MASTER'S THESIS 2022

The Evaluation of Using Backend-For-Frontend in a Microservices Environment

Samer Alkhodary

ISSN 1650-2884
LU-CS-EX: 2022-25

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-25

The Evaluation of Using Backend-For-Frontend in a Microservices Environment

Samer Alkhodary

The Evaluation of Using Backend-For-Frontend in a Microservices Environment

Samer Alkhodary
sa2808al-s@student.lu.se

June 8, 2022

Master's thesis work carried out at Qlik Foreign Parent AB.

Supervisors: Alfred Åkesson, alfred.akesson@cs.lth.se
Paul Ericsson, paul.ericsson@qlik.com

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

In modern web applications, the client keeps track of several services in a backend consisting of several microservices, known as the microservices pattern. The client makes several requests to the services to gather the needed information, which adds extra latency. The backend for frontend pattern (BFF) is a possible solution that mitigates the microservice pattern's performance overhead. The BFF pattern is where a BFF service acts as a gateway between the client and the backend services. Therefore, all communication between the services and the client goes through the BFF. This thesis investigated the BFF pattern's effects on latency, data usage, the dependencies between the client and the backend, and the client's code. We implemented the BFF pattern in Qlik's staging environment using three different technologies,i.e., gRPC, REST, and GraphQL. As a result, we found that the BFF positively impacted all the criteria mentioned above.

Acknowledgements

I would like to express our sincere gratitude to our supervisors, Alfred Åkesson, Paul Ericsson, Johan Enell, and Hermann Ronaldsson, for their continuous support and constructive feedback and provided us with everything we need to finish this thesis.

Contents

Chapter 1
Introduction

Since the boom of smartphones, the internet has played a significantly larger role in our lives, and gradually everything has become just a click away. Nowadays, we can keep track of the latest news using our favorite news platform, stay connected with other people, and shopping is easier than ever. We can buy whatever we want using only our phones. We can do all that by using different applications that we can easily download on our phones, tablets, and computers.

The majority of these applications share a similar design. They have one or more client types(mobile, browser, or desktop) and a backend system that these clients can communicate with. Many internet giants,e.g., Google, Netflix, and eBay's backend, consisted of large services that handled all the requests from the clients,i.e., the Monolithic Architecture. However, in the 2010s, they shifted to an architecture where the backend system gets divided into several smaller services that interact with each other to fulfill the purpose of the entire system. Every service handles parts of the incoming requests to the backend application, which enables the modularization of the backend system, and this architecture is called the Microservices Architecture[28].

The microservice architecture has several benefits over the monolithic architecture. First, it improves the scaling capabilities of the system because only services that handle many requests can be scaled compared to the monolithic architecture where the entire system gets scaled, requiring more resources. Moreover, with microservices, each service can use a different technology stack depending on the performance requirements [2]. Additionally, the microservices architecture increases the resilience of the system because when a service crashes, it does not crash the entire system compared to the monolithic architecture, where any crash in the program can stop the entire system [2].

The microservice architecture does not come without some drawbacks. One of the drawbacks is that the clients have to keep track of many services and the different entry points

the services offer, also known as *Endpoints*. Also, the microservices architecture forces the clients to make several requests to different endpoints and services to fetch the information they need. Compared to the monolithic architecture where everything could be fetched with one request. However, some solutions have emerged to mitigate this issue, e.g., the API gateway pattern and the Backend-for-Frontend design pattern. The API gateway is a service that sits between the clients and the backend services and acts as a single entry point to the system. First, clients send requests to the gateway, and then the API gateway contacts the relevant endpoints to gather the clients' information. Eventually, it sends back responses to the clients[1]. The Backend-For-Frontend pattern, also known as BFF, is similar to the API gateway pattern. However, every client type, e.g., mobile, desktop, tablet, browser, gets its own BFF. Consequently, the frontend team can tailor the BFFs to fit the needs of their client types. For example, they can manipulate responses from the services and only send back the data the client needs. Moreover, some logic can be moved from the client to the BFF. Consequently, the code complexity of the client codebase is reduced, which reduces the number of bugs in the client codebase. This can be particularly useful in mobile clients where publishing updates to fix bugs can be a time-consuming process because the App Store or the Play Store must approve the updates before they get released to the mobile users[12][4].

In this thesis, we will investigate the impact of the BFF design pattern on the communication between a mobile client and a backend system that consists of several microservices. Moreover, we will implement three different proof of concept BFFs using three different technologies, i.e., gRPC, GraphQL, and REST. Finally, we will investigate the effects of using those different technologies on the implementation of BFFs and conduct a threat analysis on the design pattern. Consequently, this study will help software providers make an informed decision regarding the BFF design pattern and which communication technique to use within the BFF.

We conduct this thesis at Qlik[31]. Qlik has an application called Qlik Sense SaaS [13][38] which helps users monitor their data using visualizations, charts, and data alerts that trigger when certain conditions are met. We will use the application and its backend system as a test bed for our experiments.

1.1 Objectives

- Investigate the benefits of using the BFF design pattern to optimize the communication between the mobile client and the backend system.

- Evaluate the implemented techniques with respect to maintainability and performance.

- Evaluate the BFF design pattern from a security perspective and provide recommendations that make the entire design pattern more secure.

1.2 Overview

First, in chapter two, we familiarize the reader with the core concepts and the background information needed to follow this study. After that, in chapter three, we talk about Qlik's current setup with their backend and mobile client and the challenges they face. Also, we suggest the BFF as a possible solution, and we mention the methods we want to use to investigate the impact of the BFF design pattern on the entire system. Then, in chapter four, we specify the technologies we chose to implement for the different BFFs. Finally, we also give an example that explains how those technologies function. Chapter five describes the experiments that we are conducting and lists all the metrics that we collect to compare the different implementations of the BFFs and the impact of the BFF on the entire system. We also state our results and findings in this chapter, evaluate them based on the hypothesis we have and mention any factors that could have affected the accuracy of the results. Also, in chapter five, we conduct the BFF's threat analysis to shed light on the cyber security aspects of the BFF. Then, in chapter six, we mention some related studies and research papers about the BFF design pattern, and we compare our results to their findings. Eventually, in chapter seven, we discuss the conclusion.

