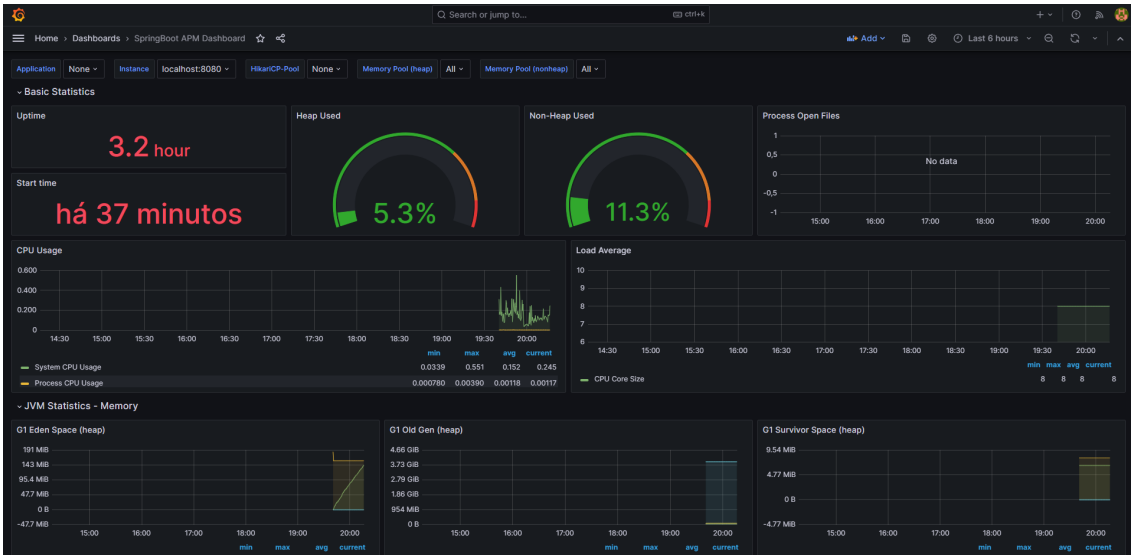


Figure 7.7: Grafana - Spring APM Dashboard Products

7.2.4 Measurement Planning

As the GQM models and the data collection tools are now defined, the current section focus on detailing how the measurements will take place, what scenarios will be involved for each strategy and in which conditions or what part of the solution will be measured. Possible outcomes of such measure can already be defined at this phase as well (Solingen and Berghout 1999).
As JMeter allows the specification of the request volume, for each of the questions a high-volume and low-volume scenario will be measured. This might be insightful or further contextualize possible results and their understanding. Furthermore, due to the GraphQL Federation nature, certain GraphQL Queries may imply a higher complexity query plan thus the usage of more subgraphs. With this in mind, for each of the GQM questions defined in Section 7.2.1, a high-complexity and low-complexity GraphQL operation scenario will be included. Finally, GraphQL subscriptions will also be in scope for the measurement so that the impact of the different strategies, if any, can be analysed on each GraphQL operation. Table 7.1 aims to provide an objective meaning to the high and low complexity, as well as high and low volumes.

Table 7.1: GQM Measurement Plan - Complexities and Volumes

Measure	**Explanation**
High-Volume	JMeter Request Volume will be set to 1000 requests
Low-Volume	JMeter Request Volume will be set to 100 requests
High-Complexity	JMeter Request will query all solution products
Low-Complexity	JMeter Request will query all solution users

With the objective of providing an overview of the measurement scope regarding GraphQL Queries, Tables 7.2, 7.3, 7.4 contain each of the scenarios for each of the GQM Model Questions that will be assessed. All the information regarding each of the GQM questions can be found in Section 7.2.1.

Table 7.2: GQM Caching Measurement Plan - GraphQL Queries Scenarios

Question	**Scenario**
1.1	High-volume, High-complexity Standard Strategy
1.1	High-volume, Low-complexity Standard Strategy
1.1	Low-volume, High-complexity Standard Strategy
1.1	Low-volume, Low-complexity Standard Strategy
1.2	High-volume, High-complexity Server-Side Caching
1.2	High-volume, Low-complexity Server-Side Caching
1.2	Low-volume, High-complexity Server-Side Caching
1.2	Low-volume, Low-complexity Server-Side Caching
1.3	High-volume, High-complexity Entity Cache-Hints Caching
1.3	High-volume, Low-complexity Entity Cache-Hints Caching
1.3	Low-volume, High-complexity Entity Cache-Hints Caching
1.3	Low-volume, Low-complexity Entity Cache-Hints Caching

Table 7.3: GQM Batching Measurement Plan - GraphQL Queries Scenarios

Question	**Scenario**
2.1	High-volume, High-complexity Standard Strategy
2.1	High-volume, Low-complexity Standard Strategy
2.1	Low-volume, High-complexity Standard Strategy
2.1	Low-volume, Low-complexity Standard Strategy
2.2	High-volume, High-complexity Request Batching
2.2	High-volume, Low-complexity Request Batching
2.2	Low-volume, High-complexity Request Batching
2.2	Low-volume, Low-complexity Request Batching

