ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

gRPC and Protobuf: Performance and API Flexibility

Luís Miguel Gonçalves Silva

MSc in Computer Engineering,

Specialisation Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, 08 July 2024

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, Porto, 08 July 2024

Luis Miguel Gonçalves Silva

Abstract

Software systems are constantly evolving, and traditional approaches such as REST API and GraphQL have been, and still are, crucial for implementing public-facing interfaces. However, gRPC allows for greater efficiency in managing multiple requests and responses through a single connection, enhancing application performance.

Technology has followed a similar trajectory, leading to challenges in traditional implementations due to the complexity of systems and diverse needs. These challenges have prompted the implementation of methods to ensure that results only contain relevant information for a given system, thereby eliminating unnecessary data. The Netflix example illustrates and implements a solution that adapts responses using gRPC and Protobuf field mask [1].

With the increasing need to make systems responsive to various devices, there is a growing inclination to use standards that offer greater flexibility and modularity in development. Protobuf, which is agnostic, allows the production of a foundation across different languages without modification requirements.

The main advantage highlighted in using Protobuf is performance, while the main disadvantage is its diminutive ecosystem. Subsequently, samples show how it is possible to eliminate irrelevant attributes in both REST with GraphQL and gRPC using the Protobuf field mask. These examples demonstrate that the latter option is more accessible and flexible. After, a systematic review is conducted using the PRISMA method to define a flexible API, identify software patterns that allow better flexibility when providing and using applications, and understand API performance and its significance.

A migration is accomplished for GraphQL and gRPC, which is performance-tested later using k6 scripts. We execute the scripts, and the metrics are subsequently collected and compared. A flexibility test is achieved by adding, changing, and removing previously implemented methods and checking the effects of the modifications.

The results show that, on average, gRPC is quicker than GraphQL, although latency peaks occur. These spikes are significantly reduced when using streams. As for the flexibility test, gRPC is more flexible due to having a well-established contract with other APIs. These contracts are then easily adapted when using field masks through their field ID.

Keywords: Flexible API, gRPC, Performance, Patterns, Migration

Table of Contents

List of Figures

List of Tables

List of Source Code Extracts

List of abbreviations

ACM	Association for Computing Machinery
AHP	Analytic Hierarchy Process
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSV	Comma-separated values
DDoS	Distributed Denial-of-Service attack
EC	Exclusion Criteria
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IC	Inclusion Criteria
ID	Identifier
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IPP	Instituto Politécnico do Porto
JSON	JavaScript Object Notation
MIT	Massachusetts Institute of Technology
ms	Milliseconds, 1s = 1000ms
MVC	Model-View-Controller Pattern
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
PROTO	Is the extension for the file of the Protocol Buffers
RDMA	Remote Direct Memory Access

REST	Representational State Transfer
RFP	Request for Proposal
RPC	Remote Procedure Call
SaaS	Software as a service
TCP	Transmission Control Protocol
THRIFT	Is the extension for Thrift files, similar to proto files
TLS	Transport Layer Security
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

This chapter includes a description of the context of the problem, the project's goals identification, the research methodology enumeration, a list of ethical considerations, and the document structure description.

1.1 Context and Problem

With the ever-evolving requirements of each new device, the need for flexible APIs is on the rise, leading to the need to request what is necessary without the unnecessary data attached.

Traditional approaches like REST API and GraphQL have widespread usage due to their inherent flexibility, and therefore, diverse industries use them. gRPC stands out by efficiently managing multiple requests and responses over a single connection, enhancing application performance and error handling.

Figure 1 - Interest in gRPC over the years [2]

gRPC is a modern open-source, high-performance Remote Procedure Call (RPC) framework. It reached 1.0 in 2016. It supports load balancing, tracing, health checking, and authentication [3][4][5].

As technology evolves, the increasing complexity of systems and the diverse requirements of each application version fall behind the necessity of refining data results. This need has amassed

growing attention, such as Netflix's example efforts to improve gRPC API responses using Protobuf field masks [1]. The escalating necessity of eliminating unnecessary fields aims to enhance API flexibility and efficiency. This upgrade is accomplished by harnessing newer technologies such as gRPC and using patterns that improve the system's maintainability and flexibility.

With the need to reduce over-fetching and the rise of gRPC in popularity, it is necessary to explore to what extent migration from a traditional technology stack such as REST and GraphQL to a gRPC and Protobuf-based one affects performance and flexibility. Both enhance the software development process by refining such metrics.

1.2 Goals

This work's main objective is to assess if using gRPC and Protobuf field masks causes an improvement in performance and API flexibility.

Due to time constraints, only the performance and flexibility improvements of gRPC are explored. Other metrics could be investigated as part of future work.

The research questions for this work, as previously stated, revolve around metrics and are the following:

- RQ1: How do gRPC and Protobuf field masks affect API performance?
- RQ2: How do gRPC and Protobuf field masks affect API flexibility?

To fully understand the research questions, we selected some relevant studies. Firstly, study what API Flexibility means. Secondly, some exemplifying patterns to improve it. And thirdly, the meaning of API performance.

The research methodology addressed these questions by conducting a controlled experiment. An open source was chosen to assess and compare the metrics it set out to analyse.

Priority to an already established application was given due to time constraints. This way, the execution of the experiments is in an application developed in C#.

The project selected is then transitioned from the original technology of REST and GraphQL to a gRPC-based architecture employing Protobuf. Subsequently, a comparison of the performance and flexibility is performed, after which a conclusion is drawn. A Gantt diagram was developed to organize the tasks and is represented by Figure 15.

1.3 Ethical Considerations

Within the ethical landscape of this work, as a student, the IPP code of conduct was used [6]. The following conducts were respected during this project: article 6th statement 2.8 mentions the usage of ideas, sentences, paragraphs, or texts without citing or not correctly referencing the sources [6]. 2.9, to present as an original work that has already been exhibited or published on another occasion without giving explicit knowledge. 2.10 says it presents work carried out with another, resulting from authorized non-compliance (**Error! Reference source not found.**, REF _Ref170294004 \h **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.** and **Error! Reference source not found.**), and 2.11, the usage of papers that have fake or tendentiously interpreted results.

Article 8 is present at the beginning of the document, which is the declaration of commitment. This commitment asserts the integrity of the academic work, emphasizing its originality and faithfulness to ethical standards, and finally, a commitment to the Code of Ethical Conduct of P. PORTO.

10th article was followed in points such as 1 a) analyse and document an investigation in a careful and weighted, e) appropriately and comprehensively citing related works, and g) consistent manner of presenting results, ensuring that the findings are verifiable and reproducible.

As a student of a master's in informatics engineering, the branch of software engineering, the code of conduct of IEEE is also relevant [7]. Some principles of the IEEE, such as honesty, integrity, upholding transparency, truthfulness throughout this paper, and professional competence, ensure that the work is of the highest quality. Refraining from conflicts of interest, such as potential biases or personal interests, could compromise the integrity of the decisions.

Other principles such as "ensuring proper and achievable goals and objectives" and "Identifying, defining and addressing ethical, economic, cultural, legal and environmental issues related to work" were addressed by discussing with the advisor to define realistic targets and to use open-source projects or of free utilization.

1.4 Document Structure

This paper features six chapters: Introduction, Background, State of the art, Application Migration, Tests and Solution Evaluation, and Conclusions.

This first chapter introduces the body of work and the project's objectives and scope.

The second chapter details some necessary information on how gRPC works and how to remove unnecessary parameters from request queries from a gRPC and a REST API standpoint.

The third chapter contains three systematic reviews: what a flexible API is, what patterns can improve the flexibility of an API, and the meaning of API performance.

The fourth chapter shines a light on the basic concepts of the base project and shows how to perform the migration for both GraphQL and gRPC.

The 5th chapter shows the developed testing mechanisms using k6 scripts, flexibility testing, the attained results from the performance tests, grading the respective application's flexibility and the statistical analysis of the performance outcomes.

For the 6th chapter, the conclusion section draws a verdict from the previously attained results, presents a list of threats to the validity of the work and lists some opportunities for future work.

Following are the references and the appendix. These sources are cited throughout the main body of work, and the attachments complement the knowledge already presented.

2 Background

This chapter is an introductory section to technologies relevant to the development of this project. Such tools as Hot Chocolate, gRPC, and others are present in Appendix C. It is also explored how gRPC and Protobuf work together to provide a working application. Subsequently, an analysis of the benefits and drawbacks of using such technology, and to finalise the chapter, examples are provided on how over-fetching in both REST with GraphQL and gRPC with Protobuf could be solved.

2.1 gRPC

gRPC, Google Remote Procedure Call, is a technology that allows a client application to call the implemented method directly using Protobuf. The server and the client applications have their proto definitions locally accessible, allowing the server to implement its interface and handle gRPC calls. As for the client, those proto files allow to call a stub, which supplies the same method as the server [8]. This is a well-established contract.

gRPC makes use of HTTP/2.0. It brings many enhancements, but the capability to server push, which is the ability to pre-emptively send packets before the client asks for them and multiplexing of requests on a single TCP connection are the most relevant. These capacities further lower the latency of requests, thus improving the performance of the systems employing such technology [9].

Protobuf is a Google implementation of the language for serializing structured data. It offers several advantages over JSON, such as reduced message size and enhanced processing speed. Because it is generated at runtime, support for multiple programming languages is guaranteed.

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

Code 1 - Example of Protobuf definition of Person [10]

The example of Code 1 is the definition of a Person object in Protobuf. After building the project, a *.proto* file is generated in the project's language. This file is then used to perform operations such as setting the name or getting the email of the person [11].

Code 2 shows a possible definition for a gRPC interface in a proto file. Where the previously mentioned server implements the actual logic behind the "grpcPersonService". The first method, "GetPerson", can be called by sending an ID and a field mask to specify which parameters to retrieve from the Person message. "Get Transactions", "Get Transactions Client

Stream", "Get Transactions Server Stream", and "Get Transactions Two Sided Stream" are some examples of how we specified streaming in Protobuf.

```
service grpcPersonService {
    rpc GetPerson (GetPersonRequest) returns (Person);
    rpc ChangeEmail (ChangeEmailRequest) returns (ChangeEmailResponse);
    rpc ChangeName (ChangeNameRequest) returns (ChangeNameResponse);
    rpc       GetTransactions       (GetTransactionsRequest)       returns
(GetTransactionsResponse);
    rpc GetTransactionsClientStream (stream GetTransactionsRequest) returns
(GetTransactionsResponse);
    rpc GetTransactionsServerStream (GetTransactionsRequest) returns (stream
GetTransactionsResponse);
    rpc   GetTransactionsTwoSidedStream   (stream   GetTransactionsRequest)
returns (stream GetTransactionsResponse);
}
```

Code 2 - grpcPersonService Example Methods Definition

```
message GetPersonRequest {
    optional int32 id = 1;
    google.protobuf.FieldMask fieldMask = 2;
}

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}

message ChangeEmailRequest {
  optional int32 person_id = 1;
  optional string new_email = 2;
}

message ChangeEmailResponse {
  optional bool success = 1;
}

message ChangeNameRequest {
  optional int32 person_id = 1;
  optional string new_name = 2;
}

message ChangeNameResponse {
  optional bool success = 1;
}

message GetTransactionsRequest {
  optional int32 person_id = 1;
}

message GetTransactionsResponse {
  repeated Transaction transactions = 1;
}

message Transaction {
  optional string transaction_id = 1;
  optional string description = 2;
```

```
  optional double amount = 3;
  optional google.protobuf.Timestamp timestamp = 4;
}
```

Code 3 - Additional Messages Representation

Code 3 is the remaining design of the message types used in the gRPC service definition. Code 1, Code 2 and Code 3 are also present in the client project to create instances of messages and perform stub calls to the gRPC service methods.

gRPC uses Protobuf as its primary communication platform, supporting JSON, Thrift, and XML. They all have varying levels of maturity. Thus, Protobuf is the safer option when building an application from scratch, as it provides more support [12].

2.1.1 Motivational Factors

Many companies changed their infrastructure and code base and implemented gRPC:

- Square migrated from an old implementation of RPC to the new gRPC. They say the main reasons to migrate were the gRPC being open source and the performance improvements compared to their in-house RPC solution [13].
- Cisco mentions that with the main benefits such as bi-directional streaming, TLS security and many supported programming languages. gRPC is the optimal unified transport protocol for configuration and telemetry driven by models [13].
- LinkedIn changed to gRPC after trying to reduce latency by using Protobuf in rest.li [14]. After successful results where they achieved 60% reduction, it was announced that they would migrate to gRPC due to “better performance, multiple language support and streaming” [14][15]. They were able to migrate 2000 services successfully from rest.li to gRPC with relative ease [16]. Streaming support was the main reason behind the switch.

With the earlier testimonials, we can deduce that many of the reasons to migrate from the old technology used were mostly related to improvements in performance. Improvements are either software or human. Those improvements are possible by the usage of HTTP/2.0 and the support of many programming languages. gRPC also has security measures that Cisco mentions, “TLS-based security”, as one of the reasons to use this technology.

2.1.2 Benefits & Downsides

The main benefits of using gRPC are its performance, latency focused APIs [17] and other previously mentioned upsides in sections 2.1 and 2.1.1). As for its downsides, a few were found, such as having a reduced ecosystem compared to the more mainstream REST API, which means there is not much support for web browsers. Another drawback is that the contract between APIs can change, meaning that the objects used as input or output of the endpoint can be

changed. Thus, there is a need to frequently adapt the proto file to both the server and the client [18]. Geewax et al. mention that the Protobuf field mask does not work well when "requesting specific information based on a relationship between the different resources" [19].

2.2 Optimizing Request Fields in REST and gRPC Apps

The focus of this section is to demonstrate how to optimise request fields within the response payload for both REST and gRPC APIs. The improvement is achieved by providing examples relevant to each API by illustrating how to remove unnecessary request fields.

2.2.1 REST API with GraphQL

For GraphQL, request field elimination is possible using directives [20]. It can, for instance, be added to the response payload with @include or @skip to remove.

```
Type Student {
 id: ID
 name: String
}

type Query {
 students: [Student]
}

query allUsers($includeId: Boolean) {
 Student {
  id @include(if: $includeId)
  name
 }
}
```

Code 4 - Student definition and Student Query definition

For this schema, we define the $includeId, a Boolean that indicates if the ID is to be contained in the payload response. The return example of the query is two Students.

```
new Student(1, "John Doe"),
new Student(2, "Jane Smith")
```

Code 5 - Student instances

If the includeId boolean value is true, then the return would be:

```
{
  "data": {
    "students": [
      {
        "id": 1,
        "name": "John Doe"
      },
```

```
        {
          "id": 2,
          "name": "Jane Smith"
        }
      ]
    }
}
```

Code 6 - JSON payload example with all parameters

As for when the ID is not to be included, the boolean being false, the resulting payload would be:

```
{
  "data": {
    "students": [
      {
        "name": "John Doe"
      },
      {
        "name": "Jane Smith"
      }
    ]
  }
}
```

Code 7 - JSON payload example with name parameter

For more complex situations, the query would have more parameters, and the student payload would require more constraints on what should be returned.

2.2.2 gRPC API

As for gRPC, employing field masks makes the implementation more adaptable as each client decides what is needed. The following example provides a view of how to implement field masks.

```
message Student {
  int32 id = 1;
  string name = 2;
}

message allUsers {
  repeated string field_mask = 1;
}
```

Code 8 - Student and field mask Protobuf definition

As previously mentioned, using field masks allows for more versatility as the field_mask parameter is defined in the message allUsers, which can specify which values to receive without the need to change the query. For instance, if we set the following as field_mask:

```
{
```

```
  "field_mask": ["id", "name"]
}
```

Code 9 - Example of field mask usage with ID and name

Then, the payload would result in all the users with all their respective information. But if we set the field_mask as:

```
{
  "field_mask": ["id"]
}
```

Code 10 - Example of field mask usage with ID

Then, all the users would only contain their IDs, resulting in the payload:

```
{
  "students": [
    {
      "id": 1
    },
    {
      "id": 2
    }
  ]
}
```

Code 11 - JSON payload example with ID parameter

The same could be performed on the student's name by simply replacing the ID with the name. The resulting payload would only contain the list of students and their names. It is also possible to use the field number instead of the field name [1].

3 State of the Art

In this chapter, three systematic reviews are carried out using the PRISMA method, which is introduced later. The first review goal is to define API flexibility. The second systematically reviews literacy to list patterns that mitigate or improve flexibility. The third and final review assesses the meaning of API performance. After which, a comparison of GraphQL, and gRPC and Protobuf comparison is made.

3.1 PRISMA Method

PRISMA, Preferred Reporting Items for Systematic Reviews and Meta-Analyses, initially appeared in 1997 and has gone through many iterations to arrive at its most recent 2020 version [21], [22], [22]. This method starts with defining the main topic and the research libraries. Following the library selection, the research question(s) are built. After that, the keyword extraction from the question(s) is executed, and these words are then used along with synonyms to construct a research query. This query is then applied in the digital library alongside some limiting factors such as the document's written language, its type, or the publication date, among other characteristics. This query request returns a list of the found documents. With the catalogue of papers available, the screening phase begins. It starts with analysing each document title and abstract and determining if it has similarities to the research question(s). The ones that do not relate to the goal are eliminated from the list, as are the inaccessible ones. When all the papers are analysed, the work can begin to answer the question(s), but it is possible to exclude documents from the list by not containing any suitable or relevant information.

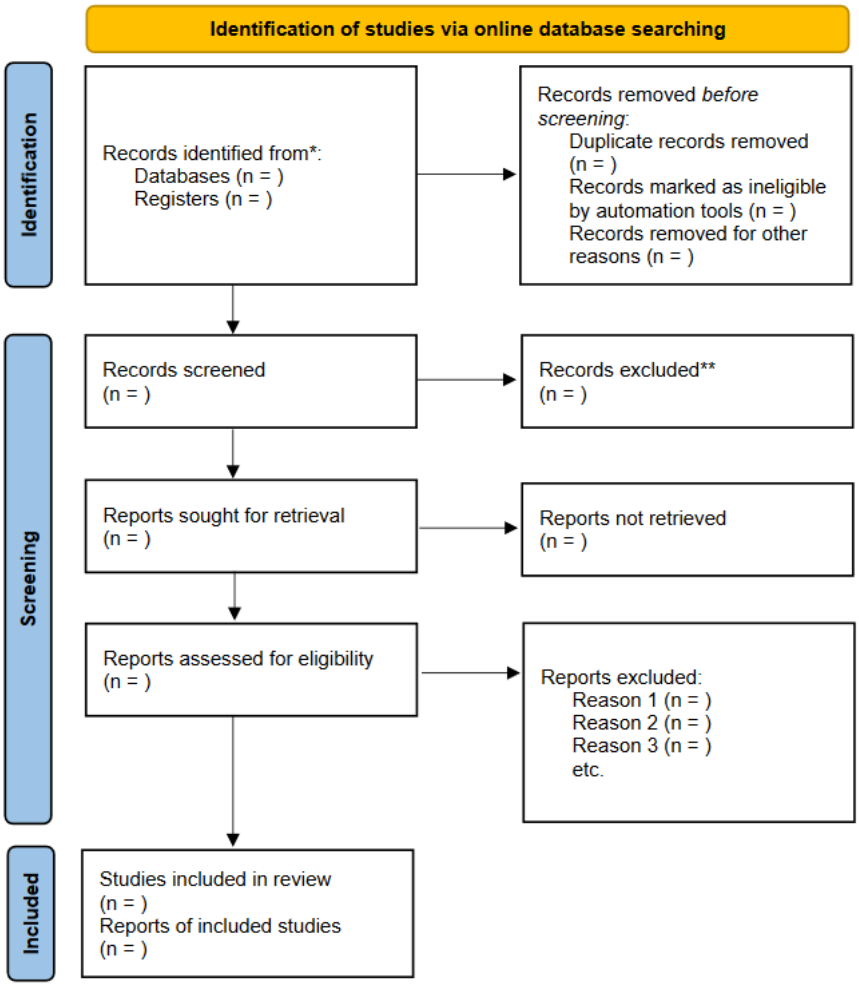

Figure 2 - PRISMA Method Diagram [22]

For this state-of-the-art, three questions were found relevant:

- “What is a flexible API?”
- “Which patterns improve flexibility on an API?”
- “What is API Performance?”

In sections 3.2, 3.3 and 3.4, the employment of PRISMA for the questions is explained in more detail.

3.2 Flexible API

This systematic review has the objective of defining what a flexible API is.

For data sources, it was used the b-on digital library, which is where the articles can be found.

To retrieve the number of studies related to “Flexible API”, the domain of “Flexible API” was established. Following, the keywords were defined, which are synonyms of the word “Definition” were attained from [23]. Table 1 lists these keywords.

Table 1 - Flexible API Domain and Keywords

Domain	Keywords
Flexible API	Definition, Explanation, Clarification, Meaning

This process was later used in the construction of the query which led to the following:

AB (Flexible and API) AND AB (definition or explanation or clarification or meaning)

Next, the definition of the inclusion and exclusion criteria for the search was done. They are vital as they help rule out the studies that do not meet the rules. These criteria revolve around the content of the papers. Table 2 and Table 3 list the defined criteria.

Table 2 - Flexible API Inclusion Criteria

Inclusion Criteria Number	Description
IC1	The paper must have been published between 2018 and 2023, i.e., it should not be more than 5 years old from the date of publication.
IC2	The paper should be available in PDF format.
IC3	The paper should be written in English.
IC4	Should be inserted in informatics, technology or information of technology disciplines

Table 3 - Flexible API Exclusion Criteria

Exclusion Criteria Number	Description
EC1	The paper is more than 5 years old (published before 2018).
EC2	The paper is not available in PDF format.
EC3	The paper isn’t written in English.
EC4	Paper is not included in the disciplines
EC5	The paper does not address topics that are relevant to the research question.

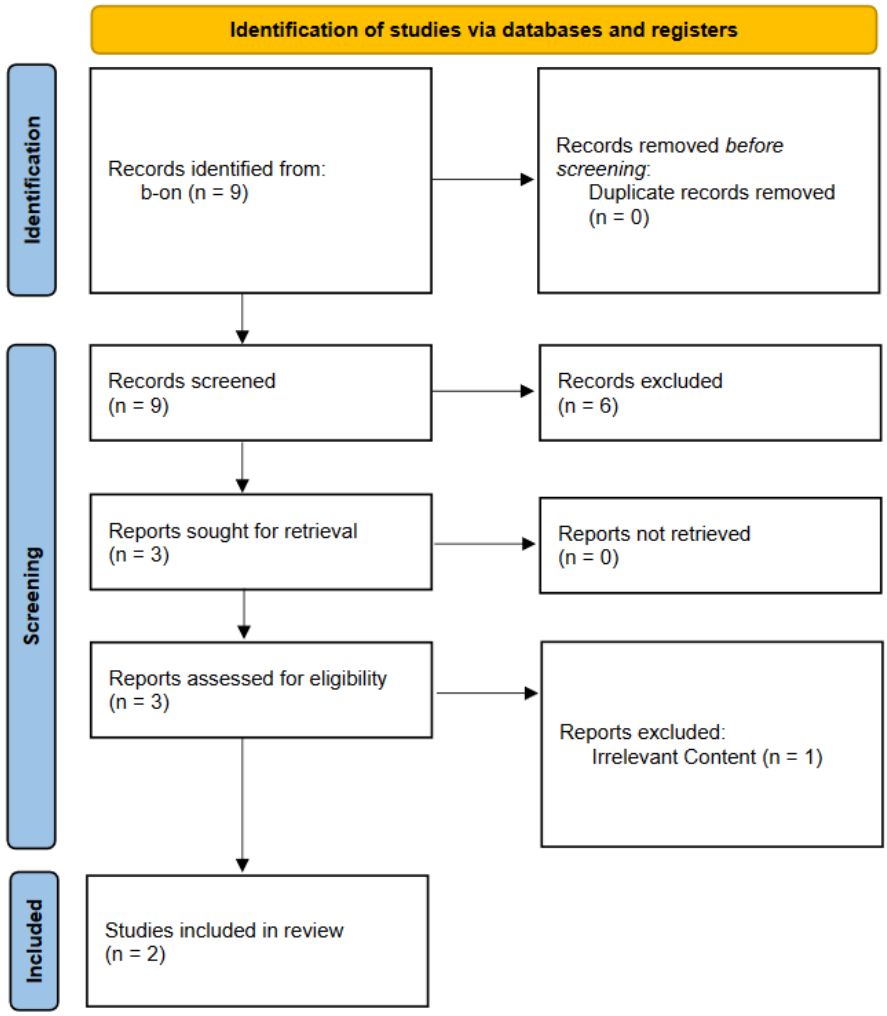

Figure 3 - PRISMA Diagram of Flexible API Systematic Review

With all preparations completed, the query was used on the b-on digital library. Nine papers resulted in the search, where zero encountered duplicate records. Subsequently, the title and abstract of the collected nine papers were analysed, which led to six articles being excluded and considered irrelevant to the research question. Following this, an additional examination was carried out on the remaining three papers, which were all accessible. Upon reading the three papers, one did not align with the goal of the research question. This process led to the inclusion of the remaining two documents in the systematic review.