Chapter 2
Background

This chapter contains the necessary background information to understand the report.

2.1 Services and Endpoints

First we explain the concept of web services and endpoints, because it is important that the readers familiarize themselves with these concepts to understand the two different service architectures that we explain next. Also, we use those concepts a lot in the paper.

2.1.1 Web Service

Web services are internet applications that can communicate with other applications using well-defined communication protocols[9]. In this thesis, we use the "services" term to refer to programs that are part of the backend system and are responsible of sending data to other programs when they are requested.

2.1.2 Endpoints

Endpoints are entry points for other web applications that allow those applications to communicate with the web service[30]. For example a user service can have these endpoints:

- GET users/v1/users
- GET users/v1/users/{id}

Sending a GET request to "user/v1/users" endpoint, would return a list of users

2.2 Service Architectures

There are several ways to design services in the backend, and here we discuss two different approaches, i.e., the monolithic and the microservices architectures. These concepts are essential for the reader to understand how the BFF pattern functions. Also, Qlik uses microservices architecture. Therefore, we conduct the entire experiments in this paper in a microservices environment.

2.2.1 Monolithic Architecture

The monolithic architecture consists of one relatively large service that handles the logic of the entire web application and is responsible for responding to all the requests that reach the application. Moreover, all the modules inside that service are dependent on each other and must be present to compile the application. For example, a monolithic webshop service is one program that responds to requests about users, products, and prices(see figure 2.1).

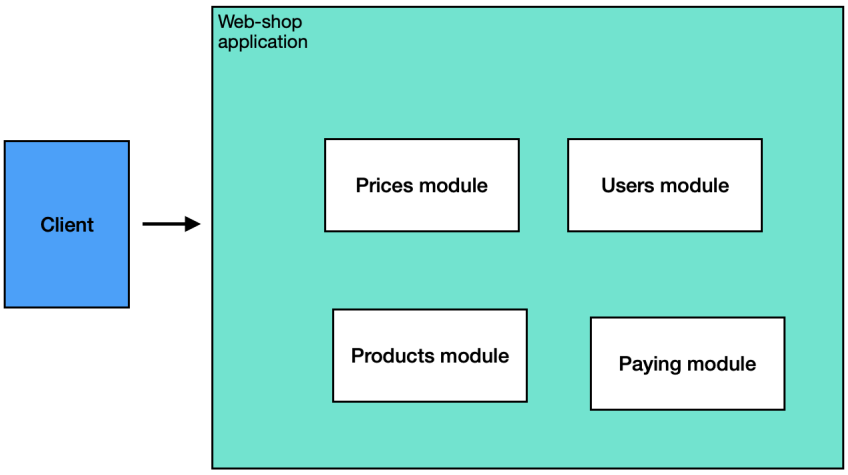

Figure 2.1: Monolithic web-shop example

2.2.2 Microservices Architecture

The microservices architecture is an architectural style where a web application is split up into several small services that can be compiled and shipped independently. Every service is specialized to handle part of the applications' logic, and all these services work together to fulfill the purpose of the web application [33]. For example, applying the microservices architecture with the previous webshop example, the webshop backend would consist of a paying service, a users service, a products service, and a prices service (see figure 2.2).

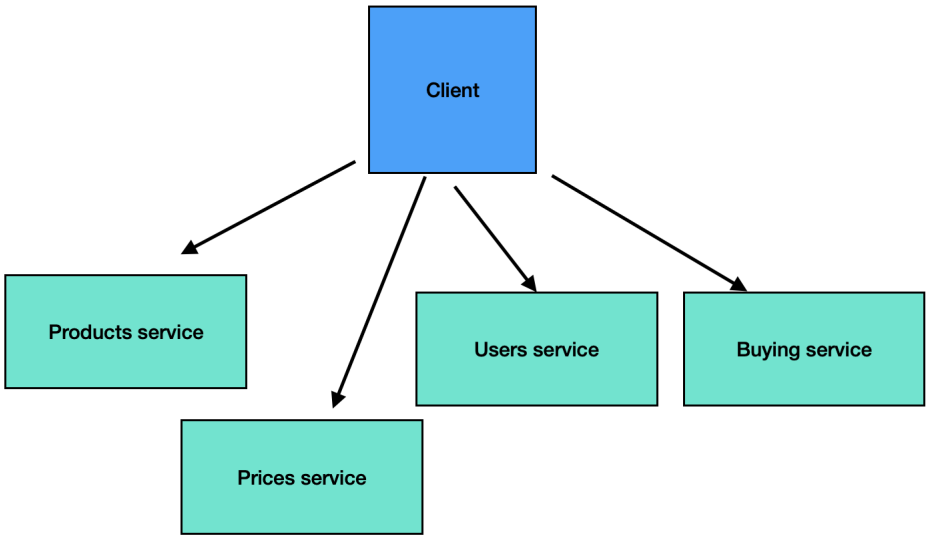

Figure 2.2: Microservices web-shop example

2.3 HTTP Protocols

Qlik uses HTTP as a communication protocol between its backend system and the different frontend clients. Also, our BFF implementations will use both HTTP/1.1 and HTTP/2. The Hypertext Transfer Protocol (HTTP) is an application-level stateless protocol[19]. The HTTP protocol allows Clients to request server resources by sending HTTP requests. The servers then send the requested resources by sending HTTP responses to the client.

2.3.1 HTTP/1.1

HTTP1.0 and HTTP1.1 create a new TCP connection for every HTTP request, which increases the latency due to the extra internet packages that are sent when creating and closing down TCP connections. However, HTTP 1.1 introduced the concept of **Persistence connection** which allows multiple HTTP requests to use the same TCP connection. However, each connection can only handle one request at a time. To solve this issue, HTTP 1.1 allows users to enable **Pipelining** which allows multiple inflight requests over the same connection, but the requests need to be served in order. As a consequence, requests that require a long time to process can block all the requests that come after it, and this problem is called **Head Of The Line Problem**[36].