Table 7.4: GQM Caching Execution Plan - GraphQL Queries Scenarios

Question	**Scenario**
3.1	High-volume, High-complexity Standard Strategy
3.1	High-volume, Low-complexity Standard Strategy
3.1	Low-volume, High-complexity Standard Strategy
3.1	Low-volume, Low-complexity Standard Strategy
3.2	High-volume, High-complexity Asynchronous Execution
3.2	High-volume, Low-complexity Asynchronous Execution
3.2	Low-volume, High-complexity Asynchronous Execution
3.2	Low-volume, Low-complexity Asynchronous Execution
3.3	High-volume, High-complexity Serial Execution
3.3	High-volume, Low-complexity Serial Execution
3.3	Low-volume, High-complexity Serial Execution
3.3	Low-volume, Low-complexity Serial Execution

Similarly, Table 7.5 provides an overview of the measurement scope regarding GraphQL Mutations for each of the defined GQM Questions.

Table 7.5: GQM Measurement Plan - GraphQL Mutations Scenarios

Strategy	Question	Scenario
Caching	1.1	High-volume Standard Strategy
Caching	1.1	Low-volume Standard Strategy
Caching	1.2	High-volume Server-Side Caching
Caching	1.2	Low-volume Server-Side Caching
Caching	1.3	High-volume Entity Cache-Hints Caching
Caching	1.3	Low-volume Entity Cache-Hints Caching
Batching	2.1	High-volume Standard Strategy
Batching	2.1	Low-volume Standard Strategy
Batching	2.2	High-volume Request Batching
Batching	2.2	Low-volume Request Batching
Execution	3.1	High-volume Standard Strategy
Execution	3.1	Low-volume Standard Strategy
Execution	3.2	High-volume Asynchronous Execution
Execution	3.2	Low-volume Asynchronous Execution
Execution	3.3	High-volume Serial Execution
Execution	3.3	Low-volume Serial Execution

As perceived by the GQM models present in Section 7.2.1, for each of these scenarios the three metrics (latency, response time and throughput) will be collected by using the tools mentioned in Section 7.2.3.

7.2.5 Data Collection

This section will focus on collecting data from the multiple scenarios defined in the Measurement Plan present in Section 7.2.4. This is a direct representation of the GQM Data Collection and Data Interpretation phases (Solingen and Berghout 1999). The structure of this chapter will follow the GQM hierarchy, thus subsections will comprise GQM questions, their metrics and the interpretation of such results.

Caching Strategies - Standard Strategy

This section refers directly to Goal 1 and Question 1.1 depicted in Figure 7.2. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 1.1) for GraphQL Queries are represented in Table 7.6.

Table 7.6: Data Collection - Standard Caching Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	10	228	212	10	228	192.9
High	Low	3	37	33	3	37	198.6
Low	High	9	23	13	9	23	20.1
Low	Low	3	8	5	3	8	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 1.1) are equally represented in Table 7.7.

Table 7.7: Data Collection - Standard Caching Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	49	45	3	49	198.1
Low	5	12	12	5	12	20.1

Caching Strategies - Server-side Strategy

This section refers directly to Goal 1 and Question 1.2 depicted in Figure 7.2. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 1.2) for GraphQL Queries are represented in Table 7.8.

Table 7.8: Data Collection - Server-side Caching Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	9	210	200	9	210	192.3
High	Low	2	48	32	2	48	198.8
Low	High	9	20	15	9	20	20.1
Low	Low	3	8	5	3	8	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 1.2) are equally represented in Table 7.9.

Table 7.9: Data Collection - Server-side Caching Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	58	52	3	58	197.6
Low	5	30	14	5	30	20.1

At first glance, the implementation of server-side caching does not appear to cause performance implications. In most scenarios, the response time, latency and throughput are maintained when compared with the standard caching strategy.
The biggest differences are recorded in the high-volume and high-complexity scenarios. In the specific case of GraphQL Queries, the 99 percentile of response time was reduced from 212ms to 200ms which represents an improvement of 6% for the 1000 JMeter requests. Despite this, GraphQL Mutation performance was aggravated from a 45ms 99 percentile of response time to 52ms, which in itself represents a 15% increase in response time for the 1000 JMeter requests.

Caching Strategies - Entity-Level Hints Strategy

This section refers directly to Goal 1 and Question 1.3 depicted in Figure 7.2. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 1.3) for GraphQL Queries are represented in Table 7.10.

Table 7.10: Data Collection - Entity-Level Cache Hints Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	10	321	285	10	321	189
High	Low	3	44	34	3	44	198.6
Low	High	10	22	13	10	22	20.1
Low	Low	3	8	5	3	8	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 1.3) are equally represented in Table 7.11.

Table 7.11: Data Collection - Entity-Level Cache Hints Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	40	35	3	40	197.9
Low	5	23	11	5	23	20.2

Contrary to the server-side caching strategy, entity-level hints collected data already produced some high variances in response time in high-volume and high-complexity scenarios, when compared with the standard strategy. Noteworthy results consist of the following:

- Response times (M1) - Query: There was a great deterioration of 34% in the 99 percentile of response time in high-volume and high-complexity scenarios, from 212ms (Standard) to 285ms (Entity-Level Hints). Despite this, response times seem to have been maintained in other GraphQL Query scenarios.
- Response times (M1) - Mutation: Response time has improved by 23% in high-volume scenarios, from 45ms (Standard) to 35ms (Entity-Level Hints) although the same situation cannot be verified in low-volume scenarios.
- Throughput (M3) - No major aggravations to throughput were registered.