In terms of API flexibility, an API can easily be changed to accommodate new use cases without making considerable changes to the code base. “the ability to change or be changed easily” [24].

Inflexibility results in APIs that cannot allow for straightforward changes without the considerable substitution of a part or, in some cases, the whole source of the project. This causes the issue of remaking software that has completed its purpose. This, in turn, can, as previously mentioned, influence external users to either adapt their solutions or use a different provider altogether [25].

Mukhiya et al. mention that using “REST rules and constraints helps deliver functioning APIs and access to resources”, but the application and usage of the previously mentioned rules cause the APIs to become inflexible [26]. One of the pointed-out reasons for this inflexibility is the need to dynamically fetch only the necessary data for the requirements of web or mobile apps. It also mentions that REST APIs have a query complexity issue where “complex and multiple requests are needed to fetch various resources”. Others like over-fetching, returning too much data, under-fetching, returning too little data, and the n+1 request complexity issue, the need to perform additional queries for the required information, are some other issues with REST APIs. GraphQL tries to solve this by allowing users to request exactly what they want with three different operations (queries, mutations, and subscriptions).

The “Podman in Action” book says the following, “more flexible by allowing users to supply their content for the web service”, meaning that allowing end users to supply their content allows for greater flexibility [27].

In summary, a Flexible API is an API that allows for change without too much complexity and issues caused by changes to the base code. The most flexible API possible is one that allows the end user to request only what is necessary without any changes. For instance, if using REST API and a user requests a new functionality, which requires the return of two data objects associated with each other, then a new endpoint must be developed to fulfil that request. On the other hand, if using GraphQL, no new endpoint is needed as the user only needs to create a new query to attain such information.

3.3 Patterns for better API flexibility

This review lists some possible patterns that can be applied when developing APIs that enhance flexibility and thus improve the development process.

It was selected the ACM Digital Library for the data sources, which is the web page where the articles are gathered.

To retrieve the number of studies related to “API Flexibility”, the domain of “Flexibility” was first established. Subsequently, some keywords synonyms of “Flexibility” were attained from [23]. These were defined and listed in Table 4.

Table 4 - API Patterns Domain and Keywords

Domain	**Keywords**
Flexibility	Flexibility, Adaptability, Versatility

This process was later used in the construction of the query, which led to the following:

("API patterns" AND ("flexibility" OR "Adaptability" OR "Versatility")) OR ("API design patterns" AND ("flexibility" OR "Adaptability" OR "Versatility")

Next, the inclusion and exclusion criteria were defined. They are essential as it helps to rule out the studies that do not fit. These criteria revolve around the content of the papers. Table 5 and Table 6 list the described criteria.

Table 5 - API Patterns Inclusion Criteria

Inclusion Criteria Number	**Description**
IC1	The paper must have been published between 2018 and 2023, i.e., it should not be more than 5 years old from the date of publication.
IC2	The paper should be available in PDF format.
IC3	The paper should be written in English.

Table 6 - API Patterns Exclusion Criteria

Exclusion Criteria Number	**Description**
EC1	The paper is more than 5 years old (published before 2018).
EC2	The paper is not available in PDF format.
EC3	The paper is not written in English.
EC4	The paper does not address topics that are relevant to the research question.

With everything ready, the query was applied alongside the criteria in Table 5 and Table 6 to the ACM Digital Library. A total of 16 papers resulted from the search. Since only one database was used, no duplicate records were found. Subsequently, an analysis of the title and abstract was conducted on the 16 papers. Five of which, deemed irrelevant to the research question, were excluded. Afterwards, a more in-depth examination was performed on the remaining 11. Five documents that weren't accessible were removed from the list. Upon reading the remaining six papers, three were considered not pertinent to the goal of the question. This resulted in the three remaining documents to be included in the systematic review.

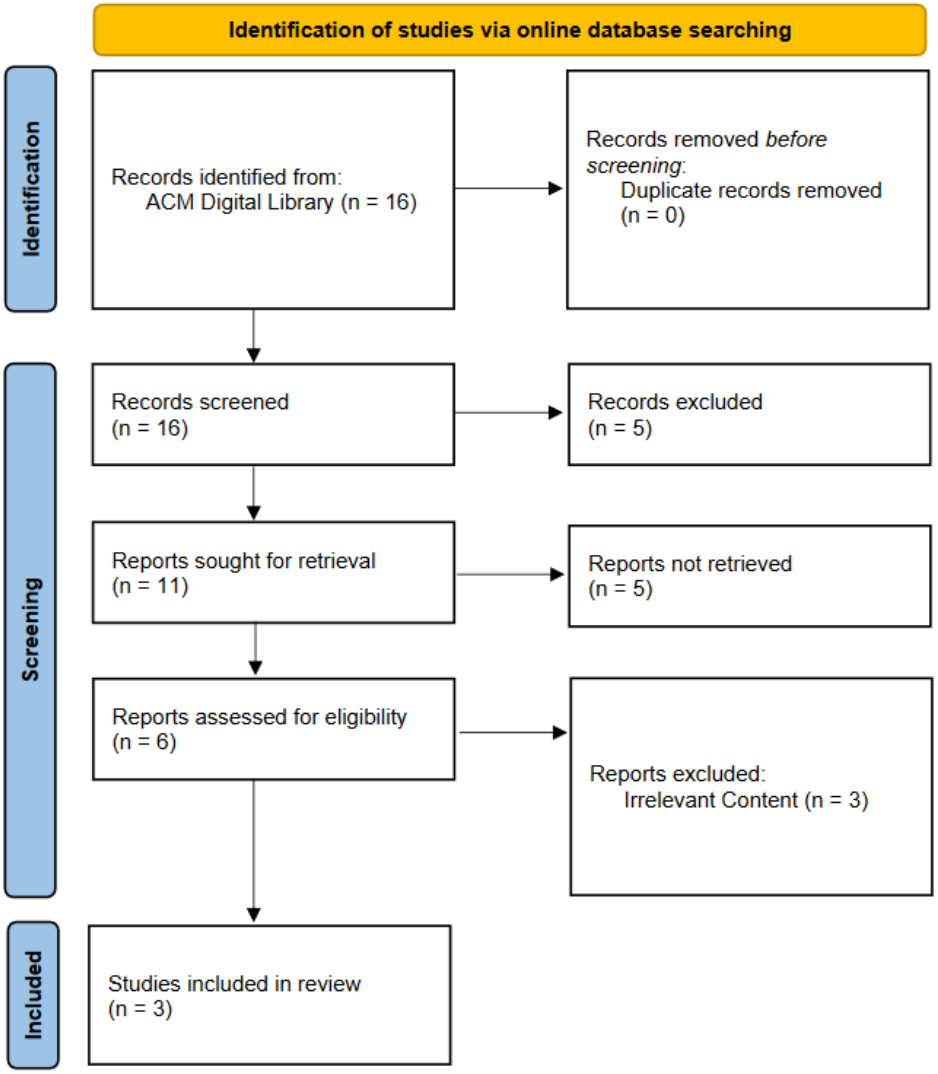

Figure 4 - PRISMA Diagram of API Patterns' Systematic Review

Most challenges in designing a good API arise from bad software practices. These poor executions cause the need to refactor software, which in turn causes other APIs to update their code. As mentioned in [25], other challenges derive from choosing the correct levels of abstraction, meaning much detail and complexity are exposed to the users and the need to evolve and maintain the software. Other challenges that should be accounted for are, for instance, the rate-limiting techniques, to not be abused by third parties, performance, scalability, and versioning. These are some of the many difficulties of designing an API.

Here, it is mentioned and explained the possible patterns that can be used to improve flexibility, either from an application architecture or a more generalized standpoint. These results pertain from the systematic review mentioned in the previous section 3.3.

Lübke et al. mention and define the following patterns [28]:

Table 7 - API Flexibility Patterns Part 1

Pattern Name	**Description**
API Description	Defines and documents technical and semantic aspects of an API for both machines and clients.
Version Identifier	Introduces explicit versioning into the messages, indicating the version of the accessed API, e.g., return 2.0.0 when accessing information.
Semantic Versioning	This versioning uses a hierarchical scheme to denote changes and compatibility levels. Changes from 1.3.0 to 1.3.5 might denote minor changes, while 1.0.0 to 2.0.0 indicates critical changes.
Two in Production	Deploys and maintains two API versions simultaneously, offering different revisions of identical features and allowing for continuous releases and deprecations.
Limited Lifetime Guarantee	Assures the stability of the published API without introducing changes for a specified fixed period.
Eternal Lifetime Guarantee	Promises uninterrupted access without disruptions or deprecation of a published API version.
Aggressive Obsolescence	Declares deprecation dates for outdated API endpoints, operations, or object representations as early as possible.
Experimental Preview Offer	Provides API access without commitments regarding functionality, stability, and longevity.

Subsequently [29] lists the patterns in Table 8:

Table 8 - API Flexibility Patterns Part 2

Pattern Name	**Description**
Processing Resource API endpoint	Allows clients to trigger actions on the provider side, exposing server-side operations with different responsibilities and effects, such as an MVC controller in spring [30].
Computation Function	Requests compute results solely from client input, creating functions for easy testing and reuse. For example, the validateCustomerRecord present in the document determines if a customer payload is valid.
State Creation Operation	Action that creates states on an API endpoint. That being write-only, receiving notifications or data from clients and storing them for later processing. Such as saving a customer to the database.
Retrieval Operation	For retrieving information owned or controlled by the provider. It is read-only and does not change server-side state, often offering search, filter, and formatting capabilities, for instance, retrieving a roster of current customers stored in the database younger than 25.
State Transition Operation	Action that causes a server-side state change, combining client input and current state to trigger a provider-side state transition. Such as updating the customer with the latest information.

Finally, [31] references the ensuing patterns:

Table 9 - API Flexibility Patterns Part 3

Pattern Name	**Description**
Information Holder Resource	Expose data classes in an API with CRUD and search operations. This is a generic pattern. Below are listed the more specialized patterns that derive from this one.
Operational Data Holder	Short-living, operational data that changes often and has many outgoing relations. This pattern can be a SaaS.
Master Data Holder	Long-living, frequently referenced, but mutable data. This would be an API that provides the used information.
Reference Data Holder	Long-living, immutable data referenced in many places. This can be a service that is designed only to look up information.
Data Transfer Resource	Shares data between communication participants without direct coupling. This pattern is a standalone application, an intermediary in communication, that stores the object for many clients.
Link Lookup Resource	A pattern that allows clients to fetch URLs to other resources without binding to their actual addresses. This is another intermediary that stores the URL to the provider.

From the patterns shown in Table 7, Table 8 and Table 9, we can conclude that patterns exist that help improve flexibility or maintainability. These patterns provide many possibilities when designing a flexible API or system(s). It is critical to be thoughtful in the design as the patterns might not be necessary in their implementation.

Some examples of improving the flexibility of an API without the usage of GraphQL or Protobuf field masks are the usage of versioning and two in-production patterns that allow for two different instances of the same API. Every different version can provide different resulting payloads or information [28]. Other examples mentioned in [29], such as the Data Transfer Resource and Link Lookup Resource, lower coupling and increase cohesion by only having one API to interact with. It also reduces maintainability and improves flexibility as it reduces dependency on an ever-changing API.

3.4 API Performance

This systematic review has the goal of defining what performance in API is.

The b-on digital library data source was used.

To retrieve the number of studies related to the theme of “API Performance”, the “API Performance” domain was established. After this, a complication of keywords was created. These consist of synonyms of “Definition” collected from [23]. Table 10 lists these synonyms.

Table 10 - API Performance Domain and Keywords

Domain	Keywords
API Performance	Definition, Explanation, Clarification, Meaning

This process was later used in the construction of the query, which led to the following:

AB (API and Performance) AND AB (definition or explanation or clarification or meaning)

Next, the inclusion and exclusion criteria were for the search defined. They are crucial as they help rule out the studies that meet the rules. These criteria revolve around the content of the papers. The depicted criteria are listed in Table 11 and Table 12.

Table 11 - API Performance Inclusion Criteria

Inclusion Criteria Number	Description
IC1	The paper must have been published between 2018 and 2023, i.e., it should not be more than 5 years old from the date of publication.
IC2	The paper should be available in PDF format.
IC3	The paper should be written in English.
IC4	The paper is an Academic Journal

Table 12 - API Performance Exclusion Criteria

Exclusion Criteria Number	Description
EC1	The paper is more than 5 years old (published before 2018).
EC2	The paper is not available in PDF format.
EC3	The paper is not written in English.
EC4	The paper does not address topics that are relevant to the research question.
EC5	The paper is not an Academic Journal

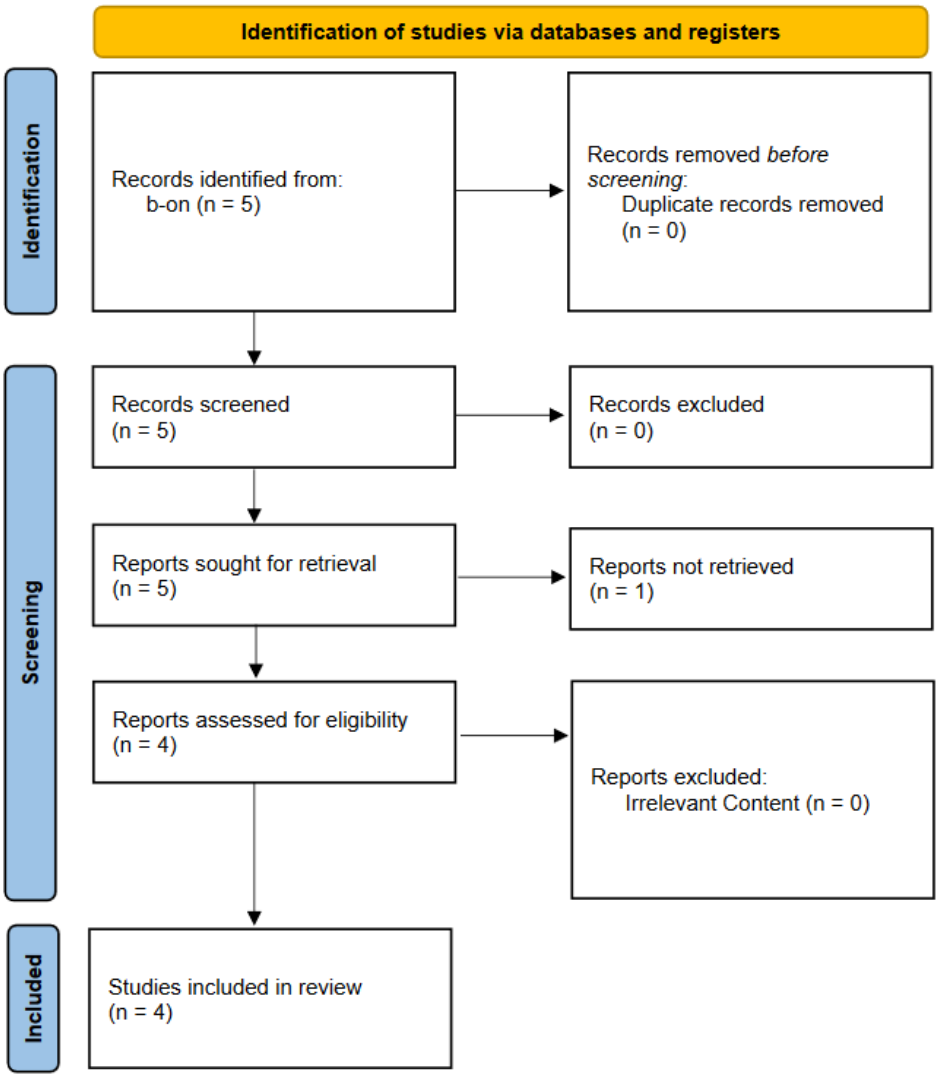

Figure 5 - PRISMA Diagram of API Performance Systematic Review

With the query and filters complete, the search was performed on the b-on digital library, resulting in 5 papers with no duplicate records. Subsequently, an analysis of the title and abstract was conducted on these five papers, leading to no exclusions as none were deemed irrelevant to the research question. Following this, it was performed an additional examination of the remaining five papers. One was inaccessible.

After reading the four papers, it was determined that all aligned with the goal of the research question. This process led to the inclusion of all the remaining documents in the systematic review.

Rinaldi et al. propose in “Improving the Performance of Actors on Multi-cores with Parallel Patterns” the use of parallel patterns to improve the performance of “high computational demanding applications” on multi-core systems, which implies adaptability, speed, and

efficiency from the scaling system. Implementing the patterns enhances the performance while keeping the safety and versatility of the program [32].

Nassif et al., in the article “Wikifying software artifacts” use wikifiers to locate URLs that link Wikipedia articles. These wikifiers were tested on multiple technologies as well as many configurations. Tokenization was one of those tested configurations which improved the performance and accuracy of the search [33].

Yang et al. attempt to enhance the performance of a streaming service by using Apache Storm and InfiniBand RDMA. The previous implementation “caused frequent context switches and buffer copies in the message transmission process”, which caused delay and reduced efficiency. Direct memory access (RDMA) reduces CPU overload and improves message throughput [34].

Garbin et al. analyse the overfitting and long training time of multilayered neural network learning. Dropout and batch normalization are the most popular implementations to tackle these issues [35].

With the prior evidence, we can confidently say that API performance encompasses metrics such as the speed, accuracy, efficiency, and adaptability of applications or systems.

3.5 Pattern Comparison of GraphQL and gRPC/Protobuf

As section 3.3 exhibited, patterns permit better flexibility if used and implemented in the correct situations. For GraphQL it allows the usage of all patterns due to the way it implements its queries and mutations for fetching and data modifications shown in [36]. As for gRPC and Protobuf, the examples shown by Netflix allow for pattern employment due to the message and service implementation in Protobuf [1][37]. As for versioning, the pattern is straightforward to implement in gRPC due to its inherent support compared to the evading stance from GraphQL [38][39].

4 Application Migration

This section presents the analysis of the selected open-source project for the technology migration. A look into the original work designs and accomplishments is done. The new application design uses gRPC and Protobuf, as was the project goal. It is migrated from GraphQL.

4.1 Base Project

The objective of the project was to develop a comprehensive parking management system. This system was built during the Software Development Laboratory (LABDSOF) of the master's degree at Instituto Superior de Engenharia do Porto (ISEP), where the goal was to work as a professional team to answer an RFP (Request for proposal) from a "client".

This system is designed to efficiently manage vehicle entry by checking available parking spaces suitable for the specific vehicle type. Additionally, it facilitates the departure process by performing automatic price calculations when exiting a park with a vehicle.

The project's source code is in an open-source GitHub repository [40].

4.1.1 Domain Model

Figure 6 shows the entities and models present in the system. The Park entity is the most relevant for this project due to its complexity. When migrating to gRPC, all models here required creating a file for each proto definition.

- **Vehicle:** As the name suggests, it represents the user's vehicles used to park.
- **VehicleType:** Represents all the vehicle types supported by the system. Each vehicle is of a singular kind.
- **User:** This entity represents the generic client, manager, or administrator.

- **ParkyWallet:** ParkyWallet is a point-based virtual wallet that can be used to pay parking bills. Parking the vehicle adds points to the user's wallet.
- **ParkyWalletMovements:** A history of all transactions performed in a ParkyWallet, being either payments or incomes.
- **EmployeeFile:** Entity meant to contain information about employees who manage the parks.
- **Park:** Physical place where the vehicle can be stored.
- **ParkingSpot:** Representation of a physical place inside a park.
- **PriceTable:** This represents all the possibilities for pricing. For each vehicle type and period of parking day, an entry exists

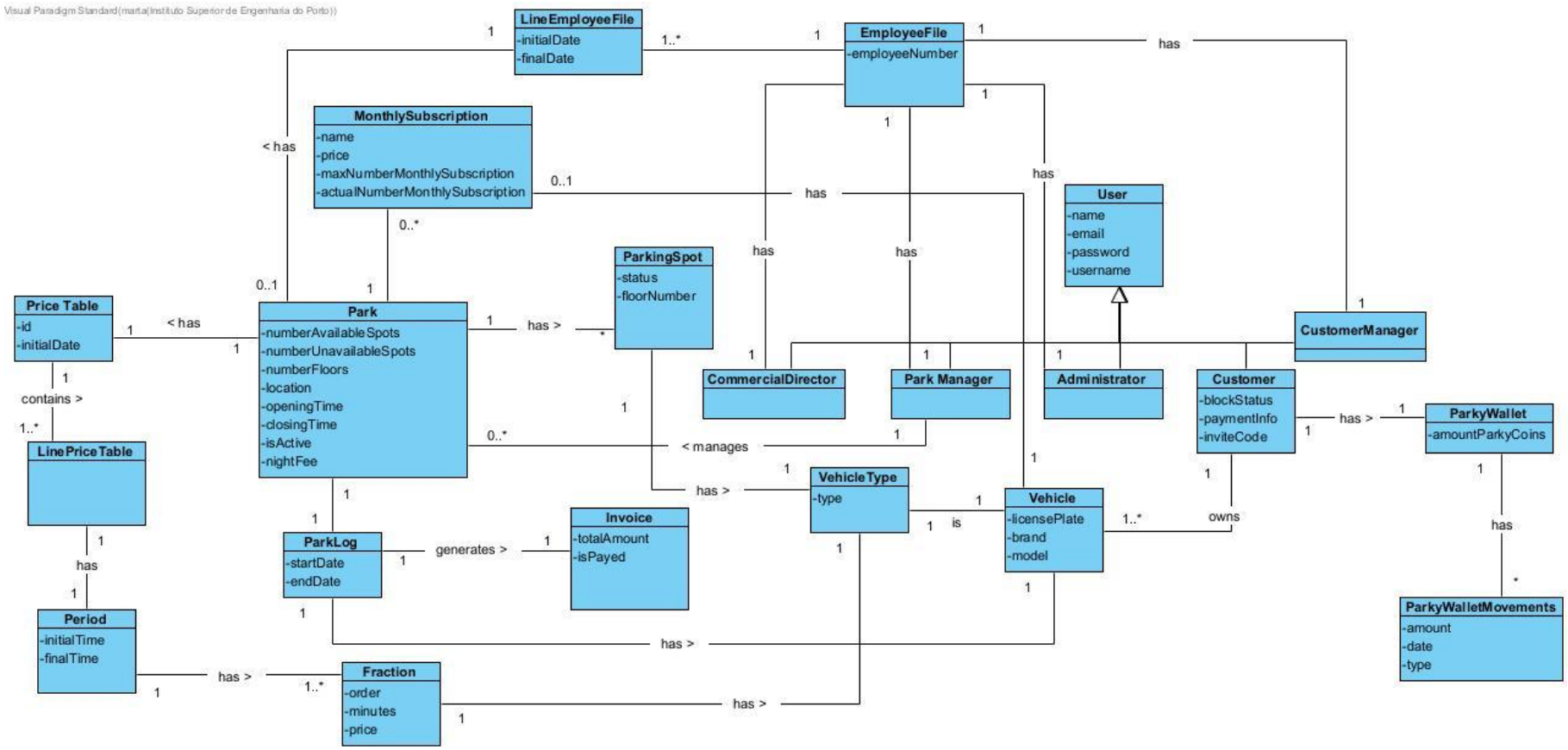

Figure 6 - Domain Model

4.1.2 Functional Requirements

As the introduction to the domain model and major entities is complete, what is next presented are the functional requirements most relevant to the project. Each following image shows the most significant use cases for each actor.

For the unregistered user the Figure 7 shows the relevant use cases.

- **User Registration:** To be able to register on the platform.

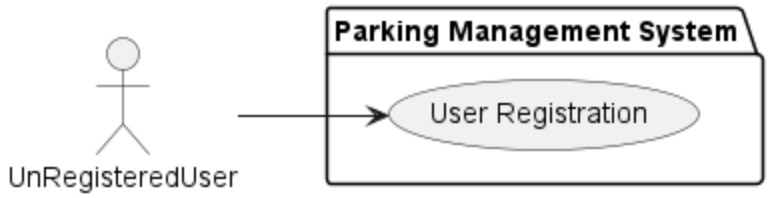

Figure 7 - Unregistered User Use Case Diagram

Park Manager actor has the use cases shown in Figure 8.