2.3.2 HTTP/2

HTTP2 is an optimization of the HTTP1.1 protocol. It allows multiplexing multiple requests over a single TCP connection,i.e., each request and response gets sent in a unique HTTP2

stream. This feature improves the HTTP1.1 Pipelining because the requests do not need to be processed in order, which mitigates the Head of the line problem in HTTP1.1 without opening several TCP connections. Also, HTTP2 introduces several new features, such as allowing servers to push data to the client in advance and compressed HTTP1 headers[7].

2.4 Staging and Production Environments

The reader needs to be familiar with the differences between the staging and the production environments because we host our BFFs in Qlik's staging environments.

- Production environment is the current released version of the services that the clients and users can contact.
- Staging environment: is an environment that contains the upcoming versions of the services in production, and it is normally a mirror of the production environment and is used for testing the services before they reach production.

2.5 Containerization and Clustering

When deploying our BFFs to the staging environment, we wrap them in Docker containers and then add them to the staging cluster using Kubernetes.

2.5.1 Containerization

When deploying a web application, developers wrap the web application and all the application's dependencies in an isolated environment. As a consequence, the programs become decoupled from the rest of the developing environment, which increases the portability of that software, thus, allowing them to run smoothly in any environment[22].

2.5.2 Docker

Docker is a technology that enables developers to encapsulate programs within containers s[21]. In order to run a container in Docker, first, we need to create a Dockerfile that contains instructions for Docker to build a Docker image. After that, Docker uses the instructions to create a read-only Docker image that contains the program's binaries and all the dependencies and configurations the program needs to execute. We run the image. Then Docker creates a container which is a writable layer on top of the read-only image. Any changes made to a running container are registered on the container layer, which means that the image does not change.

2.5.3 Kubernetes

Kubernetes is an open-source project that Google created for managing containerized services. It can deploy a certain number of containerized services and their replicas to the

specified computers. Moreover, it offers features that help monitor running services, self-healing of crashed services by re-running them automatically, and automated roll-outs of the pods(containers). Also, it handles load balancing by distributing the network traffic among the running services [24].

2.6 Gateways and BFFs

API gateways and BFFs act as facades that hide the complexity of the backend system from the clients, and they orchestrate the microservices so they can fetch any information the client requests.

2.6.1 API Gateway

An API gateway is a service that sits between the backend services and the different clients, and it acts as a single entry point to the backend system. The client first sends requests to the API gateway, and then, based on the request, the API gateway contacts the appropriate service to send a response back to the client[1]. It can decouple the application from the many services it depends on because the client only communicates with the Gateway. Also, when rendering a page that requires different data fetched from different endpoints and services, the client only needs to send one request to the Gateway. After that, the Gateway sends the necessary requests to the endpoints to build a response that has the necessary information for the client (see:2.3).

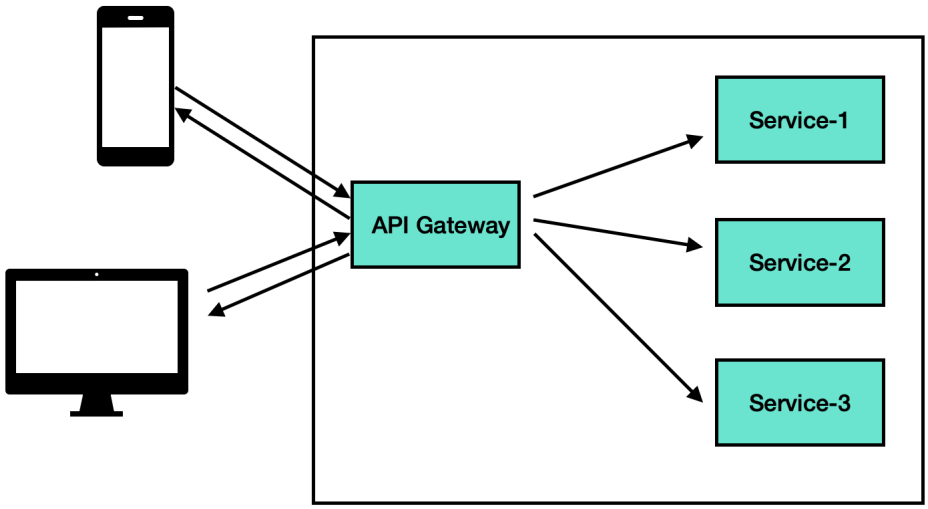

Figure 2.3: BFF Pattern

2.6.2 BFF Pattern

Backend for frontend is a design pattern where there is a service called BFF for each client type, and each of these services acts as an API gateway for its client type. Therefore, the BFF service can manipulate the responses from the microservices to fit the exact needs of the client type the BFF communicates with (see:2.4).