Batching Strategies - Standard Strategy

This section refers directly to Goal 2 and Question 2.1 depicted in Figure 7.3. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 2.1) for GraphQL Queries are represented in Table 7.12.

Table 7.12: Data Collection - Standard Batching Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	10	228	212	10	228	192.9
High	Low	3	37	33	3	37	198.6
Low	High	9	23	13	9	23	20.1
Low	Low	3	8	5	3	8	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard batching strategy (Question 2.1) are equally represented in Table 7.13.

Table 7.13: Data Collection - Standard Batching Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	49	45	3	49	198.1
Low	5	12	12	5	12	20.1

Batching Strategies - Request Batching Strategy

This section refers directly to Goal 2 and Question 2.2 depicted in Figure 7.3. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 2.2) for GraphQL Queries are represented in Table 7.14.

Table 7.14: Data Collection - Request Batching Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	10	255	232	10	255	192.2
High	Low	3	49	43	3	49	198.2
Low	High	10	24	14	10	24	20.1
Low	Low	3	10	5	3	10	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 2.2) are equally represented in Table 7.15.

Table 7.15: Data Collection - Request Batching Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	47	39	3	47	197.8
Low	5	22	11	5	22	20.2

Following previous non-standard strategies, the request batching implementation also implied some changes in the recorded metrics, mainly response times.
By comparing with the standard GraphQL batching strategy, the next points can be highlighted:

- Response times (M1) - Query: The solution response time did suffer some deterioration. Namely, the 99 percentile has increased by 9% from 212ms (Standard) to 232ms (Request Batching) on high-volume and high-complexity scenarios. The same trend is verified for high-volume low-complexity requests.
- Response times (M1) - Mutation: Response time improvements have been registered in GraphQL mutations for both high-volume (15%) and low-volume scenarios (9%).
- Throughput (M3) - No major aggravations to throughput were registered.

Execution Strategies - Standard Strategy

This section refers directly to Goal 3 and Question 3.1 depicted in Figure 7.4. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 3.1) for GraphQL Queries are represented in Table 7.16.

Table 7.16: Data Collection - Standard Execution Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	10	228	212	10	228	192.9
High	Low	3	37	33	3	37	198.6
Low	High	9	23	13	9	23	20.1
Low	Low	3	8	5	3	8	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard execution strategy (Question 3.1) are equally represented in Table 7.17.

Table 7.17: Data Collection - Standard Execution Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	49	45	3	49	198.1
Low	5	12	12	5	12	20.1

Execution Strategies - Asynchronous Strategy

This section refers directly to Goal 3 and Question 3.2 depicted in Figure 7.4. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 3.2) for GraphQL Queries are represented in Table 7.18.

Table 7.18: Data Collection - Asynchronous Execution Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	9	323	312	9	323	188.4
High	Low	3	58	50	3	58	198.1
Low	High	10	17	15	10	17	20.2
Low	Low	3	8	8	3	8	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 3.2) are equally represented in Table 7.19.

Table 7.19: Data Collection - Asynchronous Execution Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	58	57	3	58	197.8
Low	5	24	15	5	24	20.2

Adapting the solution to follow the GraphQL Java asynchronous execution strategy caused some implications for performance across all scenarios and GraphQL operations. The following points were developed by comparing with the standard GraphQL execution strategy approach:

- Response times (M1) - Query: The application of this strategy hindered the solution performance. The most significant impact can be observed in high-volume scenarios

where the response times have increased by 47% and 51% in high-complexity and low-complexity scenarios respectively. This behaviour is meaningful in low-volume scenarios as well.

- Response times (M1) - Mutation: Similarly to the results in GraphQL queries, GraphQL mutations have been affected by changes to the solution execution strategy. The most apparent impact is in the high-volume scenario where the 99 percentile response time has increased from 45ms (Standard) to 57ms (Asynchronous).
- Throughput (M3) - There is some lesser impact on the solution throughput most noticeable in the high-volume and high-complexity scenario.

Execution Strategies - Serial Strategy

This section refers directly to Goal 3 and Question 3.3 depicted in Figure 7.4. All the data collected regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 3.3) for GraphQL Queries are represented in Table 7.20.

Table 7.20: Data Collection - Serial Execution Strategy

Volume	Complexity	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	High	10	324	289	10	324	189.5
High	Low	3	53	48	3	53	198.1
Low	High	10	23	14	10	23	20.1
Low	Low	3	9	6	3	9	20.2

The collected data for GraphQL Mutations data regarding Response times (M1), Latency (M2) and Throughput (M3) in a standard caching strategy (Question 3.3) are equally represented in Table 7.21.