- **Enable/Disable Parking Spot:** Enable or disable parking spots to control the availability of parking spaces.

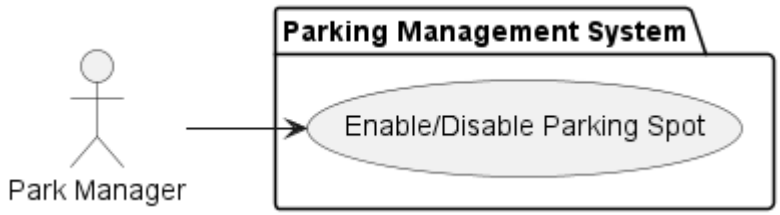

Figure 8 - Park Manager Use Case Diagram

For the Registered User the Figure 9 shows its use cases.

- **Manage Payment Methods:** Add, remove, and update payment methods for park payments.
- **Manage Vehicles:** Add, remove, and update vehicles for the system to admit registered vehicles and allow entry.
- **List Nearby Parks:** List of nearby parking locations.
- **Simulate Barrier Opening:** Open the barrier automatically when detecting a vehicle in the license plate reader.

- **Simulate Barrier Closing:** Close the barrier automatically when detecting a vehicle in the license plate reader.
- **Display Entry/Exit Messages:** Displayed messages at the entry and exit points of the park.
- **Show Parking Capacity:** View the current parking capacity of the selected park.
- **Display Parking Price:** View the parking prices for each vehicle type.
- **Filter Nearby Parks:** Filter nearby parks based on the vehicle type.
- **Manage ParkyWallet:** View the contents of the ParkyWallet.
- **Manage Account:** Delete and update account details and preferences.
- **Select Payment Method:** Select the default payment method at checkout.
- **Allow Exit Without Payment:** Exit the parking location without making the payment.

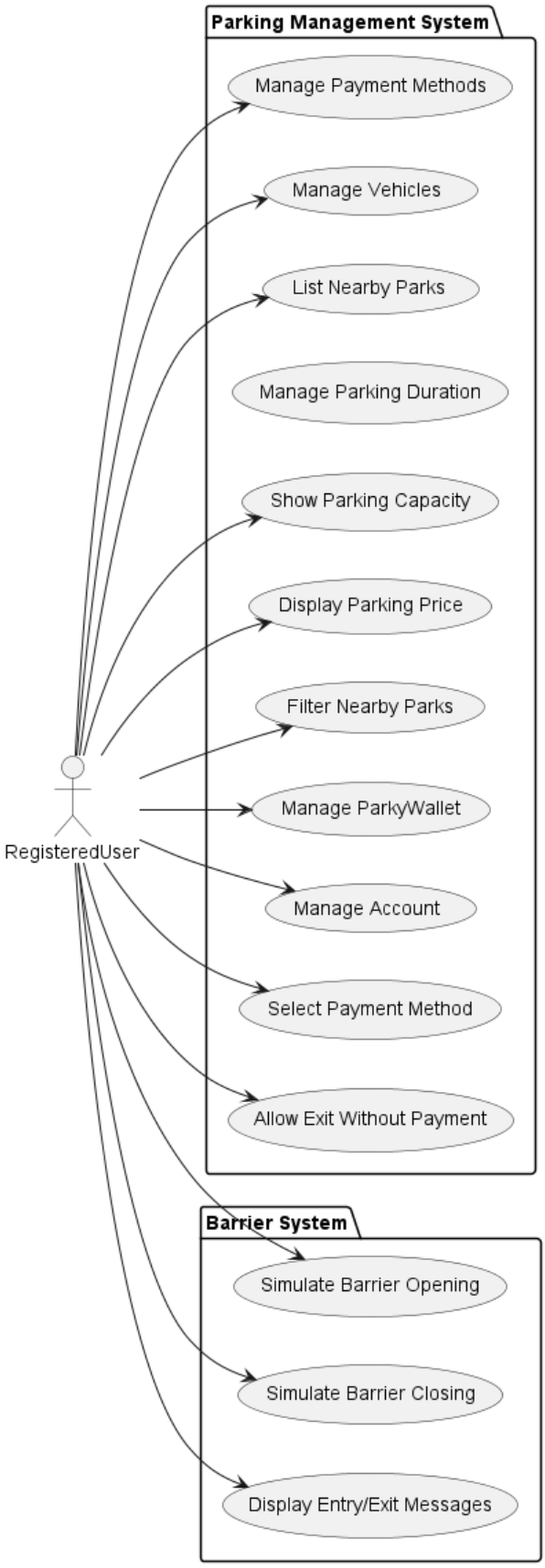

Figure 9 - Registered User Use Case Diagram

Commercial Manager has the following use cases shown in Figure 10.

- **Show Parking Capacity:** View the current parking capacity of the selected park.
- **Manage Parking Duration:** Control each vehicle park duration for pricing calculation.
- **Disallow Park Entry if Full:** Disallow entry into the park if it is full.
- **Read License Plates:** Read license plates of incoming vehicles to identify registered vehicles and allow entry.
- **Award Parky Coins on Registration and Hourly Parking:** Award Parky Coins to users upon registration and for each hour of parking.
- **Manage Parky Coin Award Values:** Update the values of Parky Coin awards.
- **Generate Parky Coin Usage Reports:** Generate reports on Parky Coin usage.
- **Register Pending Debts:** Register pending debts for users.
- **Manage Price Tables:** manage price tables to ensure that the correct pricing is assigned to each park.
- **Disallow Unsupported Vehicle Park Entry:** allow only supported vehicle types to enter the park.
- **Disallow Unregistered/Blocked User Park Entry:** Only registered and unblocked users are allowed to enter the park.

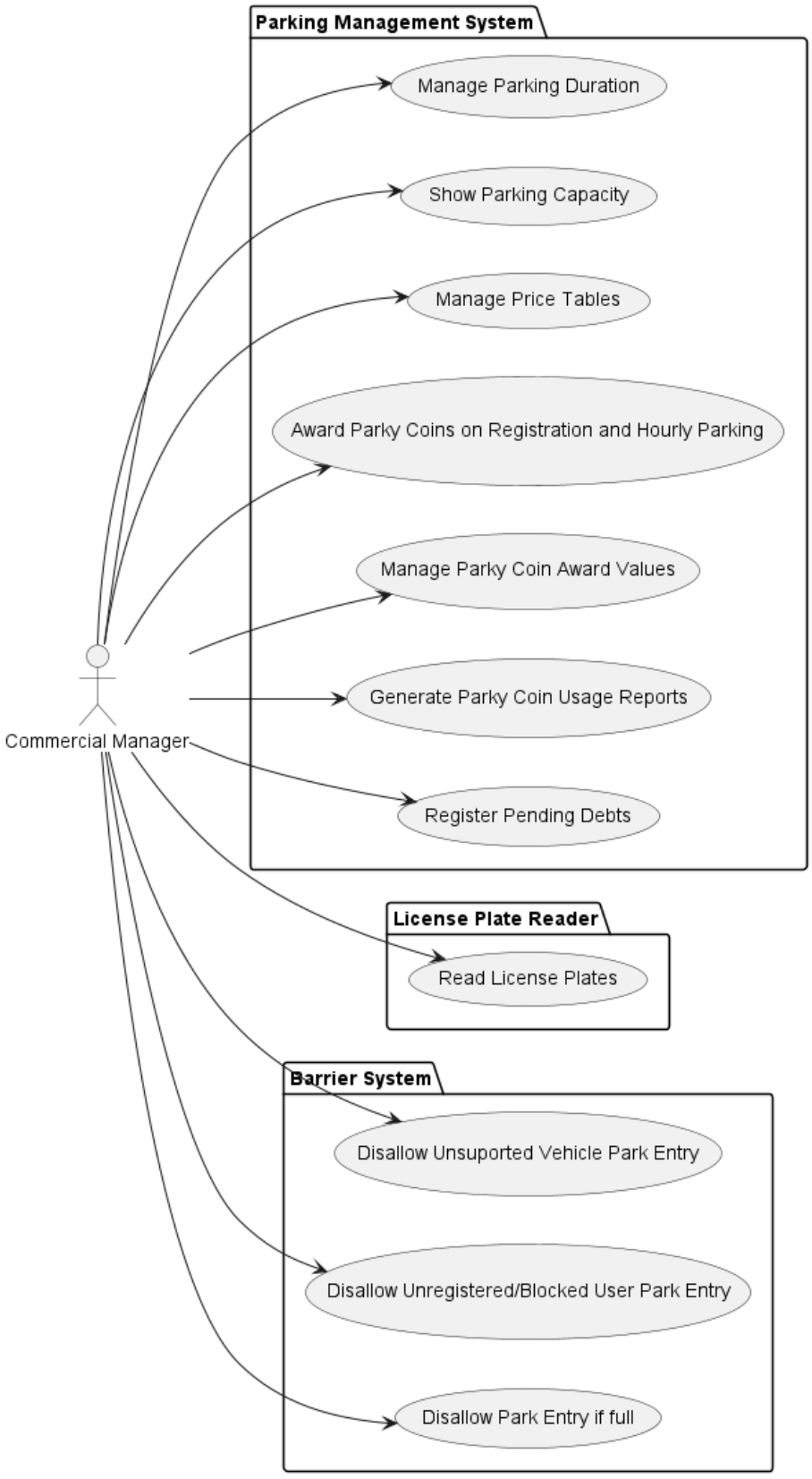

Figure 10 - Manager Use Case Diagram

4.1.3 Architecture

Figure 11 shows the logic diagram of the base system containing all components and communication possibilities.

The system took a monolithic architecture approach, meaning that most business logic is present inside the “BackOffice” component, developed in C#. As for its user-focused interfaces, two frontends were developed in Angular, one for the regular users and one for the system managers. Finally, the other interfaces are the entry and exit systems. Both handle the detection and reading of the vehicle license plate. This license plate reading triggers the business tasks for the vehicle to enter or exit the park. After the reading of the vehicle's license plate, the payment system simulates the process of paying a parking bill. All use HTTP for communication. For more information about the project, please check [41].

The project goal is to analyse the advantages of using gRPC vs GraphQL, so it was decided to perform the lowest possible number of changes conceivable to the base. Because of that, the project focus is on the “BackOffice” and “External Payment Service” components. These projects are where the new implementation is performed and tested.

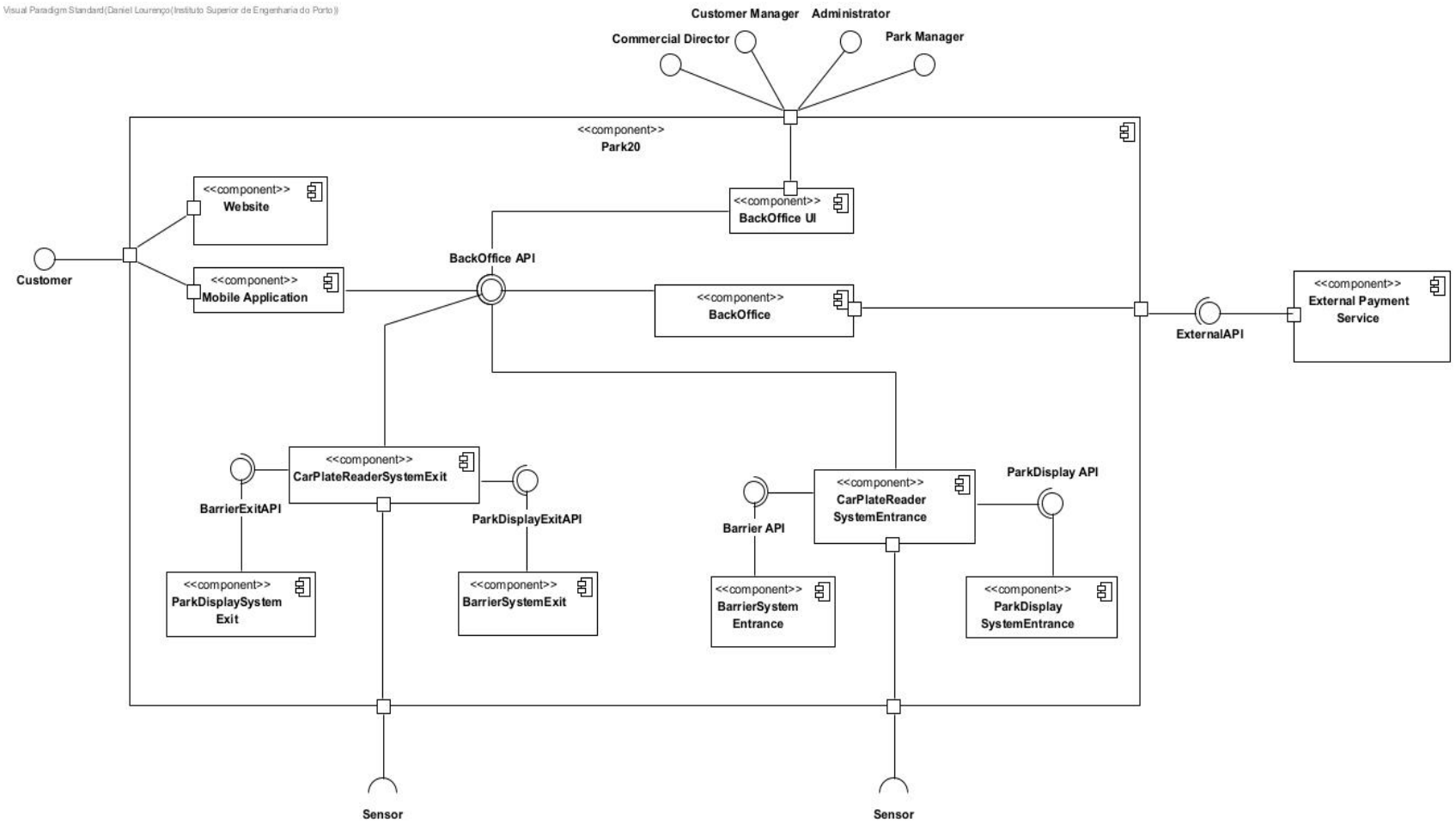

Figure 11 - Logical View (Level 2)

A multi-tiered architecture is present in both the Backoffice and the Payment Service. This architecture consists of controllers, services, models, and repositories, each tier with its responsibility in the business logic.

Figure 12 shows the backoffice architecture. The Payment Service Application is identical to the backoffice, but a database is not part of it. The succeeding list enumerates the tiers and their respective goal.

- **Controllers:** Entry points to the client communication.
- **IServices:** Interfaces that contain the methods to be implemented by the service instances.
- **Services:** Implements the logic outlined in the interface.
- **Mappers:** Maps the domain object to data transfer objects (DTOs) or vice-versa.
- **Domain:** Contains all entities that are relevant to the business.
- **DTOs:** Comprises all objects that are relevant to the client.
- **IRepositories:** Outlines the methods to be fulfilled by the repository instances.
- **DatabaseManagement:** Consists of queries used when carrying out any action to the database.
- **Infrastructure:** Contains all repositories that carry out actions to the database.

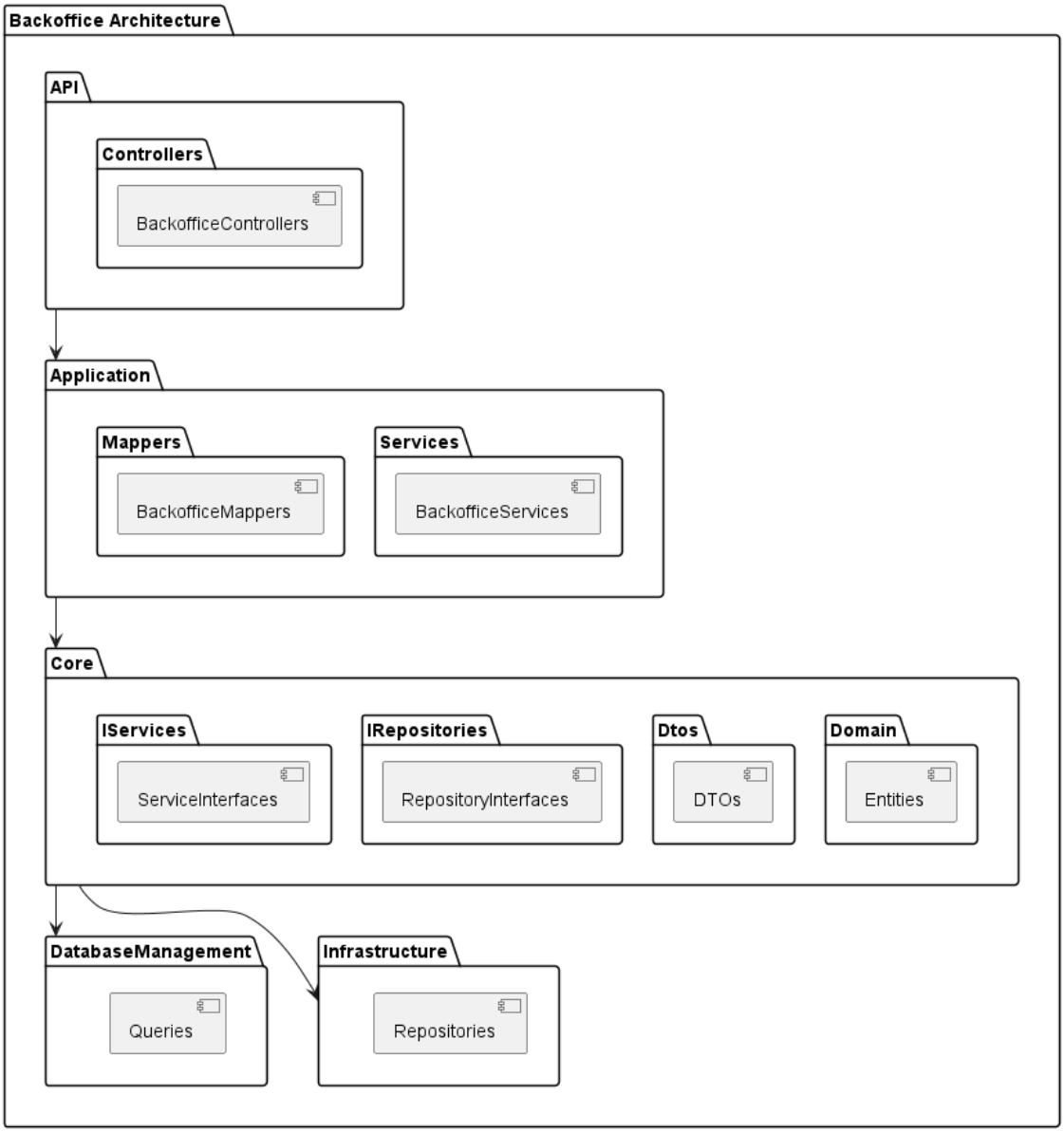

Figure 12 - Base Project Implementation View (Level 3)

More information about the project is available at the GitHub repository [41].

As remarked in section 3.3, several patterns were listed to improve flexibility without the need for a shift to newer technologies. They provide an option to avoid the time-consuming process of migrating the application(s) to newer technologies. The base application also employs the following patterns: the "Processing Resource API Endpoint," "Computation Function," "State Creation Operation," "Retrieval Operation," "State Transition Operation," and "Master Data Holder".

4.2 GraphQL Migration

To migrate the base project required the installation of dependencies such as Hot Chocolate and GraphQL Client [42][43]. These dependencies allow both a GraphQL server to be set up and to perform GraphQL requests.

The changes for the GraphQL server to be implemented were mainly in the program.cs file (Code 12), which was vital to streamline the GraphQL setup. Other changes were also required as part of the migration of logic in the HTTP controllers to either the query file if it's a method to get information, or the mutation file for every other task [44]. These changes allow the system to keep the same patterns as the original due to having a similar implementation as the HTTP endpoints in the original application.

```
builder.Services.AddGraphQLServer()
    .AddQueryType<Query>()
    .AddMutationType<Mutation>();
app.MapGraphQL("/graphql");
```

Code 12 - GraphQL Program.cs Config

The logic had only minor changes as it would call the same service logic excerpt of Code 13 is an example of such:

```
public async Task<bool> CheckIfUserIsRegistered(string username, string
password)
{
    return await _userService.CheckIfUserIsRegistered(username, password);
}
```

Code 13 - CheckRegisteredUser GraphQL Example

```
[HttpGet]
public async Task<IActionResult> CheckIfUserIsRegistered([FromQuery] string
username, string password)
{
    var response = await _userService.CheckIfUserIsRegistered(username,
password);
    return Ok(response);
}
```

Code 14 - CheckRegisteredUser Http Example

Due to applying very minor changes to the project, the architecture diagram is represented in Figure 13.

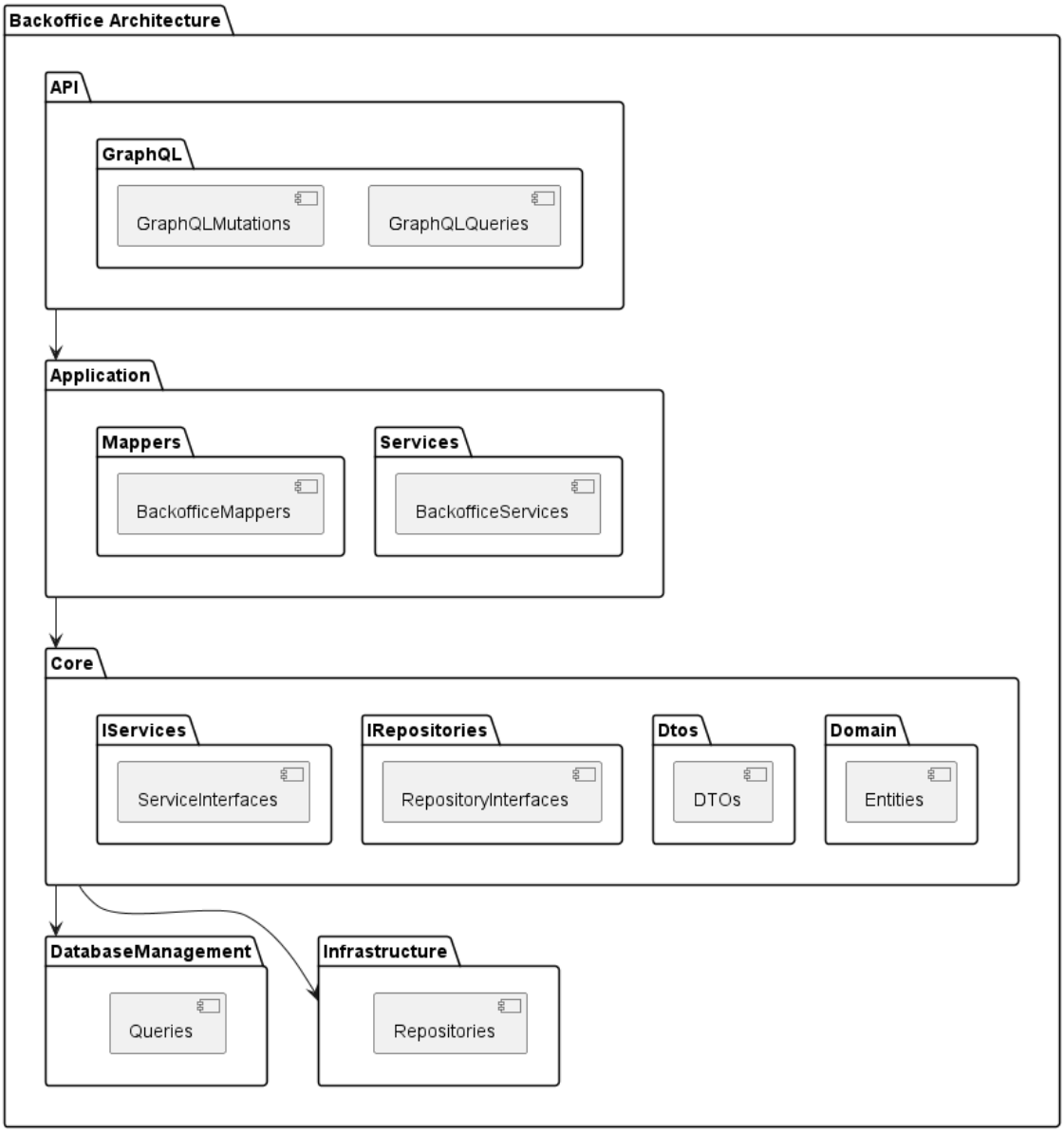

Figure 13 - GraphQL Implementation View (Level 3)

The migration only required the replication of the already developed functionalities to its relevant file, meaning that the migration for the GraphQL server worked out of the box. This was possible due to Hot Chocolate allowing for a code-first approach, thanks to a data-fetching technology called DataLoader [45].