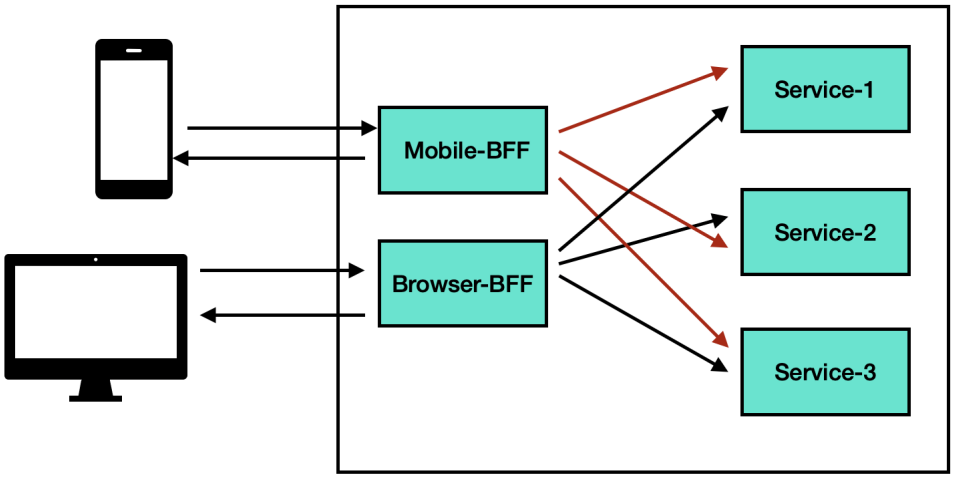

Figure 2.4: BFF Pattern

Chapter 3
Evaluation Scenario

This chapter will discuss Qlik's current solution and some of its drawbacks. After that, we will suggest using the BFF design pattern as a possible solution to mitigate those drawbacks. Lastly, we will describe a method to investigate the effects of the BFF on the entire system.

3.1 Qlik Setup

The Qlik Sense mobile application is connected to a backend system that consists of several containerized microservices that run in Kubernetes clusters. The mobile application has to keep track of several services and endpoints (see figure:3.1), and as a consequence, the app is tightly coupled to the backend services. Moreover, the application sends several HTTP requests to different endpoints to gather data to render complex information. For example, let us say that the mobile client wants to render a page called the Alert History page, which shows a list of specific alert evaluations. The application makes the following requests:

1. The client sends a request to the data-alert service to fetch the executions of the alert. Let us assume that the alert has N executions.
2. The client sends N requests to the data-alert service to fetch the results of every execution, also known as ***evaluations***.
3. The client sends a request for each evaluation to the notification service to fetch the rendering information for the evaluation.

Every API call adds extra latency to the total time needed to render the page, and in most cases, the mobile client wastes internet bandwidth to fetch more data than it needs (over-fetch). This can be a problem, especially for clients that have slower internet connections or live in places where mobile internet is relatively expensive.

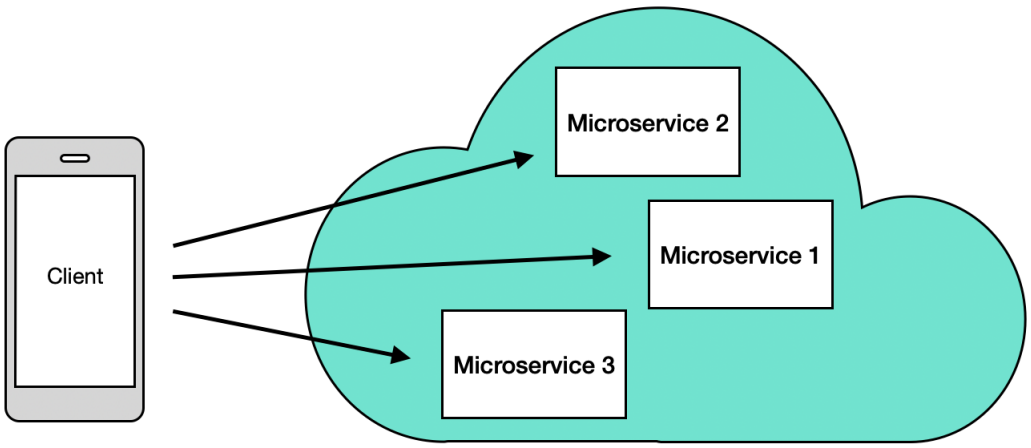

Figure 3.1: Qlik Setup

3.2 Solution

To solve these issues mentioned in the previous section, we will implement three different BFFs to sit between the microservices and the mobile application (see figure:3.2). Each BFF will use different technology, and we will use these different BFFs to compare the effect of using those different technologies when implementing a BFF. Also, we will compare those three BFFs to the current setup that does not utilize a BFF. All communication between the clients and the microservices will go through the BFFs. In the previous example, where the app wanted to fetch the data needed to render the Alert History page, The protocol becomes the following:

1. The client requests the BFF
2. The BFF sends a request to the data-alert service to fetch the executions.
3. The BFF sends N requests to the data-alert service to fetch the evaluations.
4. The BFF sends N request to the notification service to fetch the rendering information
5. The BFF sends the data to the client.

Since the BFF lives in the same cluster as the other microservices, the latency of the API calls between the microservices and the BFF should be lower than the latency between the mobile application and the microservices. Therefore, the BFF design pattern should reduce the latency of fetching the data the mobile client needs.

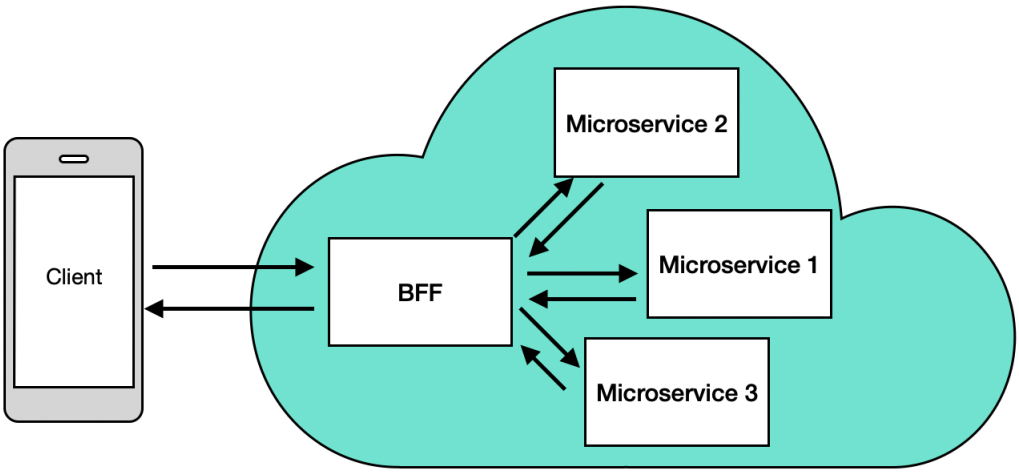

Figure 3.2: Qlik-BFF

Chapter 4
Technologies and BFFs

This chapter lists the different BFFs that we implement and the technologies that we chose to use in our BFFs. The examples we mention below are simplified examples of fetching the history page for some alert. Many fields are removed from the examples to make them easier to understand.
It is important to note that we wrote all the BFFs in Typescript, and we used Docker to containerize the services.