Table 7.21: Data Collection - Serial Execution Strategy - Mutations

Volume	Min M1	Max M1	99% M1	Max M2	Min M2	M3
High	3	34	31	3	34	197.8
Low	5	28	11	5	28	20.2

Similar to the implementation of asynchronous execution strategies, the GraphQL Java asynchronous serial execution strategy did hinder the solution performance. Despite this, GraphQL mutations did benefit from this change in strategy when compared with the standard approach. Other points worth mentioning when comparing with the standard execution strategy performance:

- Response times (M1) - Query: GraphQL queries were impacted by the change in execution strategy, although not as impacted as the implementation of the asynchronous strategy. Response times in high-volume and high-complexity scenarios increased from a 99 percentile of 212ms (Standard) to 289ms (Serial) which represents a 36% deterioration in performance. Such impact is also meaningful in other types of scenarios.
- Response times (M1) - Mutation: As stated at the beginning of the sub-section, mutations did benefit from the implementation of such a strategy. Response times have evolved from the 99 percentile of 45ms (Standard) to 31ms (Serial) which translates to a 32% improvement.
- Throughput (M3) - There is some lesser impact on the solution throughput most noticeable in the high-volume and high-complexity scenario.

General Analysis

Through the observation of the multiple collected data, it's possible to observe the following behaviours:

- Latency (M2) and Response times (M1): Due to the fact that the data collection occurs from JMeter Requests which are consumed at the local machine, most if not all latencies are equal to the recorded response times. If the performance assessment solution was hosted on cloud infrastructure it is expected that these values are affected in some shape or form.
- Standard Strategy: All the standard strategy performance data was collected for the same application, therefore the results are similar in nature. Although it's possible to apply some judgement to the values obtained for all the metrics in standard GraphQL Federation strategies, the main usage of this data is to provide a baseline for the other GQM questions data allowing for objective comparisons and conclusions.
- Scenarios: In most of the data collected, it's possible to conclude that the high-volume and high-complexity scenarios provided firmer grounds than low-volume and low-complexity for some of the conclusions, as the Graph Federation architecture and subgraphs usage increases. Despite this, most deltas in response time, latency and throughput between the standard strategy and a custom one are not too significant. In a small or less active solution, any of the changes to strategies may not cause any restrictions on performance. Similarly, if the trend maintains, higher volumes and complexities may be of concern when implementing such strategies.

7.3 Outcomes

Through the analysis of all the collected data and its interpretation in Section 7.2.5, it's possible to conclude that indeed different caching, batching or execution strategies impact GraphQL Federation performance as a whole.

Goal 1 - Caching Strategies

Specifically to the different caching strategies, while server-side caching was the best performant in GraphQL queries with a 99 percentile response time of 200ms in high volume and complexity scenarios, GraphQL Mutations were improved by the adoption of Entity-Level cache hints at the Schema level of each subgraph. This is an important insight to drive implementation options. If a given solution is GraphQL query-heavy, Entity-Level cache hints may not be the correct choice for a caching strategy, and server-side caching may prove to be the best option as per the collected data.
Despite this, in low-volume scenarios, discrepancies between the multiple strategies had a maximum of 3ms (between Server-side caching and Standard approach), thus if the GraphQL Federation handles low volumes, extra considerations need to be taken to determine if caching strategy customization is a worthwhile endeavour.

Goal 2 - Batching Strategies

Next, the results obtained for batching strategies suggest that the implementation of batching per request basis does impact negatively the performance of the GraphQL Federation solution with a 9% increase in response times. Despite this, this strategy makes 15% and 9%

performance improvements on GraphQL mutations in high-volume and low-volume scenarios respectively.
Therefore, from the results obtained from this performance assessment, if a federated architecture is a mutation-heavy solution, for example for a CQRS pattern representation of command applications (which are responsible for write operations), request batching presents improvements in performance.
Contrarily, a query-focused application may see its performance hindered by such an approach.

Goal 3 - Execution Strategies

Finally, when reviewing the results of the different execution strategies, it becomes apparent that the configuration of an asynchronous strategy impacts the solution negatively in all tested GraphQL operations.
The same trend is maintained on the serial execution strategy for GraphQL queries, but GraphQL Mutations did benefit from this approach.
Thus, from the data obtained from this performance assessment its possible to conclude that the standard execution strategies applied by GraphQL Java in a federated architecture perform the best in response time. Further analysis and consideration need to be taken if such standard strategies need to be customized, as it was perceived in this performance assessment, high-performance impacts can be expected from this.

7.4 Threats to Validity

Even though the performance assessment scope was already documented in the measurement planning and data collection tools sections, its necessary to accurately identify the threats to its validity so that these can be worked upon in new iterations or by future performance assessments driven by this thesis. The following list documents the threats identified throughout this exercise:

- Machine: All the ran tests and collected data result of JMeter collection on the author's machine. Although efforts were made to ensure that all measurements were under similar conditions, it would be meaningful to extend such tests through multiple machines as there is the possibility for different results.
- Strategy Configuration: All the implemented strategies were documented in Chapter 6. Despite this, it's important to understand that such strategies do have some configuration alternatives. For example, in the specific case of the server-side caching strategy, there are other caching engines other than Caffeine that can be used, and, even in the specific case of Caffeine, the maximum size parameter can be changed which can cause different results than the ones observed for a maximum size of 10.
- Performance Assessment Solution: As expected, the solution that was the aim of the performance assessment is not a full-fledged production-ready application. Efforts were made to make an accurate simulation of business scenarios and use cases through the requirements documented in Chapter 5 so that this gap was reduced but there can be real scenarios that are more complex than the ones utilized in this assessment. Further efforts were made with the adoption of high-volume scenarios which comprised 1000 request scenarios to test the application throughput and simulate API request stress upon it.