To perform the request to the “External Payment Service”, it was changed from a simple HTTP to a GraphQL mutation as shown in Code 15.

```
var graphQLOptions = new GraphQLHttpClientOptions
{
    EndPoint = uri,
    HttpMessageHandler = new NativeMessageHandler(),
};
```

```
GraphQLHttpClient _client = new GraphQLHttpClient(graphQLOptions, new
NewtonsoftJsonSerializer());
var completeQueryString = "mutation($input:
PaymentRequestInput!){ processPayment(paymentRequest: $input) }";
var request = new GraphQLRequest
{
    Query = completeQueryString,
    Variables = new
    {
        input = new
        {
            amount = totalCost,
            token = token
        }
    }
};
dynamic response = await _client.SendMutationAsync<dynamic>(request);
```

Code 15 - GraphQL Mutation Request

This excerpt creates a GraphQL Client with a specific URL and serializer, which, after the initialization, the mutation request is assigned to the variable, and the string containing the mutation is used to create a GraphQL request in the following client call [46]. Another possibility was to use another dependency called Strawberry Shake [47]. This dependency reduces boilerplate code when performing client requests to the backend, as shown in the example.

```
var operation = new ProcessPaymentOperation(variables =>
variables.SetInput(new PaymentRequestInput
{
    Amount = totalCost,
    Token = token
}));
var result = await client.ExecuteAsync(operation);
```

Code 16 - Strawberry Shake Mutation Example

This dependency required the schema to be present locally along with defining the queries or mutations in local files. The implemented solution only contains the first example shown. This GraphQL project is present on GitHub [48].

4.3 gRPC and Protobuf Migration

The second technology, gRPC, required more changes due to the migration of the data transfer objects to proto and the creation of the RPC services Protobuf files [49].

4.3.1 Backoffice Migration

The multi-tier architecture contained minor tweaks, which was one of the goals. What changed was that the input data was moved to Protobuf, along with the controllers which implement the Protobuf services, as Code 18 shows. Those definitions allow the system to keep the same

patterns as the original implementation and the GraphQL one. As such, it also required mapping to convert the proto-objects to the relevant DTOs. Figure 14 denotes the mentioned gRPC architecture.

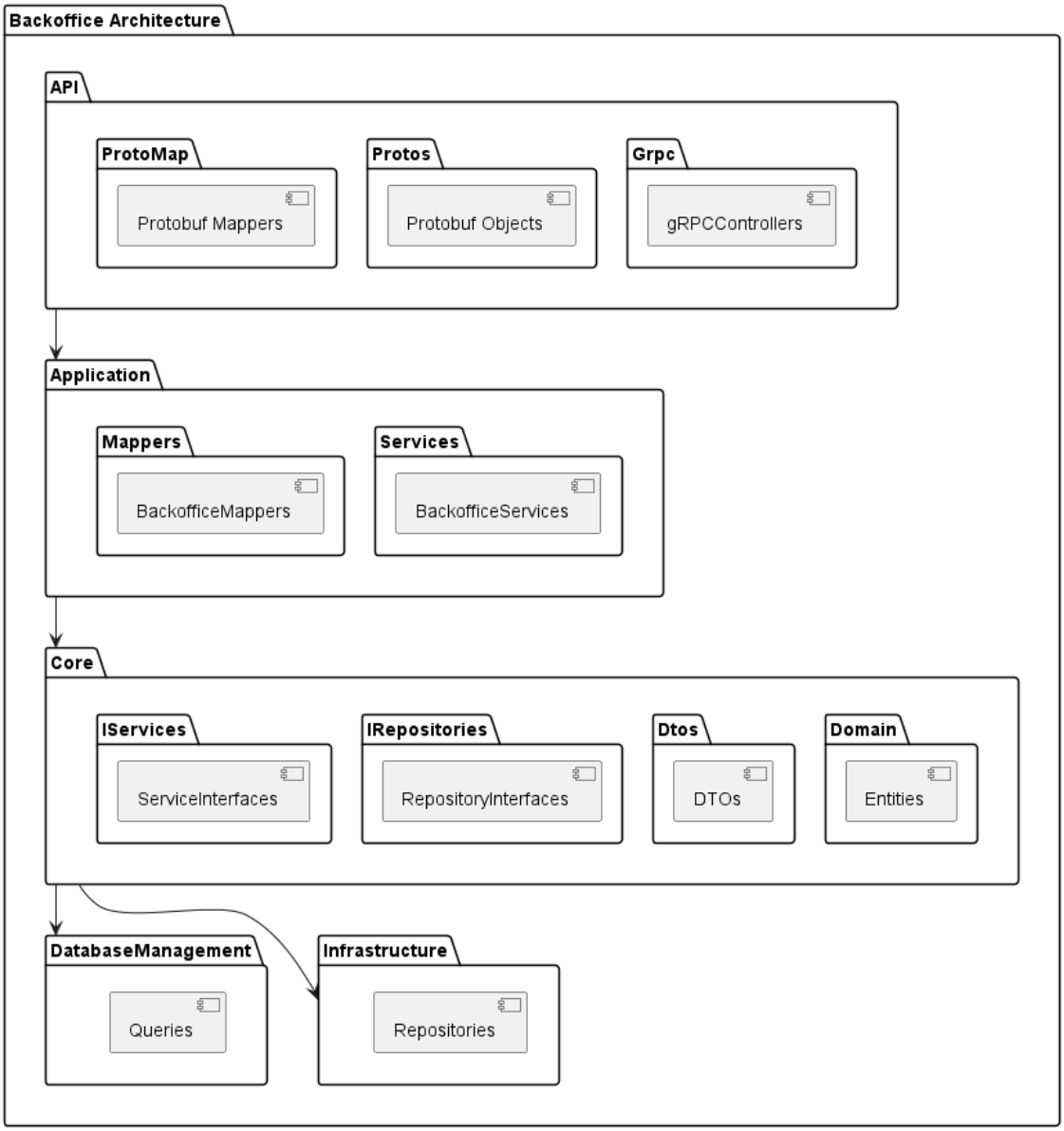

Figure 14 - gRPC Implementation View (Level 3)

The logical first step to the migration would be to create all returned objects in the controllers to Protobuf. An example of this migration is the code excerpts in Code 19, Code 20 and Code 21.

```
[HttpGet]
public async Task<IActionResult> GetAllParks()
{
    var parks = await _parkService.GetAllParks();
    return Ok(parks);
}
```

Code 17 - GetAllParks Controller Example

Code 17 shows the logic to get all information from all parks stored in the database. This is possible by calling an instance of the Park Service, which returns a Task<IActionResult> of a list of Park objects. Code 18 is the first step to define the gRPC controller, where the method name and the input and output types are specified. These types are also described in another Protobuf file. The "GetAllParks" RPC call is defined by receiving an "EmptyRequest" and returning multiple "ListPark" objects, represented in Code 19 and Code 20.

```
service ParkGrpc
{
    rpc GetAllParks (EmptyRequest) returns (ListPark);
    rpc GetAllParksClientStream(stream EmptyRequest) returns(ListPark);
    rpc   GetAllParksTwoSidedStream(stream   EmptyRequest)   returns(stream
ListPark);
    rpc GetAllParksServerStream(EmptyRequest) returns(stream ListPark);
}
```

Code 18 - Park RPC Definition

This definition contains only the field mask so it's possible to filter attributes from the return type when performing an empty message request.

```
message EmptyRequest
{
    google.protobuf.FieldMask fieldMask = 1;
}
```

Code 19 - Empty Request Definition

The "ListPark" message type defines that the field of ID 1 has a repeated object of type "Park" with the name parks. Code 21, Code 22, Code 23, Code 24, Code 25, Code 26, and Code 27 excerpts represent the rest of the park object.

```
message ListPark {
  repeated Park parks = 1;
}
```

Code 20 - List Park Response Definition

The more complex object inside the park.proto file uses some implemented subtypes and some custom Protobuf predefined types such as the timestamp. "PriceTable" and "ParkingSpot" are defined in Code 22 and Code 26 respectively.

```
message Park {
    optional int32 id = 1;
    optional int32 number_floors = 2;
```

```
    optional string park_name = 3;
    optional double latitude = 4;
    optional double longitude = 5;
    optional string location = 6;
    optional google.protobuf.Timestamp opening_time = 7;
    optional google.protobuf.Timestamp closing_time = 8;
    optional bool is_active = 9;
    optional double night_fee = 10;
    optional PriceTable price_table = 11;
    repeated ParkingSpot parking_spots = 12;
}
```

Code 21 - Park Object Definition

"PriceTable" continues mapping the park object by defining the "LinePriceTable" outlined in Code 23.

```
message PriceTable {
  optional string park_name =   1;
  optional double night_fee =   2;
  optional google.protobuf.Timestamp initial_date =   3;
  repeated LinePriceTable price_lines =   4;
}
```

Code 22 - Price Table Object Definition

This object contains only a reference to the period object also specified in Code 24.

```
message LinePriceTable {
  optional Period period =   1;
}
```

Code 23 - Line Price Table Object Definition

The period object continues to reuse the Protobuf predefined types in addition to the "Fractions" object defined in Code 25.

```
message Period {
  optional google.protobuf.Timestamp initial_time =   1;
  optional google.protobuf.Timestamp final_time =   2;
  repeated Fractions fraction_list =   3;
}
```

Code 24 - Period Object Definition

This object (Code 25) uses the "VehicleType" enum defined in the vehicle.proto file. For the enumerate to be reused, the vehicle.proto was imported at the start of the park file.

The enum is defined in Code 27.

```
message Fractions {
  optional int32 order =   1;
  optional int32 minutes =   2;
  optional VehicleType vehicle_type =   3;
  optional double price =   4;
}
```

Code 25 - Fraction Object Definition

The “ParkingSpot” object uses the optional attribute to request the parks by only the desired information.

```
message ParkingSpot {
    optional int32 parking_spot_id = 1;
    optional VehicleType vehicle_type = 2;
    optional bool status = 3;
    optional int32 floor_number = 4;
}
```

Code 26 - Parking Spot Object Definition

This vehicle type enumerate is defined as mentioned in another proto file from the park.proto. So, the enum is imported from the original and used in the Code 25 and Code 26 definitions. This object (Code 27) represents all the possible vehicle types supported in the system.

```
enum VehicleType {
  Motocycle =  0;
  Automobile =  1;
  GPL =  2;
  Electric =  3;
}
```

Code 27 - Vehicle Type Enum Definition

Code 28 shows the logic required to convert an entity park to a proto park object. In this case, it receives a list of parks and returns the object “ListPark” described in Code 20.

```
public static ListPark Map(List<Core.Domain.Park.Park> lp)
{
    List<Proto.Park> parks = [];
    lp.ForEach(p =>
    {
        parks.Add(Map(p));
    });
    return new ListPark
    {
        Parks = { parks }
    };
}
```

Code 28 - Map ListPark Object

This is an implementation of the “GetAllParks” defined in line 3 of Code 18 and a gRPC version of Code 17. The main difference between both executions is that there is a manual application of the field mask, and it uses the unary request type, meaning it’s one-to-one in terms of request

and response numbers. The iteration through the return parks is necessary, depending on the selected field mask contents, to return chosen child attributes. Netflix mentions that “you cannot select (mask) individual sub-fields in a message inside a list.” [1].

```
public class ParkGrpcService : ParkGrpc.ParkGrpcBase
{
    public override Task<ListPark> GetAllParks(EmptyRequest request,
ServerCallContext context)
    {
        ListPark filteredParks = new();
        ListPark parks = Mapper.Map(_parkService.GetAllParks().ToList());
        if (!request.FieldMask.ToString().Contains("parks"))
        {
            foreach (var park in parks.Parks)
            {
                Proto.Park filteredPark = new Proto.Park();
                request.FieldMask.Merge(park, filteredPark);
                filteredParks.Parks.Add(filteredPark);
            }
        }
        else
        {
            request.FieldMask.Merge(parks, filteredParks);
        }
        return Task.FromResult(filteredParks);
    }
}
```

Code 29 - GetAllParks Unary Implementation

“GetAllParksServerStream”, as the name indicates, is the implementation of server-side streaming where the client makes a singular request, and the server keeps sending information. For example, Code 30 only sends one response to the client. Returning a single response was made to keep the latency tests coherent and not introduce any artificial delays, but it is possible to return/”WriteAsync” the list of parks multiple times causing the client to receive all responses in the same request.

```
public override async Task GetAllParksServerStream(EmptyRequest request,
IServerStreamWriter<ListPark> responseStream, ServerCallContext context)
{
    ListPark filteredParks = new();
    ListPark parks = Mapper.Map(_parkService.GetAllParks().ToList());

    // Apply filtering based on FieldMask
    var fieldMask = request.FieldMask;

    if (fieldMask == null || !fieldMask.ToString().Contains("parks"))
    {
        foreach (var park in parks.Parks)
        {
            Proto.Park filteredPark = new Proto.Park();
            fieldMask.Merge(park, filteredPark);
            filteredParks.Parks.Add(filteredPark);
        }
    }
    else
```

```
        {
            fieldMask.Merge(parks, filteredParks);
        }
        await responseStream.WriteAsync(filteredParks);
    }
```

Code 30 - GetAllParksServerStream Server Stream Implementation

A two-sided stream is achieved in Code 31 in the method “GetAllParksTwoSidedStream”. The main contrast between this and the server streaming is the “await requestStream.MoveNext()”. This “await” causes the server to wait for the user to send a request. When a request is received, cause the server to send the list of parks.

```
    public override async Task
    GetAllParksTwoSidedStream(IAsyncStreamReader<EmptyRequest>   requestStream,
    IServerStreamWriter<ListPark> responseStream, ServerCallContext context)
    {
        while (await requestStream.MoveNext())
        {
            var request = requestStream.Current;
            ListPark filteredParks = new();
            // Apply filtering based on FieldMask
            if (request == null)
            {
                throw new RpcException(new Status(StatusCode.InvalidArgument,
    "Empty request received."));
            }

            ListPark parks = Mapper.Map(_parkService.GetAllParks().ToList());
            var fieldMask = requestStream.Current.FieldMask;

            if (fieldMask == null || !fieldMask.ToString().Contains("parks"))
            {
                foreach (var park in parks.Parks)
                {
                    Proto.Park filteredPark = new Proto.Park();
                    fieldMask.Merge(park, filteredPark);
                    filteredParks.Parks.Add(filteredPark);
                }
            }
            else
            {
                fieldMask.Merge(parks, filteredParks);
            }

            await responseStream.WriteAsync(filteredParks);
        }
    }
```

Code 31 - GetAllParksTwoSidedStream Two Sided Stream Implementation

This method, the client stream implementation of GetAllParks, “GetAllParksClientStream”, makes use of the “await requestStream.MoveNext()” while returning the object instead of writing asynchronously.

```
public override async Task<ListPark>
GetAllParksClientStream(IAsyncStreamReader<EmptyRequest> requestStream,
ServerCallContext context)
{
    ListPark filteredParks = new();
    while (await requestStream.MoveNext())
    {
        var request = requestStream.Current;

        if (request == null)
        {
            throw new RpcException(new Status(StatusCode.InvalidArgument,
"Empty request received."));
        }

        ListPark parks = Mapper.Map(_parkService.GetAllParks().ToList());
        var fieldMask = requestStream.Current.FieldMask;

        if (fieldMask == null || !fieldMask.ToString().Contains("parks"))
        {
            foreach (var park in parks.Parks)
            {
                Proto.Park filteredPark = new Proto.Park();
                fieldMask.Merge(park, filteredPark);
                filteredParks.Parks.Add(filteredPark);
            }
        }
        else
        {
            fieldMask.Merge(parks, filteredParks);
        }
    }
    return filteredParks;
}
```

Code 32 - GetAllParksClientStream Client Stream Implementation

For the backoffice to be able to call methods implemented in the payment service, the Protobuf file defined in that migration must be present in the backoffice files. After which, it is possible to perform RPC calls to the payment service.

```
private async Task<bool> SimulatePayment(string token, decimal totalCost)
        {
            PaymentResponse res;
            var fieldMask = FieldMask.FromFieldNumbers<PaymentResponse>(1,2);
            res = (await client.ProcessPaymentAsync(new PaymentRequest
{ Amount = (double)totalCost, Token = token, FieldMask = fieldMask }));
//Unary communication
            return res.Result;
        }
```

Code 33 - Call Payment Service Unary Request

The unary request is one-to-one, meaning that when “ProcessPayment” is called, one singular response is expected to arrive. We send to the API the following: the total cost, which is of a double type, a token previously added to the payment service, and a field mask. The return is a boolean that represents whether the processing was successful or not.

```
using (var call = client.ProcessPaymentServerStream(new PaymentRequest {
Amount = (double)totalCost, Token = token, FieldMask = fieldMask }))
{
    var responseStream = call.ResponseStream;
    while (await responseStream.MoveNext())         //Server sided stream
    {
        res = responseStream.Current.Result;
    }
}
```

Code 34 - Call Payment Service Server Sided Stream

The server-side stream expects multiple boolean results from a singular transaction. The RPC call of the method remains the same as the unary. This implementation is similar to Code 32 where the client waits for the server to respond to check the result.

```
var requestStream = client.ProcessPaymentTwoSideStream();         //Two sided
stream
DateTime startTime = DateTime.Now;
var responseTask = Task.Run(async () =>
{
                    await        foreach        (var        response        in
requestStream.ResponseStream.ReadAllAsync())
    {
        res = response.Result;
        DateTime endTime = DateTime.Now;
        TimeSpan duration = endTime - startTime;
        durations.Add(duration.TotalMilliseconds - 3000);
    }
});

for (int i = 0; i < 3; i++)
{
    startTime = DateTime.Now;
    await requestStream.RequestStream.WriteAsync(new PaymentRequest { Amount
= (double)totalCost, Token = token, FieldMask = fieldMask });
}

await requestStream.RequestStream.CompleteAsync();
await responseTask;
```

Code 35 - Call Payment Service Two-Sided Stream

Like Code 31, this method uses parts of both Code 34 and Code 36 to perform and receive multiple requests and responses. The “WriteAsync” and “ReadAllAsync” are some pieces of evidence of such abilities.

```
var stream = client.ProcessPaymentClientStream();
await   stream.RequestStream.WriteAsync(new   PaymentRequest   {   Amount   =
(double)totalCost, Token = token, FieldMask = fieldMask }); //Client sided
stream
await stream.RequestStream.CompleteAsync();
res = (await stream.ResponseAsync).Result;
```

Code 36 - Call Payment Service Client Sided Stream

The client stream is like Code 30, where the write async can be executed multiple times, and the client alerts the server with “CompleteAsync” when the streaming is finished. The client then expects a singular result for several transactions.

4.3.2 External Payment Service Migration

Like the backoffice migration, the first step is to define the input and output models and, subsequently, the RPC service proto files. Code 37 and Code 38 show the previously mentioned results of the migration. This migration is required for tests between service communication to be possible and analysed.

```
service PaymentGrpc {
    rpc ProcessPayment (PaymentRequest) returns (PaymentResponse);
    rpc ProcessPaymentClientStream (stream PaymentRequest) returns
(PaymentResponse);
    rpc ProcessPaymentServerStream (PaymentRequest) returns (stream
PaymentResponse);
    rpc ProcessPaymentTwoSideStream (stream PaymentRequest) returns (stream
PaymentResponse);
    rpc AddToken (AddTokenRequest) returns (AddTokenResponse);
    rpc GenerateToken (GenerateTokenRequest) returns (GenerateTokenResponse);
}
```

Code 37 - Payment RPC Service Definition

Both definitions are present in the payment service, as well as in the backoffice. This way, backoffice can perform RPC calls to this service. Code 38 is the required message types for the service definition of Code 37.

```
message PaymentRequest {
    string token = 1;
    double amount = 2;
    google.protobuf.FieldMask fieldMask = 3;
}

message PaymentResponse {
    bool result = 1;
}

message AddTokenRequest {
    string token = 1;
    google.protobuf.FieldMask fieldMask = 2;
}

message AddTokenResponse {
    bool result = 1;
}

message GenerateTokenRequest {
    int32 CardNumber = 1;
    int32 Cvv = 2;
    string FullName = 3;
    google.protobuf.Timestamp ExpirationDate = 4;
    google.protobuf.FieldMask fieldMask = 5;
```

```
}

message GenerateTokenResponse {
    string token = 1;
}
```

Code 38 - Payment Models Definition

All code presented gRPC proto definitions, and more are in the GitHub repository [50].

5 Tests and Solution Evaluation

The goal is to perform performance and flexibility testing, collect the metrics and compare the results between both technologies. An unsuccessful attempt was made to isolate the systems using docker. The results shown are without any isolation.

An introduction to the methodology used and the GQM (Goal Question Metric) is performed. A listing of all performance testing using k6 follows, just after, a section containing the description of all flexibility testing, and finally, a comparison between both technologies using the collected results for the selected metrics.

5.1 Methodology

The GQM approach entails three levels [51]:

- **Conceptual level (GOAL):** Defined for an object, based on various quality models and perspectives. Objects of measurement include products, processes, and resources.
- **Operational level (QUESTION):** A set of questions based on the goal to verify whether the objective has been attained.
- **Quantitative level (METRIC):** For each question, a set of metrics is defined, objective or subjective, to answer the question quantitatively.

The measurement model begins with a goal being set. After which, questions are defined, and these identify the primary elements. Subsequently, each question then is broken down into metrics. Each chosen metric can be used multiple times for different questions, given that it is within the original goal. After everything is set, the model is ready, and the strategy for data collection for each metric should be defined [51].

As previously stated, the GQM method evaluates the influence of migrating an application to gRPC technology from GraphQL on performance and flexibility attributes. The same questions and metrics are applied to both solutions, thus allowing the project to be evaluated by comparing the before and after, this facilitates the assessment of the achieved results.

Table 13 - GQM Goal

Goal	Uncover the effect of migrating an API from GraphQL to gRPC on its performance and flexibility.

Table 13 represents the picked goal. The following sections list the questions and metrics that are used for each of the assessed characteristics.

5.1.1 Performance

Table 14 shows the question specified for the performance characteristic and the selected metrics used to evaluate.

Table 14 - GQM Performance Questions

Question	Metrics
Is the performance of the GraphQL original application adequate?	Average, Maximum, Median, Minimum, 90th Percentile, 95th Percentile - K6
Is the performance of the gRPC migrated application adequate?	

K6 can measure performance-related metrics in milli-seconds. This tool is used to test the application, as shown in the example described in page 82. When using it, it collects the following metric information:

- **Average:** Average time of all made requests during the testing period.
- **Maximum:** The request that took the most time to be completed.
- **Median:** The value separating the higher and lower half in a data set.
- **Minimum:** The request that took the least time to conclude.
- **90th percentile:** The best 90% of the requests.
- **95th percentile:** The best 95% of the requests.

5.1.2 Flexibility

As for flexibility, Table 15 sets the metrics that are used to assess the application's flexibility trait.

Table 15 - GQM Flexibility Questions

Question	Metrics
Is the GraphQL application flexibility acceptable?	developerOpinion, subjective analysis by author. Using a scale of 1-5.
Is the new gRPC and Protobuf application flexibility acceptable?	

Due to the nature of the project, the author performs a subjective evaluation, using a scale of 1-5, by modifying the APIs and verifying their consequences, subsequently comparing the required operations on each technology.

5.2 Experiments

This section lists and performs a comparison of both previously set goals. These goals are to analyse performance and flexibility, and after the completion of the analysis, a contrast is made between the technology before and after the migration.

5.2.1 Performance Testing

K6 from Grafana labs was used to execute load tests and attain request-specific metrics [52]. This tool allowed for both GraphQL testing, with its HTTP support, and gRPC tests, where some took advantage of the streaming support provided by the tool. All tests performed 3000 requests to the system with no time constraints.

5.2.1.1 Get Operations

This section lists all tested fetching operations where the goal is to get the information as quickly and as stable as possible. It is divided by method and technology.

5.2.1.1.1 GetPartialParks

This test checks which technology is quicker at fetching only parts of the park object. For this, both GraphQL and gRPC test scripts were created and executed.