4.1 REST

This version of the BFF uses REST to communicate with the microservices, and it uses REST to communicate with the clients. REST stands for Representational State Transfer, and it is a set of rules that define how APIs must function. Any API that follows those constraints is considered to be a RESTful API[25]. Some of the REST constraints are that the communication between the server and the client must be stateless. Moreover, the client and the server must use HTTP methods to communicate with each other in a way that follows the protocol definition in RFC 2616 [34]. We chose REST because it is one of the most popular ways to design APIs. Also, the current solution in Qlik Sense Application uses REST to communicate with the backend services. The following is an endpoint in our BFF to fetch the information for the alert history page using REST:
The BFF exposes the following endpoint:

`/v1/bff/history`

The client sends a GET request to this endpoint

```
const response = fetch('https://example.com/history?id="123"')
```

The the BFF sends back the following response to the client:

```
{
   // Response from the service containing the history
   // of the alert that we asked for.
    "history": {
        "executions":[
        {
            "alertId": "123",
            "evaluation": {
                "conditionId": "some id"
            }
        },
        ]
    }
}
```

4.2 GraphQL

The GraphQL version of the BFF communicates with the microservices using HTTP1.1, and it communicates with the mobile application using GraphQL. We used the Apollo server to implement the BFF [3]. GraphQL is a query language for APIs[15], and it is governed by the GraphQL Foundation that is made up of different companies, e.g., Airbnb, AWS, Meta, and IBM. However, Facebook developed its first specifications[14]. Therefore, we Chose GraphQL because it provides a simple method to prevent over-fetching[15] as we will see in the example below. A GraphQL service first creates a schema that defines all the service's data. Then, the service defines queries that the clients can use to fetch the data defined in the schema, which means that GraphQL services execute client queries using the predefined types in the schema. This is an example of a GraphQL service: First, we define our types and query in the schema:

```
//Here first we define our custom types.
//We use the custom types to define our schema.
type Evaluation {
    endTime: String
    conditionId: String
    ownerId: String
    status: String
  }
type Execution {
    alertId: String
    evaluation: Evaluation
    text: String
}
type HistoryResponse {
    history: History
}
type History {
    executions: [Execution]
}
  //Here we define the query that the client can use
  //It requires a alertId as an argument
  //It returns the HistoryResponse object
  extend type Query {
    getHistory(
      alertId: String!
    ): HistoryResponse!
  }
```

The client send a query to the GraphQL service,

```
//This a query that has two parameters(alertId and the locale.
getHistory(alertId: "123"){
//We define inside the query the fields we want the
//response to contain.
    history{
        executions{
            alertId
            evaluation{
                condtionId
            }
        }
    }

}
```

After that, the service analyzes the queries, and then it sends back the requested fields in the GraphQL queries to the client:

```
{ //We can see that the response contains only the
//fields the client requested in the query.
    "getHistory": {
        "history": {
            "executions":[
            {
                "alertId": "123",
                "evaluation": {
                    "conditionId": "some id"
                }
            },
           ]
        }
    }
}
```

4.3 gRPC

The gRPC version of the BFF also uses REST to communicate with the backend services, and it uses gRPC to communicate with the frontend clients. gRPC (General Remote Procedure Call) is an open-source Remote Procedure Call framework created by Google, where clients can call methods in the server application on a different machine[16]. gRPC utilizes HTTP/2, and therefore, it can process multiple remote procedure calls in a single TCP connection making use of the multiplexed stream functionality in HTTP/2[11]. In gRPC, programmers first define services and then specify which methods the client applications can call with their parameters and return types using **protocol buffer (a.k.a Protobuf)**(Google's method for binary serializing structured data). Also, gRPC clients and services use binary when communicating with each other, while REST and GraphQL use text-based transfer of information.

```
// Here we define our custom types that we
// use in our communication.
message Evaluation{
    string conditionId = 2;
    string endTime = 1;
}
message Execution{
    string alertId = 1;
    Evaluation evaluation = 2;
    string text = 3;
}
// Repeated means a sequence (Array)
message History{
    repeated Execution executions = 4;
}
message HistoryRequest{
    string alertId = 1;
    string locale = 2;
}
service Users{
    // Here we define the different procedures
    // we want the use in our communication.
    // GetHistory is a procedure that takes a HistoryRequest object
    // as an argument and returns a History object as a response
    rpc GetHistory(HistoryRequest) returns Histroy{};
}
```

After specifying the data structure and the services, the protocol buffer compiler generates the server interface and the client stub. The server implements the interface and then runs a gRPC server to handle the client calls. On the client-side, the client uses the client stub to call the specified methods on the server[16].

```
// Here we create a HistoryRequest object
// Then we call the getHistory procedure that the
```

```
// client implents with the request as an argument.
const request =  new HistoryRequest().
    setAlertId("123").
    setLocale("US");
const history = client.getHistory(request);
```

The server responds with a history object that contains the same fields that are listed in the message History above.

Chapter 5
Evaluation

5.1 Backend System Setup

The backend consists of several containerized microservices hosted in a Kubernetes cluster hosted in Qlik's stage environment. Each service exposes several endpoints the clients use to communicate with it. In our experiments, we will be only interacting with the dataalerts service, the users service and the notification service which is an internal service, so it is not available for the general public. The data-alerts service exposes several endpoints that we can use to fetch data with varying degrees of complexity. Moreover, the data-alerts service sends back lots of data that the application does not need when rendering the history of an alert. On the other hand, the user service returns relatively smaller responses that limit over-fetching to minimal levels. With the help of these two services, we can study the effect the BFF has on the amount of redundant data the application receives from the backend system. Additionally, we can analyze the BFF's impact on the latency when the application asks for data that require a different number of API calls.