- Tools: As discussed in Section 7.2.3, JMeter is a highly pragmatic approach to data collection for performance tests. In the specific application used for these tests, JMeter is only allowed to collect that from the GraphQL Federation architecture as a whole (supergraph level). Efforts were made to include subgraph-level monitoring using Prometheus and Grafana but the implied time constraints for this thesis did not allow for a deeper understanding and development of these tools. It would be possible to automate the process of extracting the scrapes from the subgraph using Prometheus and have it all in a single Grafana dashboard to allow for easy and highly repeatable performance analysis at each subgraph. Such an approach could also produce further contextualized results for the performance assessment, which would enrich the insights documented in Section 7.3.

Chapter 8

Conclusion

This chapter represents the conclusion of this thesis document which focuses on explaining the work and objectives that were achieved, any challenges or obstacles faced during the elaboration of such work, future improvements, and final considerations or reflections.

8.1 Achievements

In Chapter 1, the main objective of this thesis was defined as understanding the impacts of execution, caching and batching strategies on a GraphQL architecture. Furthermore, it was important to understand which strategies are available and design an approach to assess the impact of their implementation. Looking back at these objectives, it's possible to conclude that all of them were complete.
Firstly, execution, caching and batching strategies were explored and documented allowing for the performance assessment approach to start. Next, in Chapter 4, an idea was chosen through the use of AHP to attain a GraphQL Federation solution, upon which performance tests can be run.
Following this, Chapter 5, Chapter 6 and Chapter 7 document the process of analysing possible base solution candidates, the design of the solution and the whole process from implementation to performance testing and analysis/interpretation. This includes, in Section 7.3, the answers to the defined goals and questions in the GQM approach which directly relate to the thesis research question.
Despite this, further contextualization of possible threats to validity was documented in Section 7.4 to provide the scope in which the results obtained for this thesis performance experimentation are valid and, also identify possible improvement points for future iterations. To enable such future iterations, as mentioned in Chapter 1, all the utilized source code and collected assessment data are available on a public GitHub repository, with the objective of allowing further iterations of this process and experiment.

8.2 Challenges and Future Work

The work carried out throughout this thesis was not challenge-free. GraphQL Federation it's constantly evolving which implied a lot of research, investigation and re-research during the elaboration. Furthermore, time is a very significant objective constraint for this thesis with pre-defined deadlines and delivery dates which must be met.
More specifically, in the evaluation and experimentation phase (Chapter 7), some problems were faced when adapting the solution to report Prometheus data and format it in Grafana. Currently, all the displayed data was collected through JMeter, while Prometheus and Grafana were used as support tools for subgraph monitoring. Even though JMeter is highly pragmatic, it can be restrictive in regards to what data can be collected and how this

data is formatted into a useful and intuitive format. As a future improvement, Prometheus custom scrapes and a custom Grafana dashboard can be developed to allow for an end-to-end performance analysis solution. This might accelerate the process of data collection and its analysis or comprehension.
Moreover, the performance testing can attain more levels of complexity to allow for a deeper understanding and gain further insights into GraphQL Federation performance. Currently, the only dimensions explored were volume and query complexity but there can be other dimensions such as request homogeneousness in a caching strategy test which can give extra grounds for conclusions.

8.3 Final Considerations

Despite the challenges and future improvements, all the objectives were attained through the development of this thesis. Information regarding complex themes such as microservices, architecturally meaningful patterns, GraphQL Federation peculiarities and enterprise use cases are now centralized in this document which may prove useful and help in any future work, articles or thesis.
All the code developed for the performance assessment solutions is available for everyone on GitHub (Queirós 2023). The author hopes that the approach and developed infrastructure can be re-used, improved and iterated upon in future assessments to help developers, architects and other solution stakeholders take educated decisions on its Federated Architecture.

8.4 Personal Appreciation

The work carried out across this thesis consists of an incredibly enriching experience for the author. Although some of the technologies used were already known to the author, a performance assessment is not something usual for him, which enabled the author to explore new technologies and methodologies on a deeper level.
Even though the thesis was challenging in the author's time and day-to-day activities, great satisfaction came from what was accomplished with such a meaningful technology as GraphQL Federation for the author's curiosity and knowledge.

Bibliography

Amazon (2022). *What is cloud computing? - Amazon Web Services*. url: https://aws.amazon.com/pt/what-is-cloud-computing/.

Apollo GraphQL (2022a). *Apollo Federation subgraph specification - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/subgraph-spec/.

– (May 2022b). *Gateway 2.X performance issues - Issue 1861 - apollographql/federation*. url: https://github.com/apollographql/federation/issues/1861.

– (Feb. 2022c). *GitHub - apollographql/apollo-tracing*. url: https://github.com/apollographql/apollo-tracing.

– (Dec. 2022d). *Introduction to Apollo Federation - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/.