5.2.1.1.1.1 GraphQL

This setup defines all metrics that should be captured, the unit of time shown in the summary, and the stage setup, which specifies the time limit and the number of iterations.

```
import http from "k6/http";
import { htmlReport } from "https://raw.githubusercontent.com/benc-uk/k6-reporter/main/dist/bundle.js";

// Define the load test configuration
export const options = {
  summaryTrendStats: [
    "avg", //average
    "min",
    "med", //median
    "max",
    "p(50)",
    "p(90)",
    "p(95)",
    "p(99)",
    "p(99.99)",
    "count",
  ],
  duration: '20m',
  summaryTimeUnit: "ms",
  iterations: 3000
};
```

Code 39 - Test setup for GraphQL GetPartialParks

This query calls the AllParks, selecting only some of the attributes of the park object. This request is preserved in the same format, meaning that the query stays the same as if executed by the Postman application or Banana Cake Pop, an open-source IDE, or Strawberry Shake GraphQL client, introduced in page 81 [47][53][54].

```
const query = `
query AllParks {
  allParks {
    id
    numberFloors
    parkName
    latitude
    longitude
    location
    openingTime
    closingTime
    isActive
    nightFee
  }
}
`;
```

Code 40 - Query definition for GraphQL GetPartialParks

This function is one of the four lifecycles present in k6 [55]. This method sets JSON as part of the header for the content type and performs a call to the backend with the previously created header and query.

```
export default function () {
  const headers = {
    "Content-Type": "application/json",
  };

  const res = http.post(
    "http://localhost:5000/graphql",
    JSON.stringify({ query: query }),
    {
      headers: headers,
    }
  );
}
```

Code 41 - Test function for GraphQL GetPartialParks

The function in Code 42 intends to use the htmlReport to generate HTML files with the collected metrics [56].

```
export function handleSummary(data) {
  return {
    "graphqlPartialParks.html": htmlReport(data),
  };
}
```

Code 42 - Ending script for GraphQL GetPartialParks

5.2.1.1.1.2 gRPC

For gRPC testing, the first step is to load proto files. These are the same as the backoffice application.

```
import grpc from "k6/net/grpc";
import { check } from "k6";
import { htmlReport } from "https://raw.githubusercontent.com/benc-uk/k6-reporter/main/dist/bundle.js";

const client = new grpc.Client();
client.load(
  [
    "../Park20.Backoffice.Api/Park20.Backoffice.Api/Protos/Services",
    "../Park20.Backoffice.Api/Park20.Backoffice.Api/Protos/",
  ],
  "parkgrpc.proto"
);
```

Code 43 - Proto setup script for gRPC GetPartialParks

After the proto files are loaded, the test configuration is now complete. Code 44 has the same options as GraphQL.

```
// Define the load test configuration
export const options = {
  summaryTrendStats: [
    "avg", //average
    "min",
    "med", //median
    "max",
    "p(50)",
    "p(90)",
    "p(95)",
    "p(99)",
    "p(99.99)",
    "count",
  ],
  duration: '20m',
  summaryTimeUnit: "ms",
  iterations: 3000
};
```

Code 44 - Test setup for gRPC GetPartialParks

As for executing a GetPartialParks test, we define a default function. We begin by connecting to the backoffice, creating the fieldmask, and invoking the GetAllParks specific method outlined in parkgrpc.proto file. After the call, we get a response, check its status, and close the connection as shown in Code 45.

```
export default () => {
  client.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  const req = {
    fieldMask:
```

```
"id,numberFloors,parkName,latitude,longitude,location,openingTime,closingTi
me,isActive,nightFee",
  };

  const response = client.invoke("Proto.ParkGrpc/GetAllParks", req);

  check(response, {
    "status is OK": (r) => r && r.status === grpc.StatusOK,
  });

  client.close();
};
```

Code 45 - Test function for gRPC GetPartialParks

After the test execution, an HTML report is generated. This report contains details of metrics collected during the test (Code 46).

```
export function handleSummary(data) {
  return {
    "grpcPartialParks.html": htmlReport(data),
  };
}
```

Code 46 - Ending script for gRPC GetPartialParks

5.2.1.1.2 GetAllParks

Like GetPartialParks, this is the same method where the difference is the amount of data requested. In this case, it requests all attributes of the park object.

Figure 16 is the sequence diagram for the GetAllParks GraphQL implementation. The request execution shown in Code 62 is like Code 41, where the query content differentiates them. This query is defined in Code 61.

For the gRPC implementation Figure 17 represents the sequence diagram. Code 63 like Code 45, performs a request to get the whole information from all the parks in the database. The field masks are only defined as “parks” to return all park information instead of specifying all attributes as GraphQL requires.

5.2.1.1.2.1 Streaming Implementations

GetAllParksServerStream tries to perform the same task as GetAllParks. Adaptions in this test script were necessary to allow streaming, such as creating a stream object from the client connection and the intended method to call. Instead of invoking the same way as in the unary, an “end” event is attached to the stream to close the connection when the server has finished sending information. The Figure 18 represents the sequence diagram of the server stream operation.

```
export default () => {
  client.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  const req = { fieldMask: "parks" };
```

```
  const stream = new Stream(client,
"Proto.ParkGrpc/GetAllParksServerStream");

  stream.on("end", function () {
    // The server has finished sending
    client.close();
    console.log("All done");
  });

  // send a message to the server
  stream.write(req);
};
```

Code 47 - Test function for gRPC GetAllParksServerStream

A two-sided stream is similar to the server-side and the client stream. What differentiates them is who can end the stream. In this case, the server and the client can stop sending. Code 64 is the implementation of the two-sided stream in the backoffice. Figure 19 is the sequence diagram of this two-sided stream.

As for the client stream, Code 65 is the implementation and the Figure 20 its sequence diagram. Only the client side can end the stream, as the server only sends one response.

Note: to check the information received on the script a stream.on("data") function is needed.

5.2.1.2 Create Operations

This section lists all tested create operations. The goal is for objects to be stored in the database as fast and steady as possible. It is divided by method and technology (GraphQL followed by gRPC).

5.2.1.2.1 CreateUser

This test tries to find which communication is quicker when creating a new user. In the testing script, a random string generator was used to reduce the chance of a user existing in the database.

The sequence diagram for this test is characterised by Figure 21. Like the other GraphQL tests previously shown, this one differs in the mutation setup and the parameters used for querying the backoffice presented in Code 66 and Code 67. Code 67 function generates a random string of 20 characters and uses it as input for the user data. This data is specified in the variables part of the JSON that is sent in a request to the backend.

The gRPC test of CreateUser also makes use of the string generator. In this instance, the request contains more data than when querying all parks, as it includes the required information for the user creation as shown in Code 68. The sequence diagram for this test is characterised by Figure 22.

5.2.1.3 Update Operations

This section lists all tested operations to update previously existing objects. The goal is to change the information to the newly provided one as quickly and reliably as possible. It is divided by method and technology.

5.2.1.3.1 UpdateParkingValue

This test checks the performance of both technologies for an update task. It updates the parking configuration value.

The implementation of this GraphQL mutation is denoted by Figure 23. Code 69 GraphQL query is the mutation called when the parking configuration value needs to be updated to the new value sent as input. Code 70 test function uses randomIntBetween to generate a number between 0 and 99999. This generated number is the new value for the backoffice.

The gRPC sequence diagram is represented by the Figure 24. Similar to CreateUser, Code 71 generates a random number. This method then calls the backoffice with the number in the amount field.

5.2.1.3.2 LeavePark

LeavePark test is the most complex test. This test requires a call to the Payment System so the token exists when the user tries to exit the parking lot. This token is required so that the processing of the payment is successful.

GraphQL implementation is denoted by the Figure 25. Code 72 is the initial mutation to add a token to the payment system. Code 73 token setup is executed only once at the start of the test execution. It uses the addTokenMutation to add a specific token to the system. Notice that this HTTP post is performed to a different port. This different port is the PaymentSystem. After the successful addition of the token, the execution leaveParkMutation is next. Code 74 request calls to the backoffice. Code 75 test function calls the backoffice to simulate a user exiting the park. For this, an isEntrance boolean, a string licencePlate, and a string parkName are sent to the backoffice to identify the user, as well as the exit location.

Unary gRPC sequence diagram is represented by Figure 26. The Code 76 setup has the same intent as the GraphQL one, to add a token to the PaymentSystem. In this case, it requires two clients, one for the backoffice and one for the payment system. The payment client is used only once for the payment token registration. Code 77 function calls the backoffice to alert it of a detected vehicle trying to exit.

5.2.1.3.2.1 Streaming Operations

This section lists all tested streaming operations. All subsequent methods are associated with the unary leave park request, but the objective is to use streaming as an alternative to unary communication. It tries to get the information as swiftly and as unfluctuating as possible. It's divided by the streaming type, as all use gRPC technology.

5.2.1.3.2.1.1 LeaveParkServerStream

This implementation of leave park is represented by the Figure 27. LeaveParkServerStream goal is to evaluate streaming between the backoffice and the PaymentSystem. In this case, the PaymentSystem is the server. Code 34 requires changes to support streaming. The test script is the same as Code 77. Code 48 presents the implementation for the server streaming. A single request is made from the backoffice, and multiple values can be returned as evidenced by a "WriteAsync".

```
public override async Task ProcessPaymentServerStream(Protos.PaymentRequest
request,        IServerStreamWriter<PaymentResponse>        responseStream,
ServerCallContext context)
{
    PaymentResponse response = new();

request.FieldMask.Merge(Mapper.MapPaymentReponse(_paymentService.ProcessPay
ment(Mapper.Map(request))), response);
    await responseStream.WriteAsync(response);
}
```

Code 48 - PaymentSystem ProcessPaymentServerStream Implementation

5.2.1.3.2.1.2 LeaveParkTwoSidedStream

Similar to the previous test, the script is the same by reusing Code 77. As for streaming support in the backoffice, they are present in the excerpt of Code 35. “Process Payment Two Side Stream” is the implemented method to support two-sided streaming.

Excerpts like “while(requestStream.MoveNext().Result)” and “WriteAsync” are some of the evidence of two-sided, by waiting for a client request and writing the result asynchronously. The sequence diagram of this implementation is represented by Figure 28.

```
public                override                async                Task
ProcessPaymentTwoSideStream(IAsyncStreamReader<Protos.PaymentRequest>
requestStream,     IServerStreamWriter<PaymentResponse>     responseStream,
ServerCallContext context)
{
    PaymentResponse response = new();
    while (requestStream.MoveNext().Result)
    {
        var request = requestStream.Current;

        if (request == null)
        {
            throw  new  RpcException(new  Status(StatusCode.InvalidArgument,
"Empty request received."));
        }

request.FieldMask.Merge(Mapper.MapPaymentReponse(_paymentService.ProcessPay
ment(Mapper.Map(request))), response);
    }
    await responseStream.WriteAsync(response);
}
```

Code 49 - PaymentSystem ProcessPaymentTwoSideStream Implementation

5.2.1.3.2.1.3 LeaveParkClientStream

The client-side stream was the final test, and like the last two tests, this one has a method implemented to support the client stream, and as such, the “while(requestStream.MoveNext().Result)” is the evidence. For the return, there is no “WriteAsync“ as it returns the result as a single response. The sequence diagram of this implementation is represented by Figure 29.

```
public                    override                    Task<PaymentResponse>
ProcessPaymentClientStream(IAsyncStreamReader<Protos.PaymentRequest>
requestStream, ServerCallContext context)
{
    PaymentResponse response = new();
    while (requestStream.MoveNext().Result)
    {
        var request = requestStream.Current;

        if (request == null)
        {
            throw  new  RpcException(new  Status(StatusCode.InvalidArgument,
"Empty request received."));
        }

request.FieldMask.Merge(Mapper.MapPaymentReponse(_paymentService.ProcessPay
ment(Mapper.Map(request))), response);
    }
    return Task.FromResult(response);
}
```

Code 50 - PaymentSystem ProcessPaymentClientStream Implementation

5.2.1.3.2.1.4 Leave Park Additional Metrics Collection

Due to this test trying to test the communication throughput between applications, it is necessary to collect metrics at its core just before and after the request, thus allowing the calculation values for their comparison. The removal of one second to the final value is due to an artificial delay in the payment system.

```
DateTime startTime = DateTime.Now;
dynamic response = await _client.SendMutationAsync<dynamic>(request);
DateTime endTime = DateTime.Now;
TimeSpan duration = endTime - startTime;
durations.Add(duration.TotalMilliseconds - 1000);
```

Code 51 - Request Duration Collection

After this gathering of timespans and the respective test has been completed the method in Code 52 is called. This writes the collected metrics and uses them to calculate the average, minimum, maximum, median, 90th percentile and 95th percentile and print the respective results.

```
private static readonly List<double> durations = [];
public void PrintMetrics()
{
    double sum = 0;
    durations.Sort();
    string filePath = "payment.csv";
    string content= "req;time";
    int i = 0;
    foreach (var duration in durations)
    {
        i++;
        sum += duration;
        content += "\n" + i + ";" + duration;
    }
```

```
    File.WriteAllText(filePath, content);
    if (durations.Count > 0)
    {
        double min = durations.First();
        double max = durations.Last();
        double avg = sum / durations.Count;
        double median = durations.Count % 2 == 0 ?
            (durations[durations.Count / 2 - 1] + durations[durations.Count
/ 2]) / 2.0 :
            durations[durations.Count / 2];

        double p90 = durations[(int)Math.Ceiling(0.9 * durations.Count) - 1];
        double p95 = durations[(int)Math.Ceiling(0.95 * durations.Count) -
1];

        Console.WriteLine($"Average: {avg} milliseconds");
        Console.WriteLine($"Minimum: {min} milliseconds");
        Console.WriteLine($"Median: {median} milliseconds");
        Console.WriteLine($"Maximum: {max} milliseconds");
        Console.WriteLine($"p(90): {p90} milliseconds");
        Console.WriteLine($"p(95): {p95} milliseconds");
    }
}
```

Code 52 - Calculation of Test Metrics

5.2.2 Flexibility Testing

This test checks if variations were to be made to the input or the output, which technology allows for changes without too many adjustments.

5.2.2.1 GraphQL

The first test was to change the return object where it would contain a new description of the processing, as well as the previously used boolean [57]. For the payment system to return the new parameter, it needed to be updated. This change then caused the need to update the mutation on the backoffice.

For the next test, the goal was to rename one of the fields and subsequently deprecate it [58]. This test renames a field, and like the previous test, it requires updating the mutation on the backoffice side. As for the deprecation, no warning is thrown when attempting to get the value of the “ConfirmationString”.

GraphQL supports versioning but has a stance against it [59]. So, to implement such functionality, we can either provide a new GraphQL server for each new version or add a version constraint to the fields, as the examples shown in Code 53 and Code 54.

```
services.AddGraphQLServer("v1").AddQueryType<QueryV1>();
services.AddGraphQLServer("v2").AddQueryType<QueryV2>();

app.UseEndpoints(e =>
{
    e.MapGraphQL("/graphql/v1", "v1");
```

```
    e.MapGraphQL("/graphql/v2", "v2");
});
```

Code 53 - GraphQL Server Versioning

```
{
    mutation($input: PaymentRequestInput!)
    {
        processPayment(paymentRequest: $input)
        {
            successful(versionConstraint: ">=1.0.0"),
            confirmationString(versionConstraint: ">=2.0.0")
        }
    }
}
```

Code 54 - Mutation Versioning

With this versioning, we can add a specific version when the confirmationString is meant to be returned to the requester as part of the payload by supplying a version above or equal to 2.0.0.

5.2.2.2 gRPC

For the gRPC flexibility test, the goal stayed the same. The first test was to add a new return type. This test required the proto files to be updated both on the payment system and the backoffice. Using Code 55 instead of Code 56 allows for much more flexibility as the names of the fields do not need to be written, contrary to GraphQL where fields need to match [60].

```
FieldMask.FromFieldNumbers<PaymentResponse>(1,2);
```

Code 55 - Field Mask Definition Through Field Numbers

```
FieldMask fieldMask = new FieldMask();
fieldMask.Paths.Add("result");
fieldMask.Paths.Add("confirmation");
```

Code 56 - Manual Field Mask definition in C#

By using field numbers, we can see that requires much fewer changes as the numbers are automatically translated to the respective field no matter their name [61]. The deprecation of the field also does not give a warning of its usage.

gRPC application contrary to GraphQL, supports versioning in proto definitions. They are used by defining the attribute package and supplying it with a version number, such as “package Protos.V2”. This can then be compressed and uploaded to an online repository such as Nexus or Maven [62][63]. Subsequently, the client API must add this dependency to its project.

5.3 Results

This section presents and analyses the outcomes of the tested scripts across all methods. All presented metrics are of the response times measured in milli-seconds(ms).

5.3.1 Performance

This test verifies if the technology migration from GraphQL to gRPC impacts performance using the shown tests in section 5.2.1.

5.3.1.1 GetPartialParks

Table 16 - GetPartialParks Results

Technology	Average(ms)	Maximum (ms)	Median(ms)	Minimum (ms)	90th Percentile(ms)	95th Percentile(ms)
GraphQL	11,66	39,09	11,35	9,52	12,86	13,59
gRPC	9,66	125,54	8,99	7,52	10,95	11,82

In this table, we can see that although gRPC is quicker on average there are instances where it has some significant delays, as represented by the difference between the GraphQL and gRPC maximum values.

5.3.1.2 GetAllParks

Table 17 - GetAllParks Results

Technology	Average(ms)	Maximum (ms)	Median(ms)	Minimum (ms)	90th Percentile(ms)	95th Percentile(ms)
GraphQL	15,90	2146,82	12,53	10,51	17,32	19,39
gRPC Unary	11,02	593,59	10,09	8,57	13,98	15,69
Server Stream	13,05	41,40	12,45	11,04	15,07	16,57
Two-Sided Stream	13,45	224,40	12,38	11,02	14,89	16,31
Client Stream	11,00	256,00	10,07	8,90	12,24	13,12

Similar to the results from Table 16, gRPC is quicker than GraphQL but the extreme values still occur. As for the streaming implementation, we can see that they are on average slower but

more consistent. The server stream is the most consistent as it has a much lower peak than all others.

5.3.1.3 CreateUser

Table 18 - CreateUser Results

Technology	Average(ms)	Maximum(ms)	Median(ms)	Minimum(ms)	90th Percentile(ms)	95th Percentile(ms)
GraphQL	229,98	667,14	228,09	223,58	236,79	241,39
gRPC	243,76	1100,41	240,23	221,73	265,36	272,49

This create task shows that gRPC is slower than GraphQL although the minimum is slightly lower than the GraphQL one.

5.3.1.4 UpdateParkingValue

Table 19 - UpdateParkingValue Results

Technology	Average(ms)	Maximum(ms)	Median(ms)	Minimum(ms)	90th Percentile(ms)	95th Percentile(ms)
GraphQL	4,85	34,08	4,69	4,00	5,18	5,38
gRPC	1,76	25,95	1,52	1,00	2,17	2,38

Like the results we have seen so far, all values point to gRPC being quicker than GraphQL as Table 19 shows, that the gRPC implementation bests GraphQL in all metrics.

5.3.1.5 LeavePark

Table 20 - LeavePark Results

Technology	Average(ms)	Maximum (ms)	Median(ms)	Minimum (ms)	90th Percentile (ms)	95th Percentile (ms)
GraphQL	1058,11	3926,25	1055,93	1026,27	1072,97	1080,27
GraphQL backoffice-payment communication	10,26	386,36	10,29	1,31	16,21	16,93
gRPC	1069,55	1680,54	1069,37	8,74	1084,47	1089,87
gRPC backoffice-payment communication	26,16	225,74	25,81	12,60	33,27	36,10
Server Stream	1070,17	1753,23	1069,59	124,68	1084,95	1090,49

Server Stream backoffice-payment communication	25,98	170,01	25,67	11,94	33,49	36,67
Two-Sided Stream	1073,08	4699,02	1069,11	12,98	1086,04	1092,76
Two-Sided Stream backoffice-payment communication	26,11	1061,43	24,99	12,83	32,44	35,13
Client Stream	1069,06	1687,08	1067,95	2,07	1083,82	1089,41
Client Stream backoffice-payment communication	24,96	256,75	24,52	12,28	31,75	34,49

In this "LeavePark" test, we can see that GraphQL is quicker on average than every other implementation, but the server stream allows for a much more undeviating performance.

5.3.1.6 Performance Statistical Analysis

Before the analysis, for all CSVs, the necessary data was collected after applying data treatments. Throughout the whole testing process, was used a significance level of 0.05. All presented results and files are available in the respective GitHub repositories [48][50].

To perform both normality and asymmetry testing one loop was used to perform the calculation, as shown in Code 57.

```
for(value in ls()){
  #NORMALITY TEST
  assign(paste0(value, "Normality"), lillie.test(get(value))$p.value, envir
= .GlobalEnv)
  #ASYMMETRY  TEST
  assign(paste0(value,     "Asymmetry"),     skewness(get(value)),     envir
= .GlobalEnv)
  rm(value)
}
```

Code 57 - Normality and Asymmetry tests for all data sets

A normality test was performed to verify if all samples followed a normal distribution. In order to execute such a task, the Lilliefors test was used for 21 CSVs all had 3000 entries (less than or equal to 30 samples the Shapiro test would have been used) [64].

Table 21 and Table 22 represent the results of each normality check. Due to all collected values being smaller than the significance level we can conclude that all do not follow a normal distribution.

Table 21 - GraphQL CSV Normality test

Variable Name	**Normality Value (p-value)**
graphqlAllParksNormality	0
graphqlCreateUserNormality	0
graphqlLeaveParkNormality	0
graphqlLeaveParkPaymentNormality	$4{,}64\text{x}10^{-238}$
graphqlPartialParksNormality	$9{,}55 * 10^{-247}$
graphqlUpdateParkingValueNormality	0

Table 22 - gRPC CSV Normality test

Variable Name	Normality Value (p-value)
grpcAllParksClientStreamNormality	0
grpcAllParksNormality	0
grpcAllParksServerStreamNormality	0
grpcAllParksTwoSidedStreamNormality	0
grpcCreateUserNormality	0
grpcLeaveParkClientStreamNormality	0
grpcLeaveParkClientStreamPaymentNormality	$2{,}07\text{x}10^{-152}$
grpcLeaveParkNormality	$1{,}57\text{x}10^{-307}$
grpcLeaveParkPaymentNormality	$7{,}63\text{x}10^{-100}$
grpcLeaveParkServerStreamNormality	$1{,}29\text{x}10^{-311}$
grpcLeaveParkServerStreamPaymentNormality	$1{,}49\text{x}10^{-22}$
grpcLeaveParkTwoSideStreamNormality	0
grpcLeaveParkTwoSideStreamPaymentNormality	0
grpcPartialParksNormality	0
grpcUpdateParkingValueNormality	0

Just after the normality test came the asymmetry test. As Table 23 and Table 24 show, by listing all results for each file. All the values are asymmetric as they are higher than one or lower than -1. These values mean that all data set is either rightly skewed (mean higher than median) or left skewed (mean lower than median).

Table 23 - GraphQL CSV Asymmetry test

Variable Name	**Asymmetry Value**
graphqlAllParksAsymmetry	28,97
graphqlCreateUserAsymmetry	31,17
graphqlLeaveParkAsymmetry	49,23
graphqlLeaveParkPaymentAsymmetry	32,00
graphqlPartialParksAsymmetry	5,94
graphqlUpdateParkingValueAsymmetry	15,18

Table 24 - gRPC CSV Asymmetry test

Variable Name	Asymmetry Value
grpcAllParksAsymmetry	51,74
grpcAllParksClientStreamAsymmetry	17,54
grpcAllParksServerStreamAsymmetry	3,68
grpcAllParksTwoSidedStreamAsymmetry	15,93
grpcCreateUserAsymmetry	11,72
grpcLeaveParkAsymmetry	-16,49
grpcLeaveParkClientStreamAsymmetry	-12,13
grpcLeaveParkClientStreamPaymentAsymmetry	10,81
grpcLeaveParkPaymentAsymmetry	9,25
grpcLeaveParkServerStreamAsymmetry	-7,97
grpcLeaveParkServerStreamPaymentAsymmetry	4,45
grpcLeaveParkTwoSideStreamAsymmetry	30,78
grpcLeaveParkTwoSideStreamPaymentAsymmetry	30,71
grpcPartialParksAsymmetry	19,98
grpcUpdateParkingValueAsymmetry	13,31

Finally, the Wilcoxon test was used to perform a hypothesis test on both technologies [65]. Before the test execution, these hypotheses were defined:

- Null hypothesis (H0): there is no significant difference in performance on both technologies.
- Alternative hypothesis (H1): there is a significant difference in performance on both technologies.