5.2 BFF Experimental Setup

We wrap each BFF into a Docker container, and we add it to the Kubernetes cluster in the staging environment. We can't have all three BFFs available simultaneously in the cluster. Because of that, we swap them every time we finish conducting the experiments for each BFF.

We used the metrics in the experiments to study the performance, code implementation, and security aspects of the different technologies. We hosted the three BFFs in the same cluster as the other microservices. The BFFs are connected to three services, i.e., the users' service and the data-alerts service, and the notification service, and they can send requests to the following endpoints:

- /data-alerts/{taskId}/executions
- /data-alerts/{taskId}/executions/{executionId}/evaluations
- /users/{userId}
- /notification/renderInformation

5.3 Frontend System Setup

To test the effect introduced by the BFF, we emulated the mobile client by creating four scripts. The first script directly contacts the microservices' endpoints without going through the BFF. This script aims to collect data about the system's performance without having a BFF in place. For the rest of the scripts, strictly contact the BFF when making backend system requests. We modified each script to use one of the technologies we used to implement the three BFFs,i.e., (GraphQL, REST, and gRPC). Each script contacts the BFF that uses the same communication method as the BFF,e.g., the GraphQL script communicates only with the GraphQL BFF. The purpose of the scripts is to simulate the requests that the mobile client makes when fetching the data it needs.

5.4 Performance Evaluation

In this section, we study the BFF's impact on communication performance,i.e., latency and data volume between the client and the backend system when fetching data with different complexities. We created three different experiments to measure the latency of getting the alert history for the different alerts in different network conditions,i.e., Normal(WiFi), 3G, and Edge. We use the Network Link Conditioner, which is software that can change the network environment according to the selected configurations [27]. The Network Link Conditioner allows us to alter the network's configurations for our scripts to simulate mobile application users with slower internet connections. Also, we create an experiment to measure the latency of fetching the user information using the BFFs and the current solution in all network conditions. To measure the data volume, we create an experiment that measures the amount of data the client sends receives when fetching different alerts' histories. Finally, we create one more experiment to measure the data that the client sends and receives when fetching some user's information. The configurations for the different network conditions that we use are the following:

Network	U Bandwidth	U Delay	D Bandwidth	D Delay
Normal	350.4 Mbps	1 ms	575.5 Mbps	1 ms
3G	330 kbps	100 ms	780 kbps	100 ms
Edge	200 kbps	440 ms	240 kbps	400 ms

Table 5.1: Different network conditions where:
D = Download
U = Upload

5.4.1 Experimental Setup

The mobile application shows a page called alert history for every alert, and to render that page, first, the app sends a request to the data-alerts service to fetch the alerts' executions by conducting the following operations:

1. First, it sends a request to the data-alerts service to get the alert's executions. It sends the requests to the following endpoint:
   GET /data-alerts/{taskId}/executions

2. Then, for every execution, it sends a request to the data-alerts service to fetch the evaluation data of that execution by sending requests to the following endpoint:
   GET /data-alerts/{taskId}/executions/{executionId}/evaluations

3. Eventually, the application uses the information to render the alert's history page.

In this experiment, we emulate getting the data the application needs to render the alert history page. We chose this page because it is easy for us to manipulate the number of API calls the application or the BFF needs to collect the necessary data, i.e., for an alert with N evaluations, the application needs to make 2N+1 API calls to fetch the data.
The first script contacts the microservices directly, and it follows the same steps as the application when fetching the data to render the alert history page. see figure 5.1.