– (2022e). *Introduction to Apollo Federation - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/#managed-federation.

– (Apr. 2022f). *Moving toward GraphQL consolidation - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/enterprise-guide/graphql-consolidation/#why-consolidate-your-graph.

– (2022g). *Query plans - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/query-plans.

– (2022h). *Server-side caching - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/performance/caching#setting-entity-cache-hints.

– (2022i). *Server-side caching - Apollo Server Docs*. url: https://www.apollographql.com/docs/apollo-server/performance/caching/#in-your-resolvers-dynamic.

– (2022j). *Subscriptions - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/react/data/subscriptions/.

– (2022k). *Why adopt GraphQL? - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/intro/benefits/.

– (May 2023a). *Error handling - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/apollo-server/data/errors/#omitting-or-including-stacktrace.

– (May 2023b). *Federated schemas - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/federated-types/overview/.

– (Mar. 2023c). *Federation-compatible subgraph implementations - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/federation/building-supergraphs/supported-subgraphs/.

– (Apr. 2023d). *The Rover CLI - Apollo GraphQL Docs*. url: https://www.apollographql.com/docs/rover/.

Azure, Microsoft (2022). *What is Azure—Microsoft Cloud Services | Microsoft Azure*. url: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/.

Baraki, Harun et al. (2018). "Architectural Patterns for Microservices: A Systematic Mapping Study". In: *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018* 2018-January, pp. 136–147. doi: 10.5220/0006701101360147. url: https://bia.unibz.

it / esploro / outputs / conferenceProceeding / Architectural - Patterns - for - Microservices-A-Systematic-Mapping-Study/991005773017601241.

Basili, Victor R, Gianluigi Caldiera, and H Dieter Rombach (1994). *The Goal Question Metric Approach*.

Bhayani, Arpit (2022). *BFF - Backend for Frontend - Pattern in Microservices | LinkedIn*. url: https://www.linkedin.com/pulse/bff-backend-frontend-pattern-microservices-arpit-bhayani/.

Blinowski, Grzegorz, Anna Ojdowska, and Adam Przybylek (2022). "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10, pp. 20357–20374. issn: 21693536. doi: 10.1109/ACCESS.2022.3152803.

Bremner, Jaemi (Mar. 2021). *GraphQL: Making Sense of Enterprise Microservices for the UI | by Jaemi Bremner*. url: https://blog.developer.adobe.com/graphql-making-sense-of-enterprise-microservices-for-the-ui-46fc8f5a5301.

Brito, Gleison and Marco Tulio Valente (Mar. 2020). "REST vs GraphQL: A controlled experiment". In: *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020*, pp. 81–91. doi: 10.1109/ICSA47634.2020.00016.

Facebook (2023). *GitHub - graphql/dataloader*. url: https://github.com/graphql/dataloader.

Fontão, João (June 2022). "GraphQL Federation Impact Analysis". In.

Fowler, Martin (Aug. 2019). *Microservices Guide*. url: https://martinfowler.com/microservices/.

– (Aug. 2021). *Gateway*. url: https://martinfowler.com/articles/gateway-pattern.html.

GitHub (May 2023). *Managing tags in GitHub Desktop - GitHub Docs*. url: https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/managing-commits/managing-tags-in-github-desktop#further-reading.

Github (May 2023). *GitHub*. url: https://github.com/.

Gos, Konrad and Wojciech Zabierowski (2020). "The Comparison of Microservice and Monolithic Architecture". In: *International Conference on Perspective Technologies and Methods in MEMS Design*, pp. 150–153. issn: 25735373. doi: 10.1109/MEMSTECH49584.2020.9109514.

Grafana (June 2023). *Grafana*. url: https://github.com/grafana/grafana.

GraphQL (2019a). *GraphQL - JSON Serialization*. url: http://spec.graphql.org/draft/#sec-Serialization-Format.

– (2019b). *GraphQL Spec*. url: http://spec.graphql.org/draft.

– (2020). *Execution*. url: https://graphql.org/learn/execution/.

– (2022a). *Caching | GraphQL*. url: https://graphql.org/learn/caching/.

– (2022b). *Global Object Identification | GraphQL*. url: https://graphql.org/learn/global-object-identification/.

– (2022c). *GraphQL Best Practices | GraphQL*. url: https://graphql.org/learn/best-practices/.

– (2022d). *Introspection | GraphQL*. url: https://graphql.org/learn/introspection/.

– (2022e). *Queries and Mutations | GraphQL*. url: https://graphql.org/learn/queries/.

– (2022f). *Schemas and Types | GraphQL*. url: https://graphql.org/learn/schema/.

– (2022g). *Serving over HTTP | GraphQL*. url: https://graphql.org/learn/serving-over-http/.

– (2023). *GitHub - graphql/graphql-over-http*. url: https://github.com/graphql/graphql-over-http.

Hampton, Jeff, Michael Watson, and Mandi Wise (2020). "GraphQL at Enterprise Scale A Principled Approach to Consolidating a Data Graph". In: url: https://www.apollographql.com/.

Helfer, Jonas (May 2021). *GraphQL explained - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/graphql/basics/graphql-explained/#query-execution-step-by-step.

ISO (2011). *ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. url: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en.