These hypotheses were calculated with the help of R and the script shown in Code 58.

```
# GET ALL PARKS
grpcAllParks_graphqlAllParks<-    wilcox.test(grpcAllParks,graphqlAllParks,
paired=FALSE)$p.value
grpcAllParksClientStream_graphqlAllParks                                <-
wilcox.test(grpcAllParksClientStream,graphqlAllParks, paired=FALSE)$p.value
grpcAllParksServerStream_graphqlAllParks                                <-
wilcox.test(grpcAllParksServerStream,graphqlAllParks, paired=FALSE)$p.value
grpcAllParksTwoSidedStream_graphqlAllParks                              <-
wilcox.test(grpcAllParksTwoSidedStream,graphqlAllParks,
paired=FALSE)$p.value

# GET PARTIAL PARKS
grpcPartialParks_graphqlPartialParks                                    <-
wilcox.test(grpcPartialParks,graphqlPartialParks, paired=FALSE)$p.value

# CREATE USER
grpcCreateUser_graphqlCreateUser                                        <-
wilcox.test(grpcCreateUser,graphqlCreateUser, paired=FALSE)$p.value

# UPDATE PARKING VALUE
```

```
grpcUpdateParkingValue_graphqlUpdateParkingValue                    <-
wilcox.test(grpcUpdateParkingValue,graphqlUpdateParkingValue,
paired=FALSE)$p.value

# LEAVE PARK
grpcLeavePark_graphqlLeavePark                                      <-
wilcox.test(grpcLeavePark,graphqlLeavePark, paired=FALSE)$p.value
grpcLeaveParkClientStream_graphqlLeavePark                          <-
wilcox.test(grpcLeaveParkClientStream,graphqlLeavePark,
paired=FALSE)$p.value
grpcLeaveParkServerStream_graphqlLeavePark                          <-
wilcox.test(grpcLeaveParkServerStream,graphqlLeavePark,
paired=FALSE)$p.value
grpcLeaveParkTwoSideStream_graphqlLeavePark                         <-
wilcox.test(grpcLeaveParkTwoSideStream,graphqlLeavePark,
paired=FALSE)$p.value

# LEAVE PARK PAYMENT
grpcLeaveParkPayment_graphqlLeaveParkPayment                        <-
wilcox.test(grpcLeaveParkPayment,graphqlLeaveParkPayment,
paired=FALSE)$p.value
grpcLeaveParkClientStreamPayment_graphqlLeaveParkPayment            <-
wilcox.test(grpcLeaveParkClientStreamPayment,graphqlLeaveParkPayment,
paired=FALSE)$p.value
grpcLeaveParkServerStreamPayment_graphqlLeaveParkPayment            <-
wilcox.test(grpcLeaveParkServerStreamPayment,graphqlLeaveParkPayment,
paired=FALSE)$p.value
grpcLeaveParkTwoSideStreamPayment_graphqlLeaveParkPayment           <-
wilcox.test(grpcLeaveParkTwoSideStreamPayment,graphqlLeaveParkPayment,
paired=FALSE)$p.value
```

Code 58 - R Code for Hypothesis Testing

A table with all p-values compared values can also be seen in Table 25 for a more compact summary. Below is present a list of all Wilcox p-value comparisons that were done:

- GetAllParks:
  - gRPCAllParks and GraphQLAllParks: 0
  - gRPCAllParksClientStream and GraphQLAllParks: 0
  - gRPCAllParksServerStream and GraphQLAllParks: $0{,}21$
  - gRPCAllParksTwoSidedStream and GraphQLAllParks: $0{,}35$
- GetPartialParks:
  - gRPCPartialParks and GraphQLPartialParks: 0
- CreateUser:
  - gRPCCreateUser and GraphQLCreateUser: $2{,}67x10^{-51}$
- UpdateParkingValue
  - gRPCUpdateParkingValue and GraphQLUpdateParkingValue: 0
- LeavePark:
  - Full Communication:
    - gRPCLeavePark and GraphQLLeavePark: $6{,}51x10^{-246}$
    - gRPCLeaveParkClientStream and GraphQLLeavePark: $3{,}87x10^{-219}$
    - gRPCLeaveParkServerStream and GraphQLLeavePark: $1{,}10x10^{-263}$

        - gRPCLeaveParkTwoSidedStream and GraphQLLeavePark: $1{,}64x10^{-263}$
    - Payment Service Communication:
        - gRPCLeaveParkPayment and GraphQLLeaveParkPayment: 0
        - gRPCLeaveParkPayment and GraphQLLeaveParkPayment: 0
        - gRPCLeaveParkPayment and GraphQLLeaveParkPayment: 0
        - gRPCLeaveParkPayment and GraphQLLeaveParkPayment: 0

Due to the significance level being 0.05, all but gRPCAllParksServerStream and gRPCAllParksTwoSidedStream reject the null hypothesis. However, CreateUser and LeavePark were significantly slower on gRPC than GraphQL.

5.3.2 Flexibility

Both technologies have breaking changes, and both support versioning. However, GraphQL does not recommend its usage. Instead, a version-less schema is the technology's focus. This option causes an issue where maintainability will eventually become a bigger problem in deprecating old fields. They first need to annotate the fields with @deprecated and subsequently check if the depending APIs no longer use that field. Due to those reasons a developerOpinion of 3 is given to this technology.

As for gRPC and Protobuf, both approaches can be used, but the capability to select fields by their field number lowers the chance of breaking changes when renaming an attribute. This ability allows it to handle changes better than GraphQL. A potential way to improve versioning management is to have them as dependencies in a repository such as Nexus or Maven [62][63]. Both technologies, from a backend-to-backend perspective, require recompilation if the schema is modified. Because of these reasons a developerOpinion of 4 is given to gRPC and Protobuf.

5.4 Summary

Performance was the first characteristic that was analysed. The study results validate if there is indeed an increase in speed when using gRPC instead of GraphQL, as section 2.1 mentioned. An investigation was performed, to authenticate these claims with the assistance of the K6 tool in section 5.2.1. Which then resulted in a comparative analysis in section 5.3.1. There was a significant decrease in latency, and the maximum delay was significantly reduced, especially in the streaming implementations.

Secondly, the flexibility analysis was done. To accomplish this test changes were made to both GraphQL and gRPC applications, which subsequently were studied in section 5.2.2 and compared in section 5.3.2. The technologies were mostly similar but gRPC stood out with the possibility to get the fields by their ID, which allowed for more changes without causing issues

to the relevant applications. GraphQL received a classification of 3, and as for gRPC and Protobuf it received a grade of 4. This was due to the recompilation requirement when modifications to the source code were performed.

This analysis confirmed that using gRPC and Protobuf improved both the performance and the flexibility. However, the migration duration was the downside, as the GraphQL implementation was done in just a few hours.

6 Conclusions

This section summarises what was performed, threats to validity, future work, contributions, and a personal note about the aftermath of the developed project.

6.1 Goals Achieved

This document lists all stages done to evaluate the effectiveness of migrating from GraphQL to gRPC and Protobuf in the performance and flexibility categories.

As previously outlined in section 1.2, to achieve this study some objectives were outlined:

- **Study what API flexibility is, what patterns improve flexibility and what API performance means:** Research on the identified relevant questions was performed in sections 3.2, 3.3 and 3.4. Flexibility is defined as "the ability to change or be changed easily" without undergoing a whole replacement of the source code. A list of the discovered patterns is present in the document. The meaning of performance is defined as "the speed, accuracy, efficiency, and adaptability of systems" achieving its goal.
- **Migrate the Application from GraphQL to gRPC and Protobuf:** When selecting the software to migrate, due to time constraints to complete this work, it was given precedence to previous work where the knowledge of an already developed application, allowed for quicker development, as shown in section 4. After that, the migration began, intending to keep the application as untampered as possible, as shown in sections 0 and 4.3, where it was possible to maintain all patterns used in the original application.
- **Assessment of the results:** To assess the outcomes, we used the GQM (Goal Question Metric) approach to select the necessary metrics for evaluation. These metrics were measured using the k6 tool introduced on page 82, and through a subjective analysis by the author, it was identified that they assess the elements to analyse performance and flexibility. For the performance review, we developed multiple test scripts in JavaScript and executed them using k6. As for the flexibility evaluation, modifications were done for both GraphQL, and gRPC and Protobuf applications and a subsequent comparison of the effects. Each application was graded using the developerOpinion metric. GraphQL received a 3, and gRPC and Protobuf a 4.

In conclusion, we successfully addressed the questions regarding performance and flexibility. Field masks, specifically those using field numbers, allow for improved flexibility. As for the performance, as seen in 5.3.1, it indicates that gRPC has lower latency values than GraphQL, with special notice to the streaming maximum values that showed a substantial improvement.

6.2 Threats to Validity

During the project's development, concerns were raised about potential threats to its validity. These concerns could influence results.

Using a pre-selected project could influence the results attained when using other projects that employ other coding languages.

The reliance on a single machine, with somewhat older hardware, to perform the load tests. Different and varying conditions could allow for different results.

Due to GraphQL, gRPC and Protobuf technological differences, it is impossible to determine where the performance improvement comes from the technology change or the whole capabilities it encompasses, such as HTTP 2.0.

An unsuccessful attempt was made to isolate the systems using docker. Isolating the system to perform tests inside docker could also influence the results.

K6 implementation of gRPC streaming is an experimental feature and could contain faults, which can cause additional delays that could affect the collected metrics.

6.3 Future Work

We analysed the advantages and disadvantages of using gRPC in the application during the project. An opportunity for potential investigation could be how it also affects the frontend application with its HTTP 2.0 support and whether it should be considered a replacement to GraphQL in backend-to-frontend applications. Frontend support is available through gRPC-web, which can help choose which technology to use [66].

6.4 Contributions

This document was created for the master's thesis at ISEP to study the impact of migrating from GraphQL to gRPC and Protobuf on the performance and flexibility of the selected application.

The previously introduced selected application underwent a migration from GraphQL to gRPC. The objective of this project was to evaluate both flexibility and performance between the technologies and determine what their respective impacts were. This study showed that gRPC seems to be a viable option. It brings upsides, being a performance increase, streaming support, and the ability to rename fields without breaking the API contract, thanks to the field mask capability, "FromFieldNumbers", but on the downside, the learning curve and the migration to Protobuf might be some deterrents to the technology.

All repositories that were used in the making of this project are available on GitHub with an MIT license. There are three repositories: one with the original application developments [40],

another with the GraphQL implementation [48], and the third with the gRPC and Protobuf development [50]. These are free to be used for testing, improvements, or to perform any other study.

6.5 Personal Appreciation

The author found that the journey allowed for personal and professional development. The project allowed the author room to enhance skills and expertise in software engineering. This experience has been instrumental in shaping the author's professional growth and will continue to impact the professional career positively.

Bibliography References

[1] A. Borysov and R. Gardiner, “Practical API Design at Netflix, Part 1: Using Protobuf FieldMask | by Netflix Technology Blog | Netflix TechBlog.” Accessed: Nov. 28, 2023. [Online]. Available: https://netflixtechblog.com/practical-api-design-at-netflix-part-1-using-protobuf-fieldmask-35cfdc606518

[2] Google, “grpc - Explore - Google Trends.” Accessed: Dec. 21, 2023. [Online]. Available: https://trends.google.com/trends/explore?date=2018-12-01%202023-12-14&q=grpc&hl=en-GB

[3] V. Talwar, “gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments | Google Cloud Blog.” Accessed: Oct. 28, 2023. [Online]. Available: https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments

[4] L. Safran, “Error handling,” *gRPC*. [Online]. Available: https://grpc.io/docs/guides/error/

[5] X. Wang, “Authentication,” *gRPC*. [Online]. Available: https://grpc.io/docs/guides/auth/

[6] Instituto Politécnico do Porto. [n.d.]. Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto, “Diário da República, 2.ª série PARTE E Artigo 2.º”, Accessed: Dec. 17, 2023. [Online]. Available: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf

[7] D. Gotterbarn, K. Miller, and S. Rogerson, “Software engineering code of ethics,” *Commun ACM*, vol. 40, no. 11, Nov. 1997, doi: 10.1145/265684.265699.

[8] T. Zhen Yong, “Introduction to gRPC | gRPC.” Accessed: May 11, 2024. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/

[9] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2),” May 2015, doi: 10.17487/RFC7540.

[10] D. Castro, “Protocol Buffers Documentation.” Accessed: May 09, 2024. [Online]. Available: https://protobuf.dev/

[11] D. Castro, “Overview | Protocol Buffers Documentation.” Accessed: Nov. 01, 2023. [Online]. Available: https://protobuf.dev/overview/

[12] G. Hotelling and E. Anderson, “FAQ | gRPC.” Accessed: Oct. 28, 2023. [Online]. Available: https://grpc.io/docs/what-is-grpc/faq/

[13] P. Chalin, “About gRPC | gRPC.” Accessed: Oct. 28, 2023. [Online]. Available: https://grpc.io/about/

[14] M. Damiani, “Rest.li - A framework for building RESTful architectures at scale.” Accessed: Mar. 16, 2024. [Online]. Available: https://linkedin.github.io/rest.li/

[15] R. Gancarz, “Why LinkedIn chose gRPC+Protobuf over REST+JSON: Q&A with Karthik Ramgopal and Min Chen - InfoQ.” Accessed: Mar. 16, 2024. [Online]. Available: https://www.infoq.com/news/2023/12/linkedin-grpc-protobuf-rest-json/

[16] K. Silz, “QCon London: gRPC Migration Automation at LinkedIn - InfoQ.” Accessed: Apr. 13, 2024. [Online]. Available: https://www.infoq.com/news/2024/04/qcon-london-grpc-linkedin/

[17] husobee, “REST v. gRPC.” Accessed: Dec. 21, 2023. [Online]. Available: https://husobee.github.io/golang/rest/grpc/2016/05/28/golang-rest-v-grpc.html

[18] K. Indrasiri and D. Kuruppu, “GRPC : up and running: building cloud native applications with Go and Java for docker and kubernetes,” 2020, Accessed: Dec. 21, 2023. [Online]. Available: https://www.oreilly.com/library/view/grpc-up-and/9781492058328/

[19] J. Geewax, “API Design Patterns,” pp. 18–22, 2021, Accessed: Dec. 21, 2023. [Online]. Available: https://www.manning.com/books/api-design-patterns

[20] B. Kuo, “Queries and Mutations | GraphQL.” Accessed: Nov. 28, 2023. [Online]. Available: https://graphql.org/learn/queries/#directives

[21] M. J. Page *et al.*, “Updating guidance for reporting systematic reviews: development of the PRISMA 2020 statement,” *J Clin Epidemiol*, vol. 134, pp. 103–112, Jun. 2021, doi: 10.1016/j.jclinepi.2021.02.003.
[22] M. J. Page *et al.*, “The PRISMA 2020 statement: an updated guideline for reporting systematic reviews,” *Syst Rev*, vol. 10, no. 1, p. 89, 2021, doi: 10.1186/s13643-021-01626-4.
[23] Dictionary, “Synonyms and Antonyms of Words | Thesaurus.com.” Accessed: Dec. 21, 2023. [Online]. Available: https://www.thesaurus.com/
[24] Cambridge University Press & Assessment, “FLEXIBILITY | English meaning - Cambridge Dictionary.” Accessed: Jun. 29, 2024. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/flexibility
[25] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, “API designers in the field: Design practices and challenges for creating usable APIs,” *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, vol. 2018-October, pp. 249–258, Oct. 2018, doi: 10.1109/VLHCC.2018.8506523.
[26] S. K. Mukhiya, F. Rabbi, V. K. I Pun, A. Rutle, and Y. Lamo, “A GraphQL approach to Healthcare Information Exchange with HL7 FHIR,” *Procedia Comput Sci*, vol. 160, pp. 338–345, Jul. 2019, doi: 10.1016/j.procs.2019.11.082.
[27] D. Walsh, *Podman in action secure, rootless containers for Kubernetes, microservices, and more*. 2023. Accessed: Dec. 22, 2023. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edsebk&AN=3520272&site=eds-live
[28] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker, “Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles,” in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, in EuroPLop ’19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3361149.3361164.
[29] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, and M. Stocker, “Interface Responsibility Patterns: Processing Resources and Operation Responsibilities,” in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, in EuroPLoP ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3424771.3424822.
[30] S. Baslé, “Annotated Controllers :: Spring Framework.” Accessed: Dec. 08, 2023. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller.html
[31] O. Zimmermann, C. Pautasso, D. Lübke, U. Zdun, and M. Stocker, “Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources,” in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, in EuroPLoP ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3424771.3424821.
[32] L. Rinaldi, M. Torquati, D. De Sensi, G. Mencagli, and M. Danelutto, “Improving the Performance of Actors on Multi-cores with Parallel Patterns,” *Int J Parallel Program*, vol. 48, no. 4, pp. 692–712, 2020, doi: 10.1007/s10766-020-00663-1.
[33] M. Nassif and M. P. Robillard, “Wikifying software artifacts,” *Empir Softw Eng*, vol. 26, no. 2, p. 31, 2021, doi: 10.1007/s10664-020-09918-4.
[34] S. Yang, S. Son, M.-J. Choi, and Y.-S. Moon, “Performance improvement of Apache Storm using InfiniBand RDMA,” *J Supercomput*, vol. 75, no. 10, pp. 6804–6830, 2019, doi: 10.1007/s11227-019-02905-7.
[35] C. Garbin, X. Zhu, and O. Marques, “Dropout vs. batch normalization: an empirical study of their impact to deep learning,” *Multimed Tools Appl*, vol. 79, no. 19, pp. 12777–12815, 2020, doi: 10.1007/s11042-019-08453-9.
[36] A. Ambasht, “API Integration using GraphQL,” *International Journal of Computer Trends and Technology*, vol. 71, no. 8, pp. 28–33, Aug. 2023, doi: 10.14445/22312803/IJCTT-V71I8P104.
[37] A. Borysov and R. Gardiner, “Practical API Design at Netflix, Part 2: Protobuf FieldMask for Mutation Operations | by Netflix Technology Blog | Netflix TechBlog.” Accessed: Jun. 23, 2024. [Online]. Available: https://netflixtechblog.com/practical-api-design-at-netflix-part-2-protobuf-fieldmask-for-mutation-operations-2e75e1d230e4
[38] gRPC, “GRPC Core: gRPC Versioning Guide.” Accessed: Jun. 23, 2024. [Online]. Available: https://grpc.github.io/grpc/core/md_doc_versioning.html

[39] D. POSTOLOV, “GraphQL Best Practices | GraphQL.” Accessed: Jun. 23, 2024. [Online]. Available: https://graphql.org/learn/best-practices/#versioning
[40] L. Silva, “GitHub - 1181479/Park_Management_System.” Accessed: May 20, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System
[41] L. Silva, “Park_Management_System/Documentation.md at main · 1181479/Park_Management_System · GitHub.” Accessed: May 26, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System/blob/main/Documentation.md
[42] ChiliCream, “Introduction - Hot Chocolate - ChilliCream GraphQL Platform.” Accessed: Apr. 06, 2024. [Online]. Available: https://chillicream.com/docs/hotchocolate/v13
[43] A. Rose, I. Maximov, and S. Krueger, “GitHub - graphql-dotnet/graphql-client: A GraphQL Client for .NET Standard.” Accessed: Apr. 06, 2024. [Online]. Available: https://github.com/graphql-dotnet/graphql-client
[44] L. Silva, “added graphql to backoffice · 1181479/Park_Management_System_GraphQL@5f91fcc · GitHub.” Accessed: Apr. 06, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GraphQL/commit/5f91fcc6851c111fc9fd0745ec51987133e5ff5b#diff-ab3b5849cf76978fc155344e546af472e990699d6de5b464a1a87ad6a7f06008
[45] ChiliCream, “DataLoader - Hot Chocolate - ChilliCream GraphQL Platform.” Accessed: Apr. 06, 2024. [Online]. Available: https://chillicream.com/docs/hotchocolate/v13/fetching-data/dataloader
[46] L. Silva, “Park_Management_System_GraphQL/Park20.Backoffice.Api/Park20.Backoffice.Application/Services/PaymentService.cs at main · 1181479/Park_Management_System_GraphQL · GitHub.” Accessed: Apr. 06, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GraphQL/blob/main/Park20.Backoffice.Api/Park20.Backoffice.Application/Services/PaymentService.cs#L265-L282
[47] ChiliCream, “Get started with Strawberry Shake in a Console application - Strawberry Shake - ChilliCream GraphQL Platform.” Accessed: Apr. 06, 2024. [Online]. Available: https://chillicream.com/docs/strawberryshake/v13/get-started/console
[48] L. Silva, “GitHub - 1181479/Park_Management_System_GraphQL.” Accessed: Apr. 06, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GraphQL
[49] L. Silva, “Park_Management_System_GRPC/Park20.Backoffice.Api/Park20.Backoffice.Api/Protos at main · 1181479/Park_Management_System_GRPC · GitHub.” Accessed: Apr. 07, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GRPC/tree/main/Park20.Backoffice.Api/Park20.Backoffice.Api/Protos
[50] L. Silva, “GitHub - 1181479/Park_Management_System_GRPC.” Accessed: Apr. 14, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GRPC/
[51] V. R. Basili, G. Caldiera, and H. D. Rombach, “THE GOAL QUESTION METRIC APPROACH”.
[52] Grafana Labs, “Load testing for engineering teams | Grafana k6.” Accessed: Apr. 17, 2024. [Online]. Available: https://k6.io/
[53] Postman, “Postman API Platform | Sign Up for Free.” Accessed: Apr. 18, 2024. [Online]. Available: https://www.postman.com/
[54] ChiliCream, “Introduction - Banana Cake Pop - ChilliCream GraphQL Platform.” Accessed: Apr. 18, 2024. [Online]. Available: https://chillicream.com/docs/bananacakepop/v2
[55] Grafana Labs, “Test lifecycle.” Accessed: Apr. 18, 2024. [Online]. Available: https://k6.io/docs/using-k6/test-lifecycle/
[56] B. Coleman, “GitHub - benc-uk/k6-reporter: Output K6 test run results as formatted & easy to read HTML reports.” Accessed: Apr. 18, 2024. [Online]. Available: https://github.com/benc-uk/k6-reporter
[57] L. Silva, “added flexibility test · 1181479/Park_Management_System_GraphQL@a8fdc2d · GitHub.” Accessed: Apr. 21, 2024. [Online]. Available:

https://github.com/1181479/Park_Management_System_GraphQL/commit/a8fdc2d3aacf3c6ca45df675ff3eca06c2eb495c
[58] L. Silva, “added deprecation test and name changing test · 1181479/Park_Management_System_GraphQL@fdedc48 · GitHub.” Accessed: Apr. 21, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GraphQL/commit/fdedc4867a46b131e2a3963906f8a9e131e556f5#diff-23b27d3970ddbe5f0f02ca7eaa35c9b1addded07954ff004849d097ed55ccda7
[59] D. POSTOLOV, “GraphQL Best Practices | GraphQL.” Accessed: May 25, 2024. [Online]. Available: https://graphql.org/learn/best-practices/#versioning
[60] L. Silva, “added flexibility test · 1181479/Park_Management_System_GRPC@4675c53 · GitHub.” Accessed: Apr. 21, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GRPC/commit/4675c537bc718b4b81e97e5b8a1e5593d5d762ea
[61] L. Silva, “added deprecation test and parameter change test · 1181479/Park_Management_System_GRPC@56e2ff1 · GitHub.” Accessed: Apr. 21, 2024. [Online]. Available: https://github.com/1181479/Park_Management_System_GRPC/commit/56e2ff10845bce3738467fa48b06369a61534912
[62] Nexus, “Nexus Repository Manager.” Accessed: May 19, 2024. [Online]. Available: https://repository.apache.org/#welcome
[63] Maven, “Maven Repository: Search/Browse/Explore.” Accessed: May 19, 2024. [Online]. Available: https://mvnrepository.com/
[64] H. Abdi and P. Molin, “(No Title).” [Online]. Available: http://www.utd.edu/
[65] A. Soetewey, “Wilcoxon test in R: how to compare 2 groups under the non-normality assumption? - Stats and R.” Accessed: Jun. 03, 2024. [Online]. Available: https://statsandr.com/blog/wilcoxon-test-in-r-how-to-compare-2-groups-under-the-non-normality-assumption/
[66] L. Costea, “Basics tutorial | Web | gRPC.” Accessed: Apr. 27, 2024. [Online]. Available: https://grpc.io/docs/platforms/web/basics/
[67] Postman, “What is Postman? Postman API Platform.” Accessed: May 15, 2024. [Online]. Available: https://www.postman.com/product/what-is-postman/

Appendix A Project Plan

Gantt diagram to represent all tasks that were performed during the project.