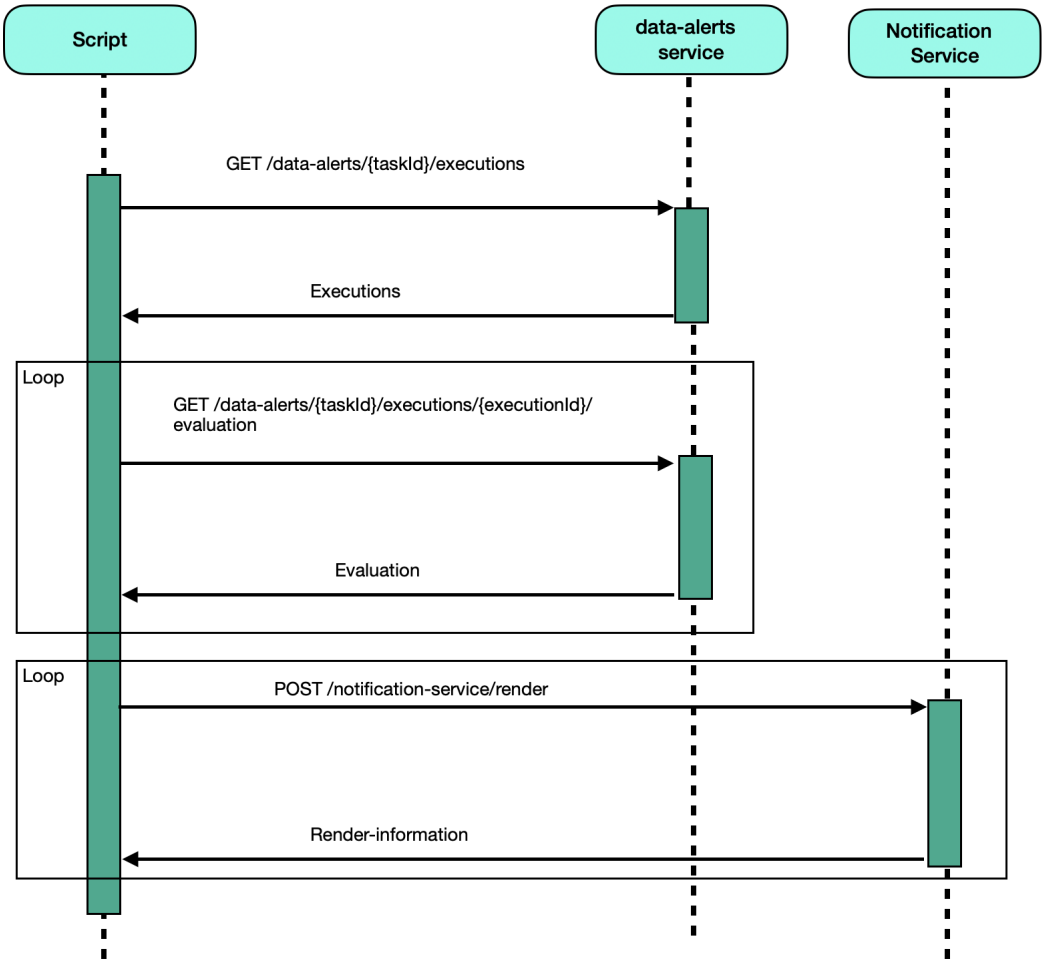

Figure 5.1: Current solution script

The other scripts first send a request to the BFF, and then the BFF sends requests to the previously mentioned endpoints, and eventually, it sends back a response to the clients. See figure 5.2.

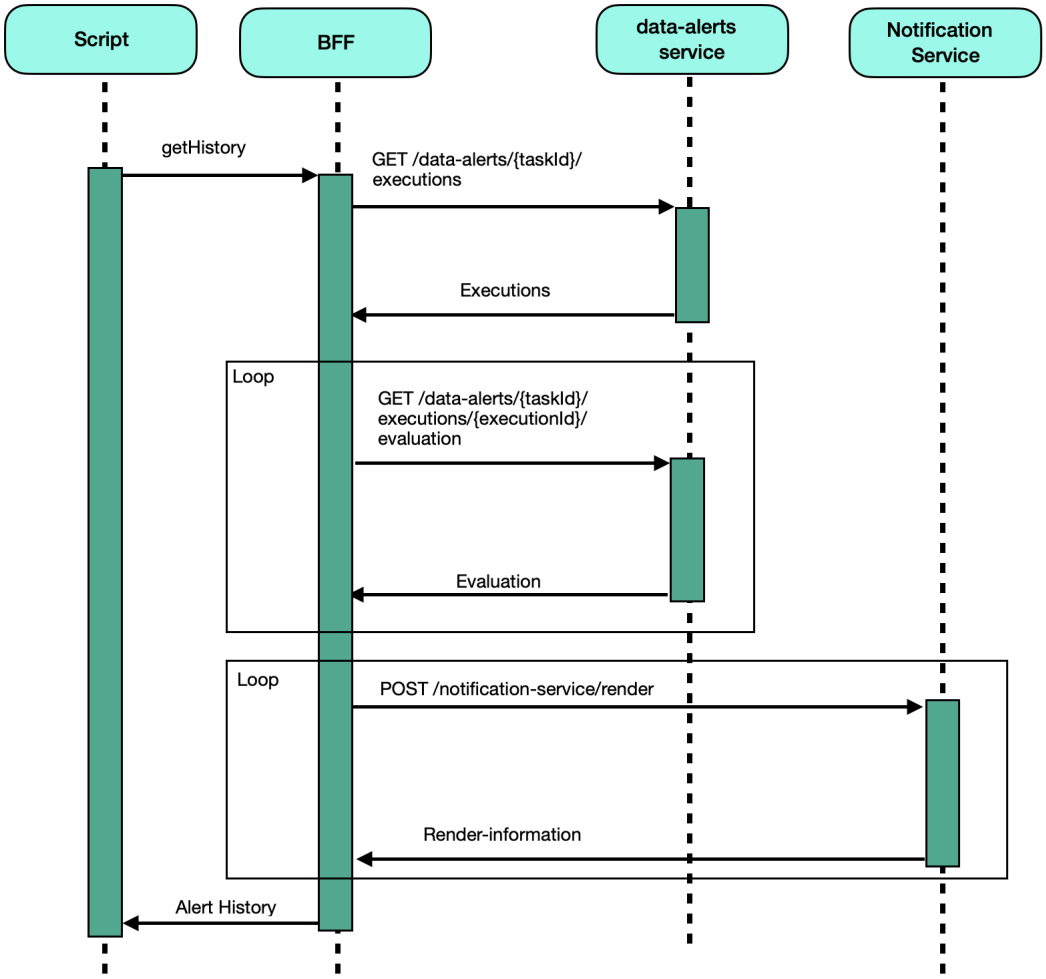

Figure 5.2: Scripts that use BFFs

First, we create five identical alerts in Qlik's stage environment to evaluate each alert a different number of times. The more times we evaluate an alert, the more API calls will be needed to get the alert's history, and the larger the history response will get.

Alert	# Evaluations (N)
Alert-1	1
Alert-2	4
Alert-3	7
Alert-4	10
Alert-5	13

Table 5.2: Alerts

Then we run the four scripts to fetch the data the application needs to render the history page for the first alert 300 times. After that, we run the four scripts another 300 times each to get the data needed for the second alert. Then, we repeat the process for the rest of the

alerts. Next, we measure the duration of each request by registering the time before making each request and then registering the time after receiving the response. Then, we subtract the two times. Finally, we save the measurements into a file. We use the generated data to calculate the means, standard deviations, and lower and upper confidence interval bounds for the latencies. This method is similar to the experimental setup of [18]. First, the authors of [18] measure the response times while varying the concurrent users' numbers and then use the times to calculate the means. Eventually, they use the means to present their results. This method is quite similar to our method. However, we calculate the standard deviations and the confidence intervals to give a more accurate representation of our results. The paper compares the performance impact when the system handles several users simultaneously.
We create four different experiments to measure the performance differences between the four different solutions (gRPC, REST, GraphQL, and current solution) in different network conditions 5.1. We use the five alerts we mentioned earlier in these experiments.