ISO 25000 (2011). *ISO 25010*. url: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

Isquick, David (July 2021). *Moving from Schema Stitching to Federation: How Expedia improved performance - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/announcement/expedia-improved-performance-by-moving-from-schema-stitching-to-apollo-federation/.

Java, GraphQL (2022). *Execution*. url: https://www.graphql-java.com/documentation/execution/.

– (Mar. 2023). *Getting started | GraphQL Java*. url: https://www.graphql-java.com/documentation/getting-started/.

Jesus França, Luciano Cavalcante de et al. (2020). "AHP Approach Applied To Multi-Criteria Decisions In Environmental Fragility Mapping". In: *Floresta* 50 (3), pp. 1623–1632. issn: 19824688. doi: 10.5380/RF.V50I3.65146.

JetBrains (Dec. 2022). *Code completion*. url: https://www.jetbrains.com/help/idea/auto-completing-code.html.

JMeter, Apache (June 2023). *Apache JMeter open-source load testing tool for analyzing and measuring the performance of a variety of services*. url: https://github.com/apache/jmeter.

Jones, Doc (Nov. 2022). *What Is GraphQL? Part 1: The Facebook Years | Postman Blog*. url: https://blog.postman.com/what-is-graphql-part-one-the-facebook-years/.

Kalske, Miika, Niko Mäkitalo, and Tommi Mikkonen (Feb. 2018). "Challenges When Moving from Monolith to Microservice Architecture". In: doi: 10.1007/978-3-319-74433-9_3.

Koen, Peter et al. (2001). "Providing Clarity and A Common Language to the "Fuzzy Front End". In: *Research-Technology Management* 44 (2), pp. 46–55. issn: 1930-0166. doi: 10.1080/08956308.2001.11671418. url: https://www.tandfonline.com/action/journalInformation?journalCode=urtm20.

Kruchten, Philippe B. (1995). "The 4+1 View Model of Architecture". In: *IEEE Software* 12 (6), pp. 42–50. issn: 07407459. doi: 10.1109/52.469759. url: https://www.researchgate.net/publication/220018231_The_41_View_Model_of_Architecture.

Kuc, Dariusz (May 2023a). *apollographql/federation-jvm-spring-example: Apollo Federation JVM example implementation using Spring for GraphQL*. url: https://github.com/apollographql/federation-jvm-spring-example.

– (Apr. 2023b). *JVM support for Apollo federation*. url: https://github.com/apollographql/federation-jvm.

Lauretis, Lorenzo De (Oct. 2019). "From monolithic architecture to microservices architecture". In: *Proceedings - 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops, ISSREW 2019*, pp. 93–96. doi: 10.1109/ISSREW.2019.00050.

Lewis, James and Martin Fowler (Mar. 2014). *Microservices - a definition of this new architectural term*. url: https://www.martinfowler.com/articles/microservices.html.

Manes, Ben (Apr. 2023). *ben-manes/caffeine: A high-performance caching library for Java*. url: https://github.com/ben-manes/caffeine.

Meta (Sept. 2015). *GraphQL: A data query language - Engineering at Meta*. url: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/.

Micrometer (June 2023). *Micrometer - GitHub*. url: https://github.com/micrometer-metrics/micrometer.

MicroServiceIO (2018). *Pattern: Monolithic Architecture*. url: https://microservices.io/patterns/monolithic.html.

MicroserviceIO (2018a). *API gateway pattern*. url: https://microservices.io/patterns/apigateway.html.

– (2018b). *Pattern: Database per service*. url: https://microservices.io/patterns/data/database-per-service.html.

Microsoft (2020). *Materialized View pattern - Azure Architecture Center*. url: https://learn.microsoft.com/en-us/azure/architecture/patterns/materialized-view.

Naman and Ida Bagus Vohra Kerthyayana Manuaba (2022). "Implementation of REST API vs GraphQL in Microservice Architecture". In: pp. 45–50. doi: 10.1109/ICIMTech55957.2022.9915098. url: https://ieeexplore.ieee.org/document/9915098.

Netflix (May 2023). *Getting Started - DGS Framework*. url: https://netflix.github.io/dgs/getting-started/.

Nicola, Susana (2022). *Análise de Valor*. url: https://moodle.isep.ipp.pt/mod/resource/view.php?id=120311.

Overflow, Stack (2022a). *Stack Overflow Insights*. url: https://insights.stackoverflow.com/survey.

– (2022b). *Stack Overflow Trends*. url: https://insights.stackoverflow.com/trends?tags=graphql.

Palacio, Bernardo (Nov. 2022). *Netflix/dgs-federation-example*. url: https://github.com/Netflix/dgs-federation-example.

Ponce, Francisco, Gaston Marquez, and Hernan Astudillo (Nov. 2019). "Migrating from monolithic architecture to microservices: A Rapid Review". In: *Proceedings - International Conference of the Chilean Computer Science Society, SCCC* 2019-November. issn: 15224902. doi: 10.1109/SCCC49216.2019.8966423.

prakarsh (Aug. 2020). *SpringBoot APM Dashboard | Grafana Labs*. url: https://grafana.com/grafana/dashboards/12900-springboot-apm-dashboard/.