	Month					
Activity	January	February	March	April	May	June
Set Selection Criteria and Open Source Project Search						
Define criteria for project selection						
Conduct a search for potential projects						
Project Selection and Initial Analysis						
Evaluate and select a suitable open-source project						
Perform an initial analysis of the project's codebase						
Approval of Project						
Seek teachers approval based on the initial analysis						
As-Is To-Be Comparison between Project and Possible Solution						
Analyze the current state of the project						
Design a solution						
Implementing the Solution						
Begin coding and building the proposed solution						
Testing and Debugging						
Conduct comprehensive testing of the implemented solution						
Documentation						
Document the implemented solution						
Experimentation Preparation						
Define experimental parameters and metrics						
Experimentation, Data Collection, and Analysis						
Experiment on project						
Collect relevant data						
Analyze collected data against predefined metrics						
Compare collected data with original project						
Final Documentation and Report						
Document the entire process, including experimentation and results						

Figure 15 - Gantt Chart Diagram

Appendix B Authorization to use the Selected Project

As required it was requested the authorization of all contributors in the project construction. Following is their approval for the project usage in this work. To respect their privacy the images have been removed.

Appendix C Other Tools and Technologies

The technologies depicted in this section have the aim of introducing tools that are used along the project.

Postman

Postman is an API Platform to help build and test applications. This application allows testing for the following request types: HTTP, GraphQL, gRPC, WebSocket, Socket.IO and MQTT. Developers can create requests, define parameters, and handle authentication, simplifying the API development process. Additionally, Postman facilitates collaboration among team members through features like sharing collections, collaborating on tests, and adding comments on requests. It also has documentation generation capabilities to automate the creation of API documentation, thus speeding up the development and understanding process of new tests [67].

Hot Chocolate

Hot Chocolate is an open-source C# implementation of a GraphQL server. It allows software developers to add and change GraphQL APIs with ease. Some features are schema-first development, runtime schema corrections, subscriptions, authentication and authorization support, query batching and many other functionalities. The main goal of this open-source GraphQL server is to offer a flexible and powerful solution for a GraphQL server that incorporates .NET applications without much difficulty [42][45].

Strawberry Shake

As for Strawberry Shake, it's an open-source GraphQL client that can be used in tandem with Hot Chocolate to enhance .NET applications further with GraphQL capabilities. It delivers the means to connect and consume GraphQL APIs efficiently. This library overhauls and simplifies the entire communication process. It supplies features like query batching, automatic serialization/deserialization and type-safe querying using generated C# classes constructed on GraphQL schemas [47].

K6

K6 was chosen to perform load tests since it supported both GraphQL, unary gRPC and gRPC streaming. This application is an open-source load-testing tool used predominantly for API performance testing. It allows for traffic simulation and analysis of the system status during various load conditions.

K6 is used when executing test scripts in JavaScript. It allows for complex scenarios, such as test scenarios with dynamic data and authentication. This tool also collects and provides detailed metrics of system performance. These metrics then allow developers to identify performance issues and execute the required changes, such as ensuring the reliability and scalability of their applications. Furthermore, it offers integrations with various CI/CD tools and platforms, making it an effective tool for performance testing [52].

K6 GraphQL Test

Code 59 shows a JavaScript file to find the title of the first issue from K6 using GraphQL. It is possible by providing an access token from GitHub, setting in the request headers, and using the query “FindFirstIssue”. After the request is sent the script sleeps for a third of a second.

```
import http from "k6/http";
import { sleep } from "k6";

const accessToken = "YOUR_GITHUB_ACCESS_TOKEN";

export default function () {
  const query = `
    query FindFirstIssue {
      repository(owner:"grafana", name:"k6") {
        issues(first:1) {
          edges {
            node {
              title
            }
          }
        }
      }
    }`;

  const headers = {
    Authorization: `Bearer ${accessToken}`,
    "Content-Type": "application/json",
  };
```

```
http.post(
    "https://api.github.com/graphql",
    JSON.stringify({ query: query }),
    { headers: headers }
  );
  sleep(0.3);
}
```

Code 59 - GraphQL Example K6 Test [52]

K6 gRPC Test

gRPC example Code 60 firstly loads the “hello.proto” file, which contains the required definitions and is used when executing. In a JavaScript file, a hello request with the Bert name is sent to the server, after which the return is written out on the executing console, and finally, the connection is closed.

```
import grpc from 'k6/net/grpc'
import { sleep, check } from 'k6'

const client = new grpc.Client()
client.load(['definitions'], 'hello.proto')

export default function () {
  client.connect('grpcb.in:9001', { plaintext: false })

  const data = { greeting: 'Bert' }
  const response = client.invoke('hello.HelloService/SayHello', data)
  check(response, {
    'status is OK': (r) => r && r.status === grpc.StatusOK,
  })
  console.log(JSON.stringify(response.message))

  client.close()
  sleep(1)
}
```

Code 60 - gRPC Example K6 Test [52]

Appendix D GetAllParks Implementations

The following diagrams and code scripts represent implementations of GetAllParks for GraphQL, and gRPC and Protobuf.

The Figure 16, Code 61 and Code 62 show the GraphQL application.

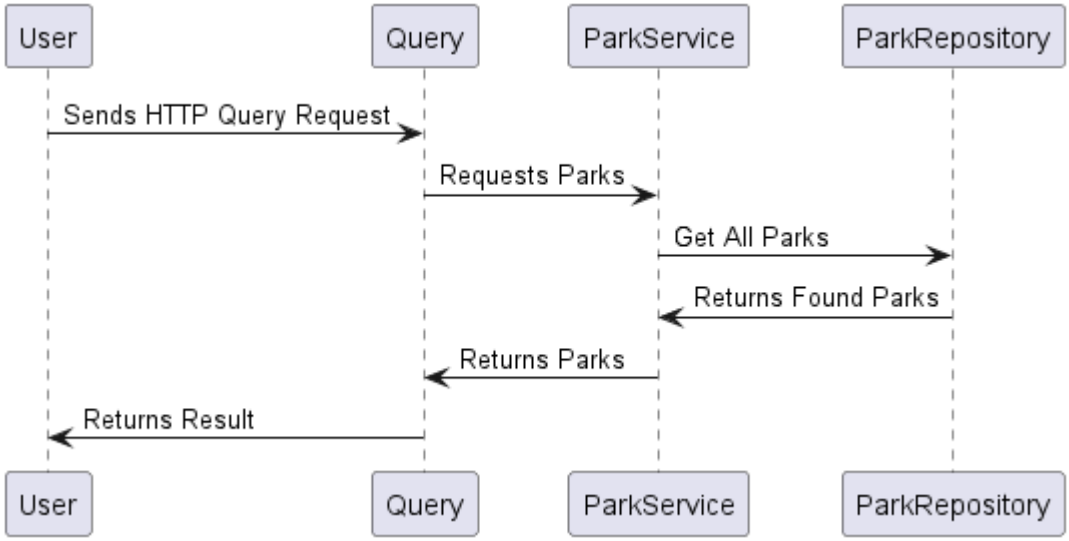

Figure 16 - Get All Parks GraphQL

```
const query = `
query AllParks {
  allParks {
      id
      numberFloors
      parkName
      latitude
      longitude
      location
      openingTime
      closingTime
      isActive
      nightFee
      priceTable {
          priceTableId
          initialDate
          linePrices {
              linePriceTableId
              period {
                  periodId
                  initialTime
                  finalTime
                  fractions {
                      fractionId
                      order
```

```
                    minutes
                    vehicleType
                    price
                }
            }
        }
    }
    parkingSpots {
        parkingSpotId
        vehicleType
        status
        floorNumber
    }
  }
}
`;
```

Code 61 - Query definition for GraphQL GetAllParks

```
export default function () {
  const headers = {
    "Content-Type": "application/json",
  };

  const res = http.post(
    "http://localhost:5000/graphql",
    JSON.stringify({ query: query }),
    {
      headers: headers,
    }
  );
}
```

Code 62 - Test function for GraphQL GetAllParks

The Figure 17 and Code 63 show the gRPC application.

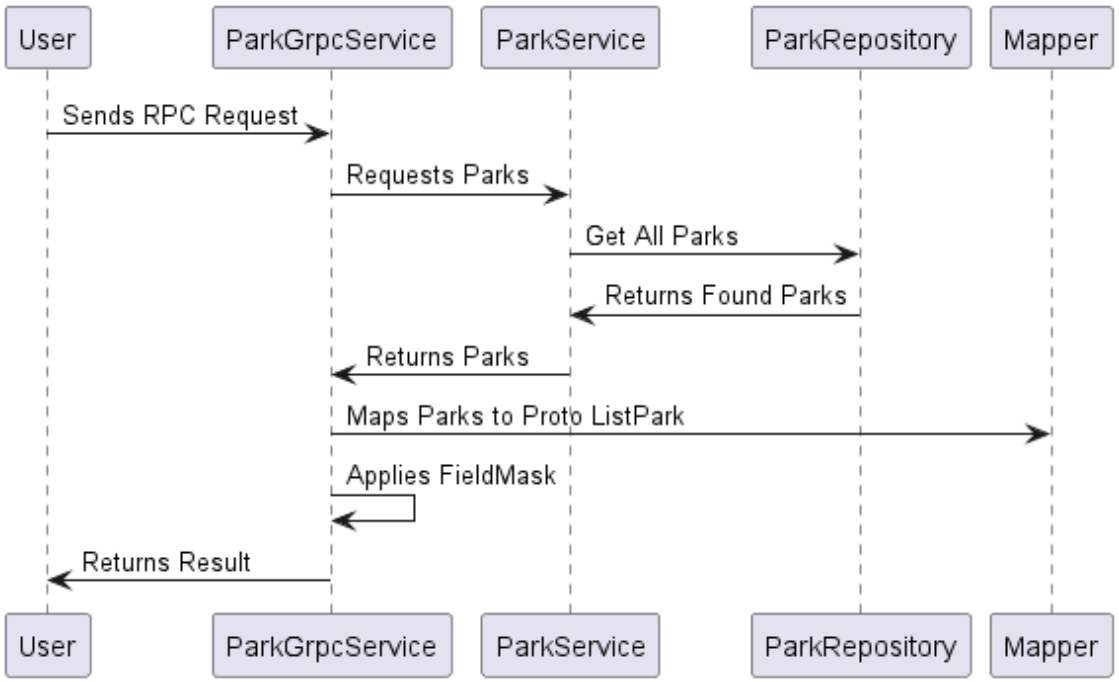

Figure 17 - Get All Parks Unary

```
export default () => {
  client.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  const req = { fieldMask: "parks" };

  const response = client.invoke("Proto.ParkGrpc/GetAllParks", req);

  check(response, {
    "status is OK": (r) => r && r.status === grpc.StatusOK,
  });

  client.close();
};
```

Code 63 - Test function for gRPC GetAllParks

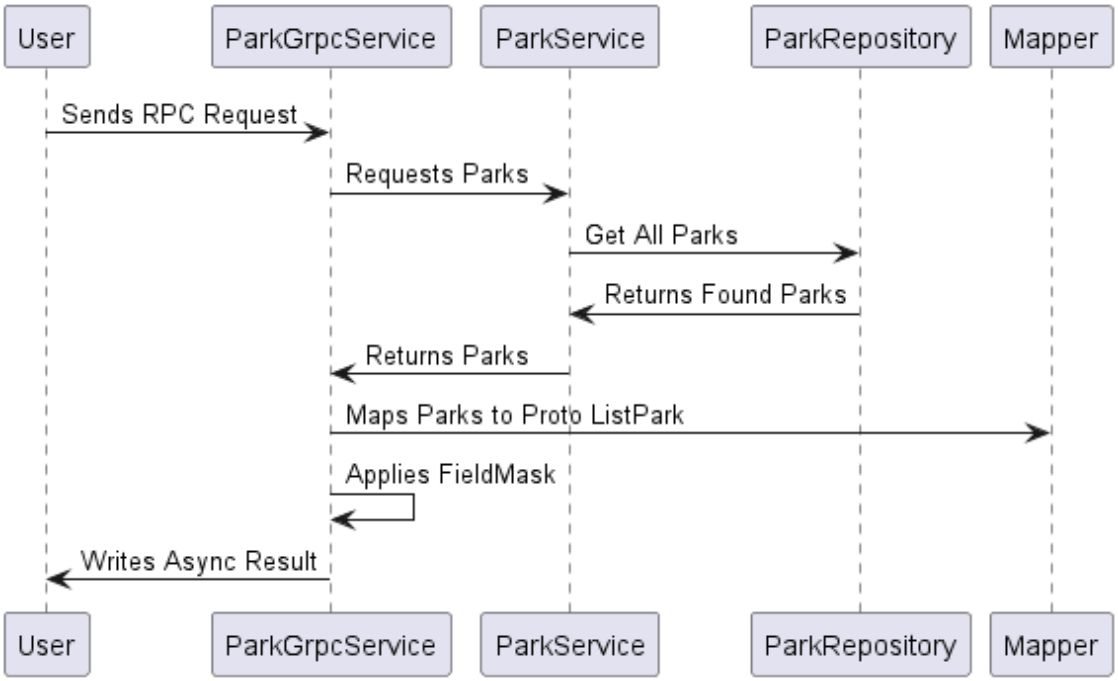

Figure 18 - Get All Parks Server Stream

The Figure 19 and Code 64 show the gRPC two sided stream application.

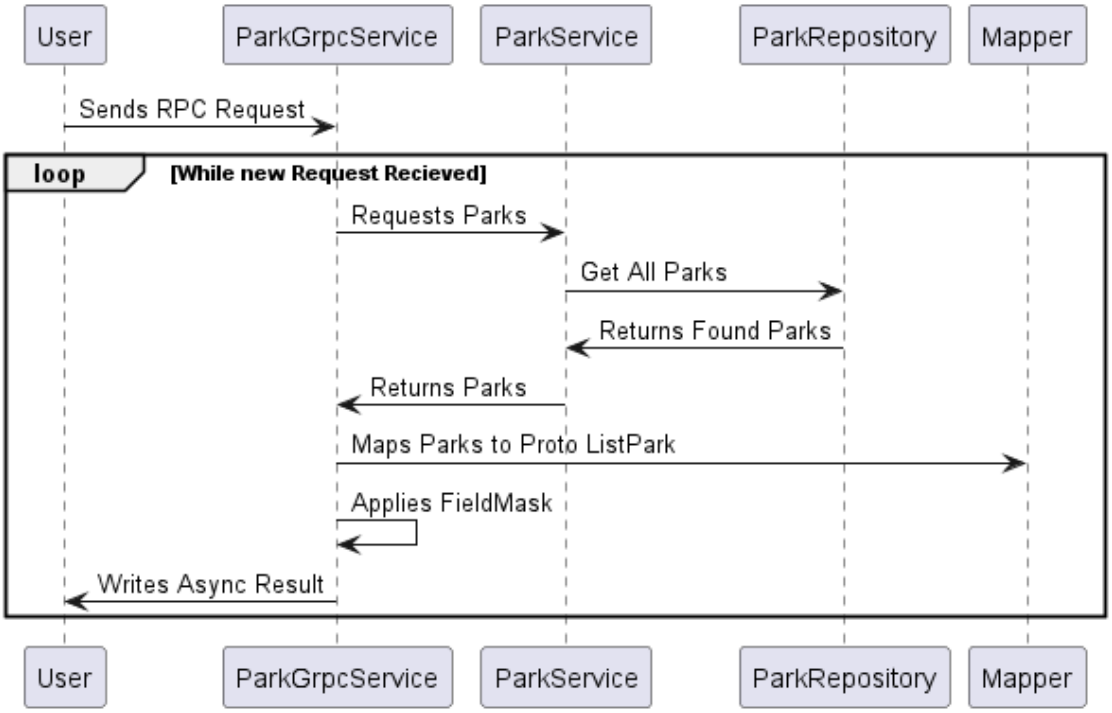

Figure 19 - Get All Parks Two Sided Stream

```
export default () => {
  client.connect("localhost:7000", {});
```

```
  // Create the payload with the fields and field mask
  const req = { fieldMask: "parks" };
  const stream = new Stream(client,
"Proto.ParkGrpc/GetAllParksTwoSidedStream");

  stream.on("end", function () {
    // The server has finished sending
    client.close();
    console.log("All done");
  });

  // send a message to the server
  stream.write(req);
  stream.end();
};
```

Code 64 - Test function for gRPC GetAllParksTwoSidedStream

The Figure 20 and Code 65 show the gRPC client stream application.

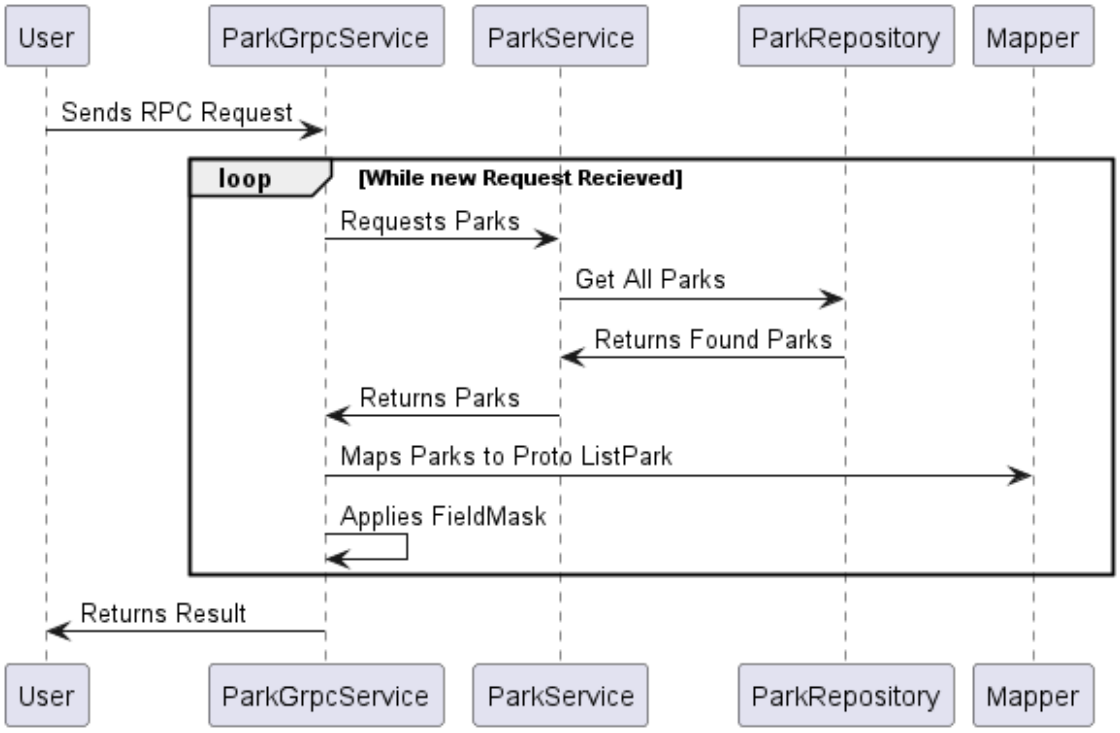

Figure 20 - Get All Parks Client Stream

```
export default () => {
  client.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  const req = { fieldMask: "parks" };
  const stream = new Stream(client,
"Proto.ParkGrpc/GetAllParksClientStream");

  stream.on("end", function () {
    // The server has finished sending
    client.close();
```

```
    console.log("All done");
  });

  // send a message to the server
  stream.write(req);
  stream.end();
};
```

Code 65 - Test function for gRPC GetAllParksClientStream

Appendix E CreateUser Sequence Diagrams

The following diagrams and code scripts represent implementations of GetAllParks for GraphQL, and gRPC and Protobuf.

The Figure 21, Code 66 and Code 67 show the GraphQL application.

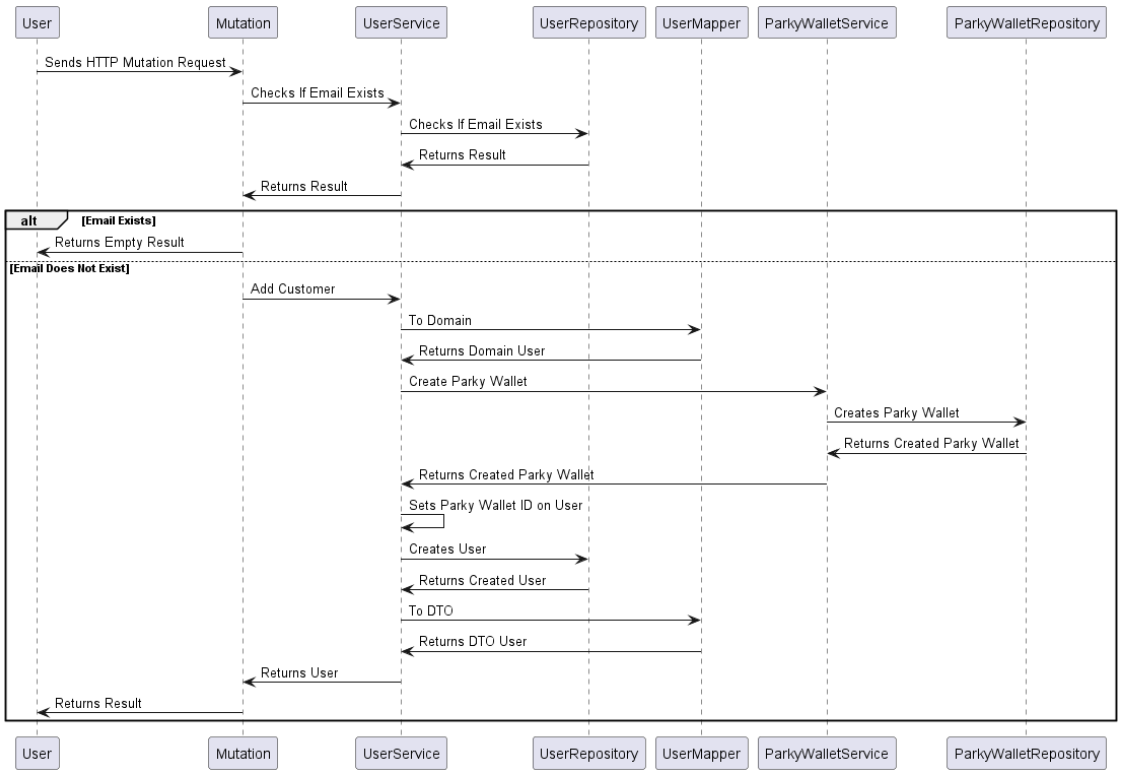

Figure 21 - Create User GraphQL

```
const query = `
mutation ($input: CreateCustomerRequestDtoInput!) {
  addCustomer(createCustomerRequestDto: $input) {
    name
    email
    username
  }
}
`;
```

Code 66 - Mutation query for GraphQL CreateUser

```
export default function () {
  const headers = {
    "Content-Type": "application/json",
  };

  let value = randomString(20);
```

```
  const res = http.post(
    "http://localhost:5000/graphql",
    JSON.stringify({
      query: query,
      variables: {
        input: {
          name: value,
          password: value,
          email: value,
          username: value,
        },
      },
    }),
    {
      headers: headers,
    }
  );
}
```

Code 67 - Test function for GraphQL CreateUser

The Figure 22 and Code 68 show the gRPC application.

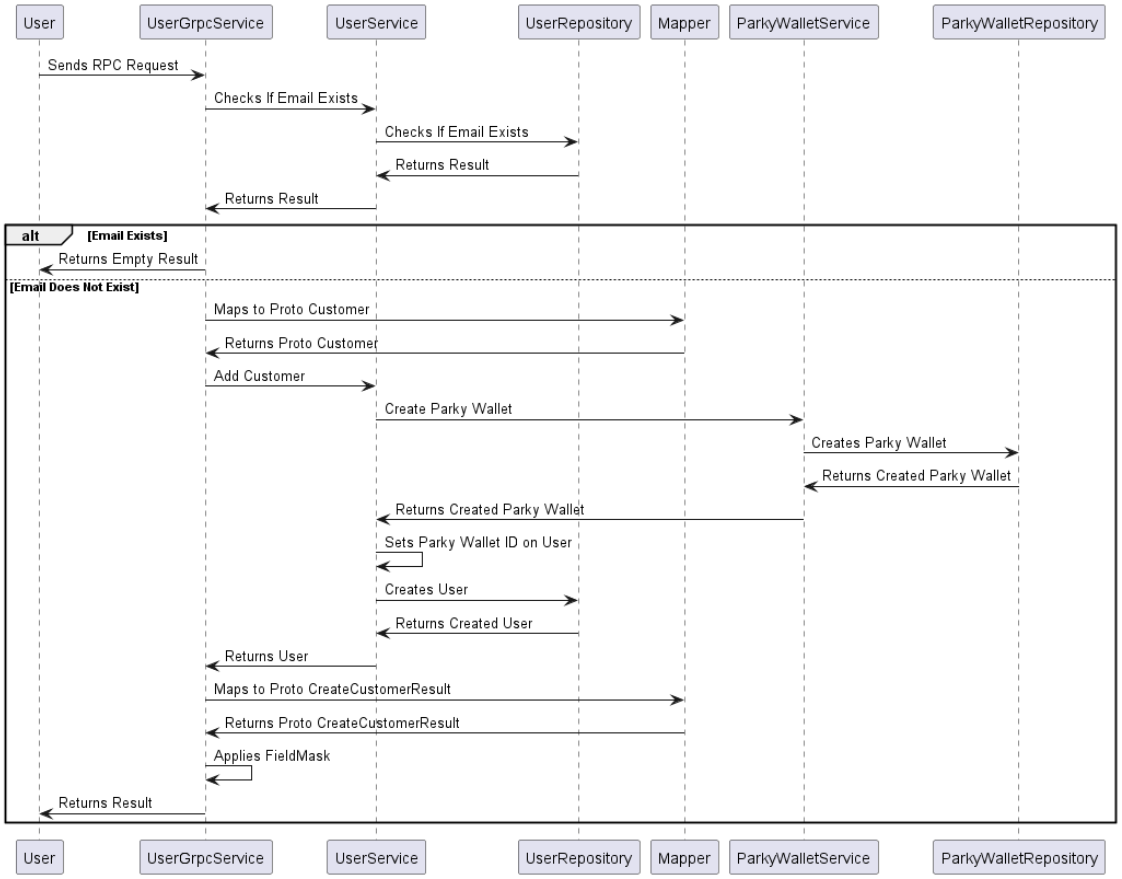

Figure 22 - Create User Unary

```
export default () => {
  client.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  let value = randomString(20);
  const req = {
    Name: value,
    Password: value,
    Email: value,
    Username: value,
    fieldMask: "Name,Email,Username",
  };

  const response = client.invoke("Proto.UserGrpc/AddCustomer", req);

  check(response, {
    "status is OK": (r) => r && r.status === grpc.StatusOK,
  });

  client.close();
};
```

Code 68 - Test function for gRPC CreateUser

Appendix F UpdateParkingValue Sequence Diagrams

The following diagrams and code scripts represent implementations of UpdateParkingValue for GraphQL, and gRPC and Protobuf.

The Figure 23, Code 69 and Code 70 show the GraphQL application.

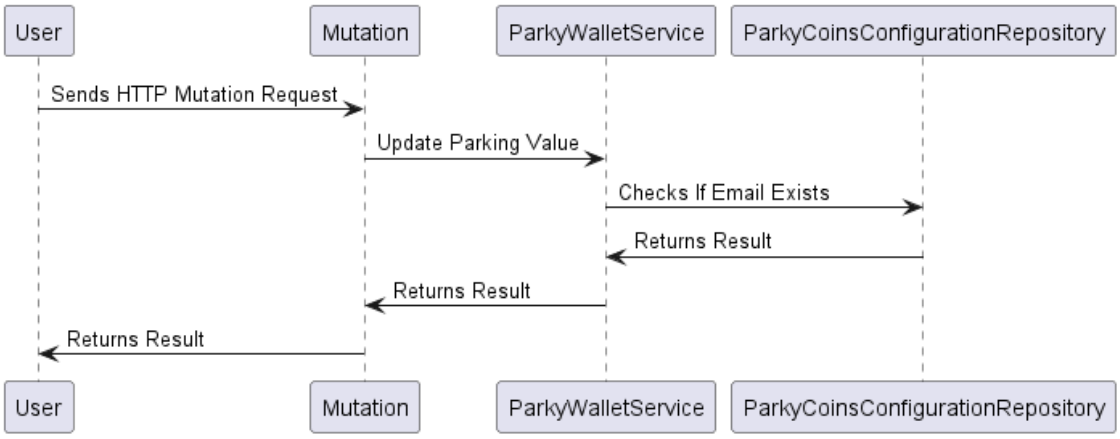

Figure 23 - Update Parking Value GraphQL

```
const query = `
mutation ($input: Int!) {
  updateParkingValue(value: $input)
}
`;
```

Code 69 - Mutation for GraphQL UpdateParkingValue

```
export default function () {
  let value = randomIntBetween(0, 99999);
  http.post(
    "http://localhost:5000/graphql",
    JSON.stringify({
      query: query,
      variables: {
        input: value,
      },
    }),
    {
      headers: headers,
    }
  );
}
```

Code 70 - Test function for GraphQL UpdateParkingValue

The Figure 24 and Code 71 show the gRPC application.

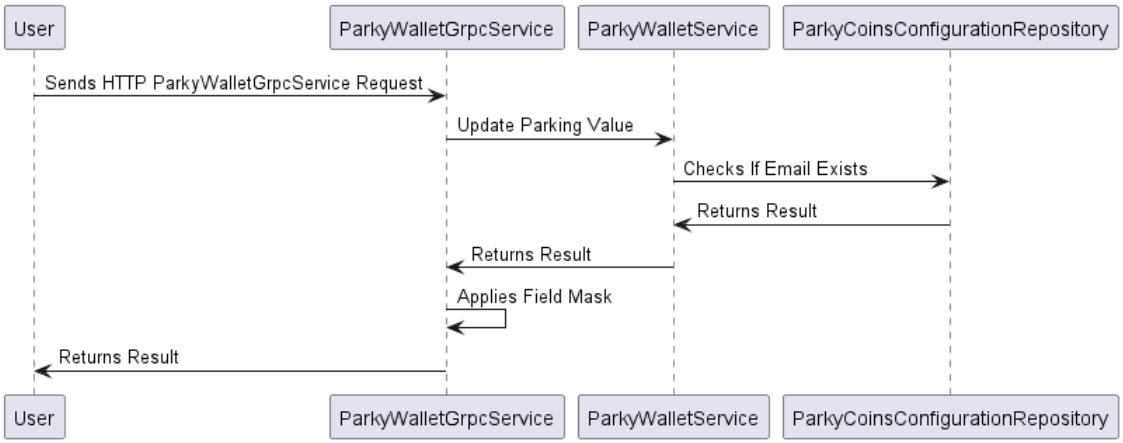

Figure 24 - Update Parking Value Unary

```
export default () => {
  client.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  let value = randomIntBetween(0, 99999);
  const req = {
    amount: value,
    fieldMask: 'isSuccessful',
  };

  const response = client.invoke("Proto.ParkyGrpc/UpdateParkingValue",
req);
  check(response, {
    "status is OK": (r) => r && r.status === grpc.StatusOK,
  });

  client.close();
};
```

Code 71 - Test function for gRPC UpdateParkingValue

Appendix G LeavePark Sequence Diagrams

The following diagrams and code scripts represent implementations of LeavePark for GraphQL, and gRPC and Protobuf.

The Figure 25, Code 72, Code 73, Code 74 and Code 75 show the GraphQL application.

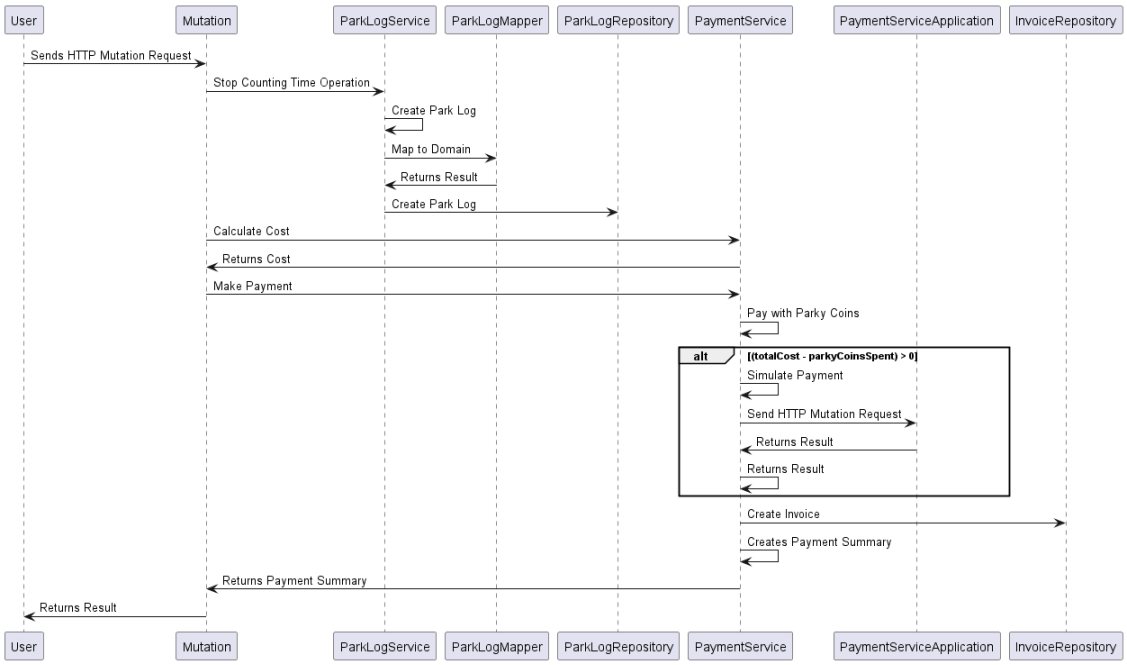

Figure 25 - Leave Park GraphQL

```
const addTokenMutation = `
mutation ($token: String!){
  addToken(token: $token)
}
`;
```

Code 72 - Mutation for GraphQL AddToken

```
export function setup() {
  http.post(
    "http://localhost:5004/graphql",
    JSON.stringify({
      query: addTokenMutation,
      variables: {
        token: "tok_123456789",
      },
    }),
    {
      headers: headers,
    }
  );
```

```
}
```

Code 73 - Setup function for GraphQL LeavePark

```
const leaveParkMutation = `
mutation ($input: ParkingSpotsUpdateRequestDtoInput!) {
  leavePark(requestDto: $input) {
    isSuccessfull
    parkyCoinsAmount
    otherPaymentMethodAmount
    totalCost
  }
}
`;
```

Code 74 - Mutation for GraphQL LeavePark

```
export default function () {
  http.post(
    "http://localhost:5000/graphql",
    JSON.stringify({
      query: leaveParkMutation,
      variables: {
        input: {
          isEntrance: false,
          licensePlate: "ABC123",
          parkName: "Central Park",
        },
      },
    }),
    {
      headers: headers,
    }
  );
}
```

Code 75 - Test function for GraphQL LeavePark

The Figure 26, Code 76 and Code 77 show the gRPC application.

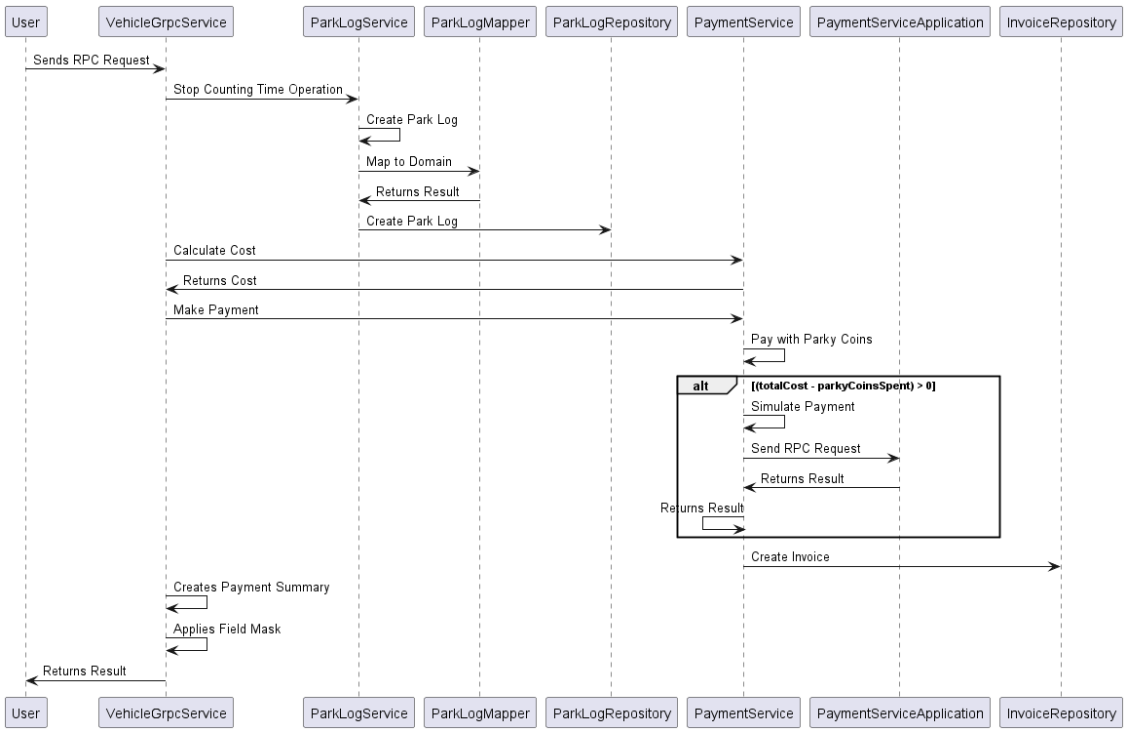

Figure 26 - Leave Park Unary

```
export function setup() {
  paymentClient.connect("localhost:7004", {});

  // Create the payload with the fields and field mask
  const req = {
    token: "tok_123456789",
    fieldMask: "result",
  };

  const response = paymentClient.invoke("Protos.PaymentGrpc/AddToken",
req);

  check(response, {
    "status is OK": (r) => r && r.status === grpc.StatusOK,
  });

  paymentClient.close();
}
```

Code 76 - Setup function for gRPC LeavePark

```
export default () => {
  backofficeClient.connect("localhost:7000", {});

  // Create the payload with the fields and field mask
  const req = {
    isEntrance: false,
    licensePlate: "ABC123",
    parkName: "Central Park",
```

```
    fieldMask:
"isSuccessfull,parkyCoinsAmount,otherPaymentMethodAmount,totalCost",
  };

  const                              response                              =
backofficeClient.invoke("Proto.VehicleGrpc/LeavePark", req);

  check(response, {
    "status is OK": (r) => r && r.status === grpc.StatusOK,
  });

  paymentClient.close();
};
```

Code 77 - Test function for gRPC LeavePark

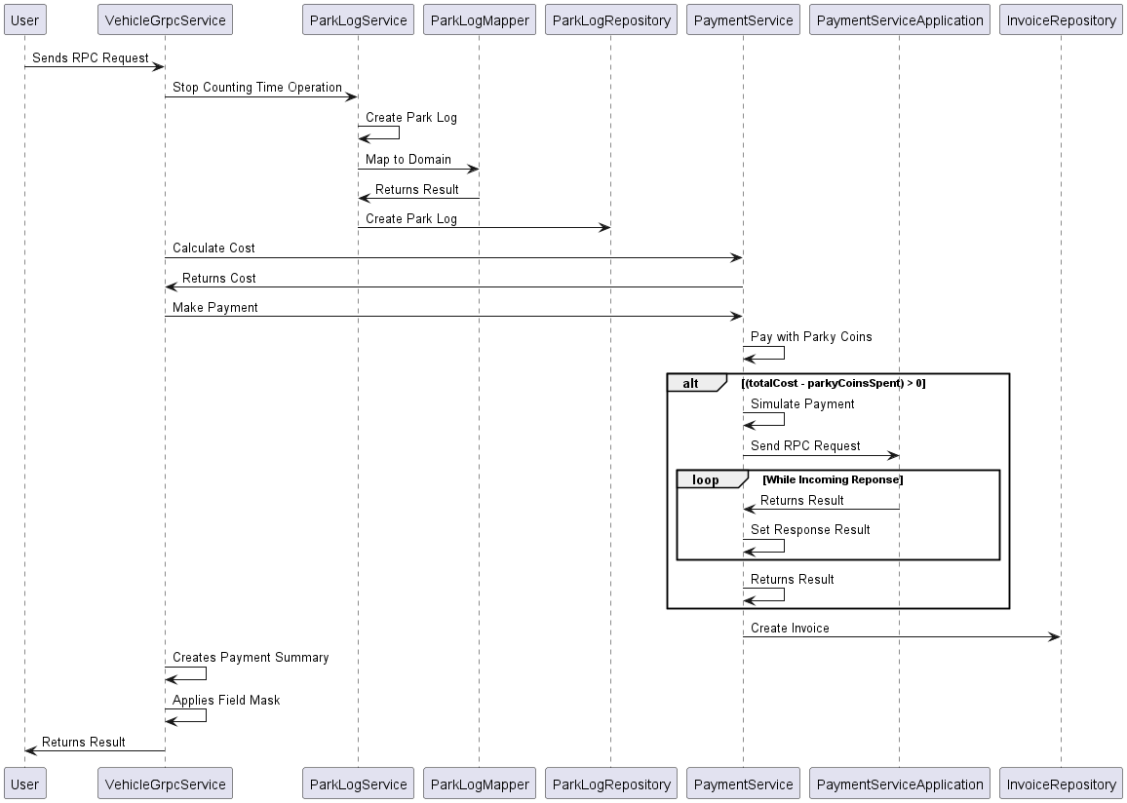

Figure 27 - Leave Park Server Stream

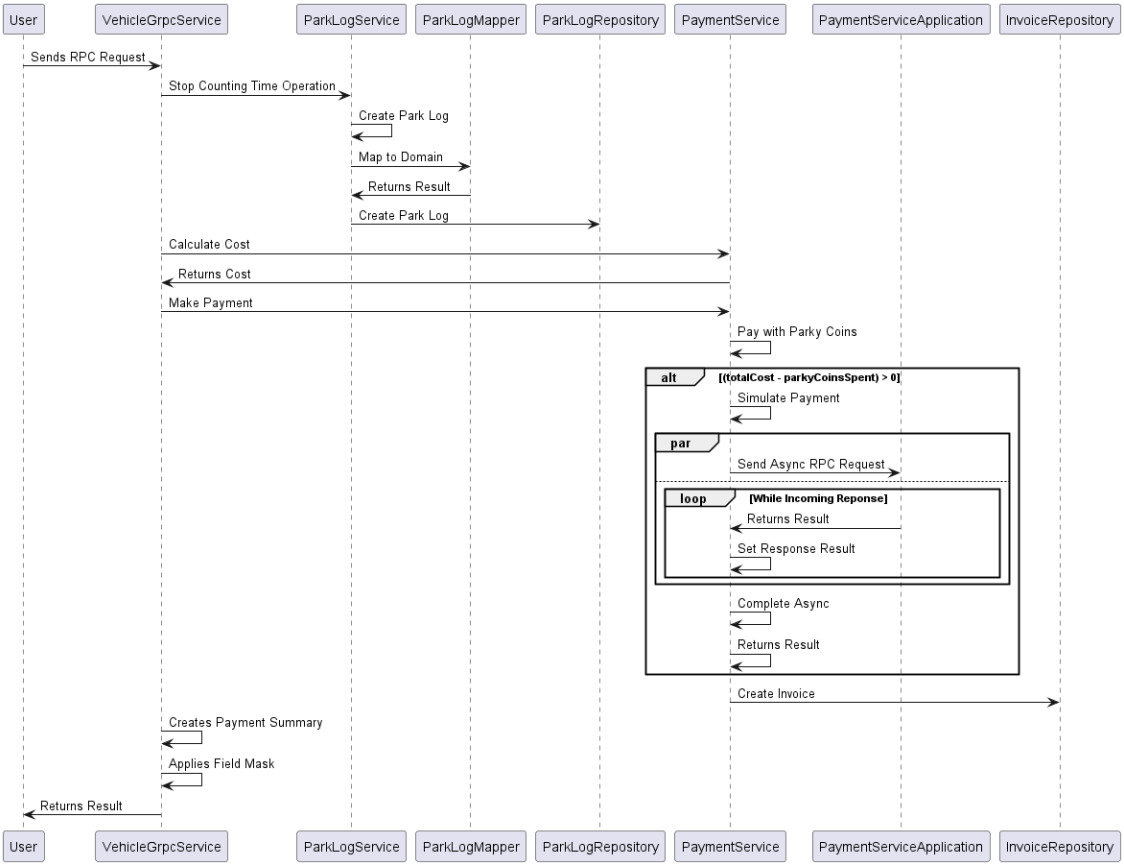

Figure 28 - Leave Park Two Sided Stream

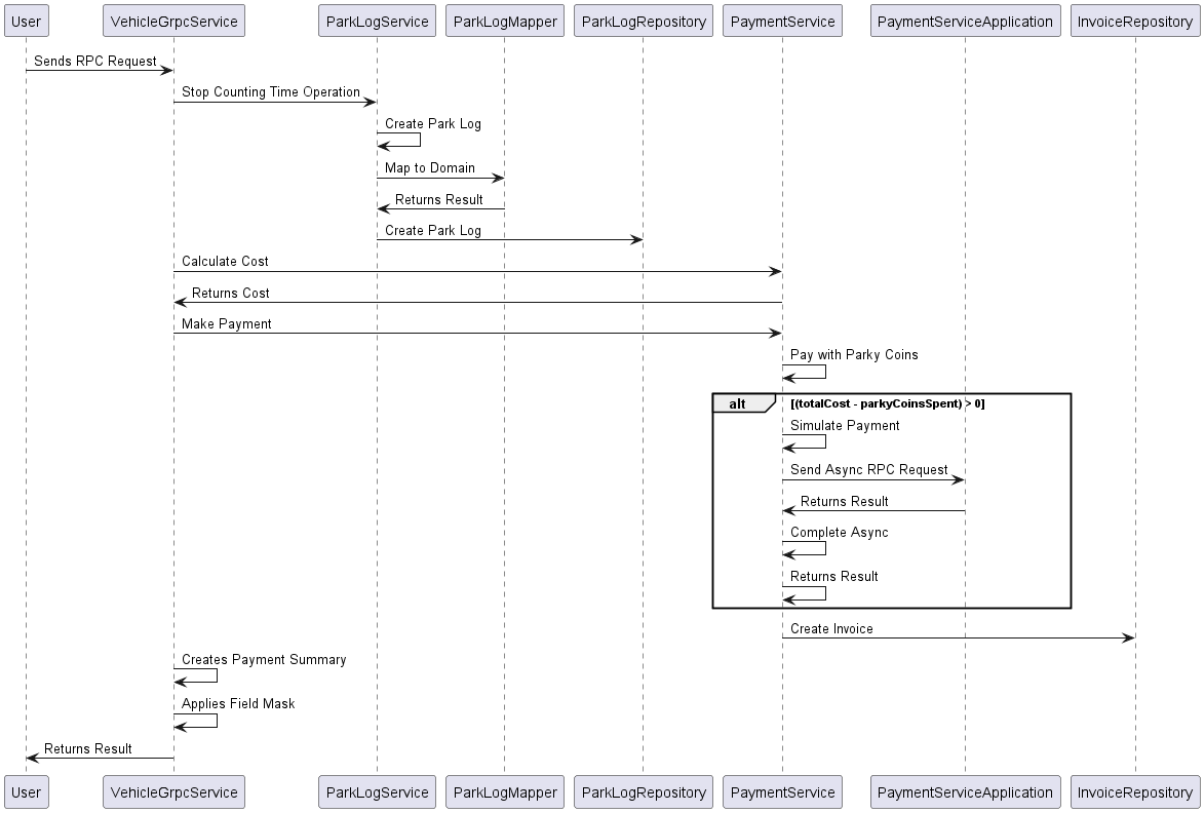

Figure 29 - Leave Park Client Stream

Appendix H Hypothesis Results Table

This table compiles all calculated values for the hypothesis testing to determine if without a doubt gRPC was quicker than GraphQL.

Table 25 - Hypothesis Results Table

Variable Name	Value
grpcAllParks_graphqlAllParks	0
grpcAllParksClientStream_graphqlAllParks	0
grpcAllParksServerStream_graphqlAllParks	0,21
grpcAllParksTwoSidedStream_graphqlAllParks	0,35
grpcCreateUser_graphqlCreateUser	$2{,}67x10^{-51}$
grpcLeavePark_graphqlLeavePark	$6{,}51x10^{-246}$
grpcLeaveParkClientStream_graphqlLeavePark	$3{,}87x10^{-219}$
grpcLeaveParkClientStreamPayment_graphqlLeaveParkPayment	0
grpcLeaveParkPayment_graphqlLeaveParkPayment	0
grpcLeaveParkServerStream_graphqlLeavePark	$1{,}10x10^{-263}$
grpcLeaveParkServerStreamPayment_graphqlLeaveParkPayment	0
grpcLeaveParkTwoSideStream_graphqlLeavePark	$1{,}64x10^{-263}$
grpcLeaveParkTwoSideStreamPayment_graphqlLeaveParkPayment	0
grpcPartialParks_graphqlPartialParks	0
grpcUpdateParkingValue_graphqlUpdateParkingValue	0