Prometheus (Feb. 2023). *Prometheus Overview*. url: https://prometheus.io/docs/introduction/overview/.

Queirós, Leandro (June 2023). *federation-jvm-spring-tmdei: Apollo Federation JVM implementation for Performance Assessment*. url: https://github.com/G4LKK/federation-jvm-spring-tmdei.

Rashid, Aaqib and Amit Chaturvedi (Feb. 2019). "Cloud Computing Characteristics and Services A Brief Review". In: *International Journal of Computer Sciences and Engineering* 7 (2), pp. 421–426. doi: 10.26438/IJCSE/V7I2.421426.

Relay (2020a). *Relay - GraphQL*. url: https://relay.dev/.

– (2020b). *Relay - GraphQL Cursor Connections Specification*. url: https://relay.dev/graphql/connections.htm.

Richardson, Chris (Oct. 2018). *Microservices patterns with examples in Java*. Ed. by Simon and Schuster, pp. 1–520. isbn: 1638356327. url: https://books.google.com/books/about/Microservices_Patterns.html?hl=pt-PT&id=QTgzEAAAQBAJ.

Roksela, Piotr, Marek Konieczny, and Slawomir Zielinski (July 2020). "Evaluating execution strategies of GraphQL queries". In: *2020 43rd International Conference on Telecommunications and Signal Processing, TSP 2020*, pp. 640–644. doi: 10.1109/TSP49548.2020.9163501.

Saaty, Thomas L. (Sept. 1990). "How to make a decision: The analytic hierarchy process". In: *European Journal of Operational Research* 48 (1), pp. 9–26. issn: 0377-2217. doi: 10.1016/0377-2217(90)90057-I.

Sampaio, Adalberto R. et al. (Sept. 2017). "Supporting Microservice Evolution". In: IEEE, pp. 539–543. isbn: 978-1-5386-0992-7. doi: 10.1109/ICSME.2017.63. url: http://ieeexplore.ieee.org/document/8094458/.

Schmidt, Geoff and Matt DeBergalis (2022). *Agility Principles - Principled GraphQL*. url: https://principledgraphql.com/agility.

Schrade, Kyle (Nov. 2020). *9 Lessons From a Year of Apollo Federation - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/backend/federation/9-lessons-from-a-year-of-apollo-federation/.

– (May 2021). *Caching Strategies in a Federated GraphQL Architecture - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/backend/federation/caching-strategies-in-a-federated-graphql-architecture/.

Shikhare, Tejas (Nov. 2020). *How Netflix Scales its API with GraphQL Federation*. url: https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2.

Solingen, Rini van. and Egon. Berghout (Jan. 1999). "The goal/question/metric method : a practical guide for quality improvement of software development". In: p. 199.

Spring (May 2023a). *Configuring the Cache Storage - Spring Framework*. url: https://docs.spring.io/spring-framework/reference/integration/cache/store-configuration.html#cache-store-configuration-caffeine.

– (May 2023b). *Spring Boot - Metrics*. url: https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html.

– (May 2023c). *Spring Boot Actuator - GitHub*. url: https://github.com/spring-projects/spring-boot/tree/v3.1.0/spring-boot-project/spring-boot-actuator.

– (May 2023d). *Spring for GraphQL Documentation*. url: https://docs.spring.io/spring-graphql/docs/current/reference/html/#execution.graphqlsource.operation-caching.

– (May 2023e). *Spring for GraphQL Documentation - Batching*. url: https://docs.spring.io/spring-graphql/docs/current/reference/html/.

State of GraphQL (2022). *The State of GraphQL 2022*. url: https://2022.stateofgraphql.com/en-US/.

Streams, Reactive (2022). *Reactive Streams*. url: http://www.reactive-streams.org/.

Stubailo, Sashko (Sept. 2017). *GraphQL schema stitching - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/backend/graphql-schema-stitching/.

Stünkel, Patrick et al. (July 2020). "GraphQL Federation: A Model-Based Approach". In: *21* 19 (2). issn: 1660-1769. doi: 10.5381/JOT.2020.19.2.A18. url: https://hvlopen.brage.unit.no/hvlopen-xmlui/handle/11250/2719130.

Trends, Google (2022). *GraphQL - Explore - Google Trends*. url: %5Curl%7Bhttps://trends.google.com/trends/explore?date=2020-12-31%5C%202023-01-24&q=GraphQL%7D.

Valdivia, J. A. et al. (Dec. 2020). "Patterns Related to Microservice Architecture: a Multivocal Literature Review". In: *Programming and Computer Software* 46 (8), pp. 594–608.

issn: 16083261. doi: 10.1134/S0361768820080253/TABLES/3. url: https://link.springer.com/article/10.1134/S0361768820080253.

Walraven, Martijn (Oct. 2017). *Exposing trace data for your GraphQL server with Apollo Tracing - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/announcement/platform/exposing-trace-data-for-your-graphql-server-with-apollo-tracing/.

Zeaaraoui, Adil et al. (2013). "User stories template for object-oriented applications". In: *2013 3rd International Conference on Innovative Computing Technology, INTECH 2013*, pp. 407–410. doi: 10.1109/INTECH.2013.6653681.