Experiment 1

In this experiment, we fetch the history of each alert 300 times in the Normal network condition, which is the WiFi we have, without manipulating the network configurations.
After that, we repeat the previous experiment in a 3G network and finally in an edge network to measure the latency for the different solutions in different network conditions.

Experiment 2

We run the scripts to fetch a single user's information 300 times. Then, we repeat the experiment for all three different network conditions. This experiment aims to study the possible communication overhead that the BFFs might introduce because fetching the user's information involves contacting one service only.
The first script fetches the user's information according to the following sequence diagram:5.3

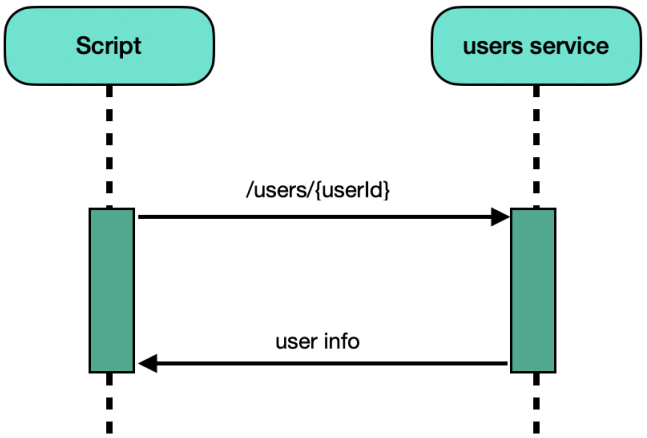

Figure 5.3: Current solution script

The rest of the scripts send a request to the BFF and then the BFF contacts the users service

and returns the response back to the scripts. See the call graph:5.4

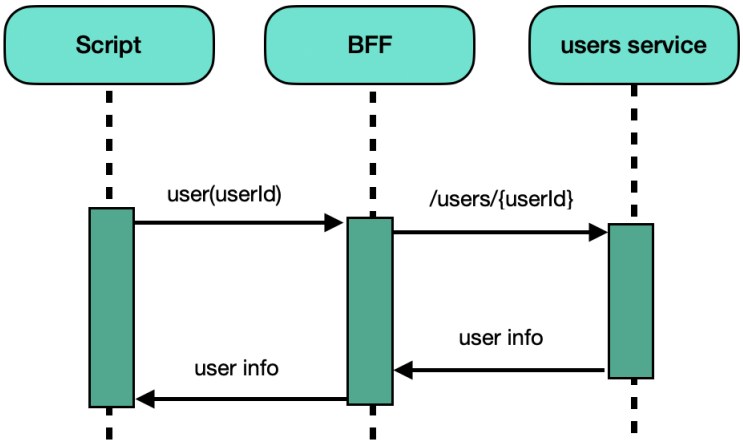

Figure 5.4: Scripts that use BFFs

We measure the latency of fetching the user's information in different network conditions.

Experiment 3

In this experiment, we run the (REST, GraphQL, and the current solution) scripts 300 times each to fetch the data needed to render the alerts using the BFFs and the current solution while registering the packets' sizes using Wireshark. Then, we repeat the experiment, but this time we use the scripts to fetch the information for a specific user. Finally, we calculate the total packets' sizes means for every solution. This experiment aims to study the effect of the BFF pattern on the size of data clients upload and download when fetching data with different sizes and complexities using the different BFFs and the current solution. After running the scripts for each test scenario,i.e., each alert and user information, we filter the packets between the client and the backend system. Then we get the accumulated sizes of those packets by summing the sizes of all the packets between the client and the backend system.
Then we calculate the mean using the following formula:

$$mean = \frac{\text{accumulated packets' sizes}}{300}$$

This way we calculate the mean of the amount of data transferred to fetch the requested resources.

Experiment 4

We can not host the gRPC BFF in Qlik's staging due to restrictions in the Kubernetes Cluster's configurations. Therefore, we created this experiment to gather the metrics we need to compare the BFFs. In this experiment, we host the three BFFs on a computer (outside the Qlik's staging cluster) and run the scripts from a different computer. The scripts send requests to the BFFs. The BFFs contact the staging cluster to gather the information they need then they send responses back to the scripts. We fetch the histories of the different alerts and the user information while measuring the latency and the responses' sizes. We compare the results from this experiment with the other experiments to get an idea of how the gRPC BFF would perform in the staging cluster.

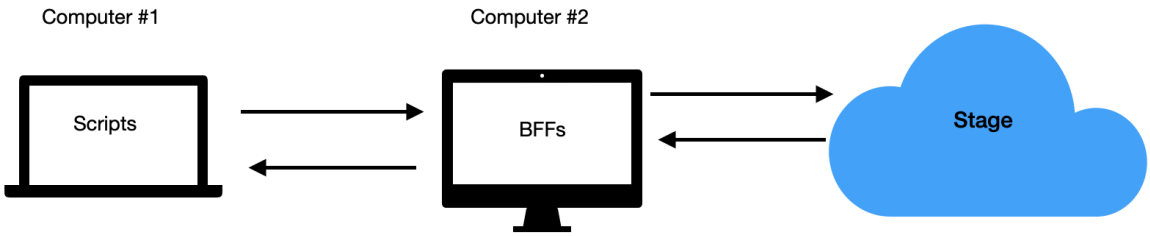

Figure 5.5: Experiment 4

5.4.2 Results

Experiment 1

Here we can see the results from the first experiment in the "normal" network condition. We created a table that shows which scripts we used, the mean standard deviation, 95% confidence interval and the number of iterations for each alert (See the tables in the appendix A A.1,A.2,A.3,A.4,A.5). The graph below visualizes the results from the first experiment, and we can see that the BFFs are significantly faster than the current solution that does not use a BFF. The confidence interval for the GraphQL BFF and the REST BFF overlap, and therefore, we can say that there is no significant difference in performance between the two BFFs.

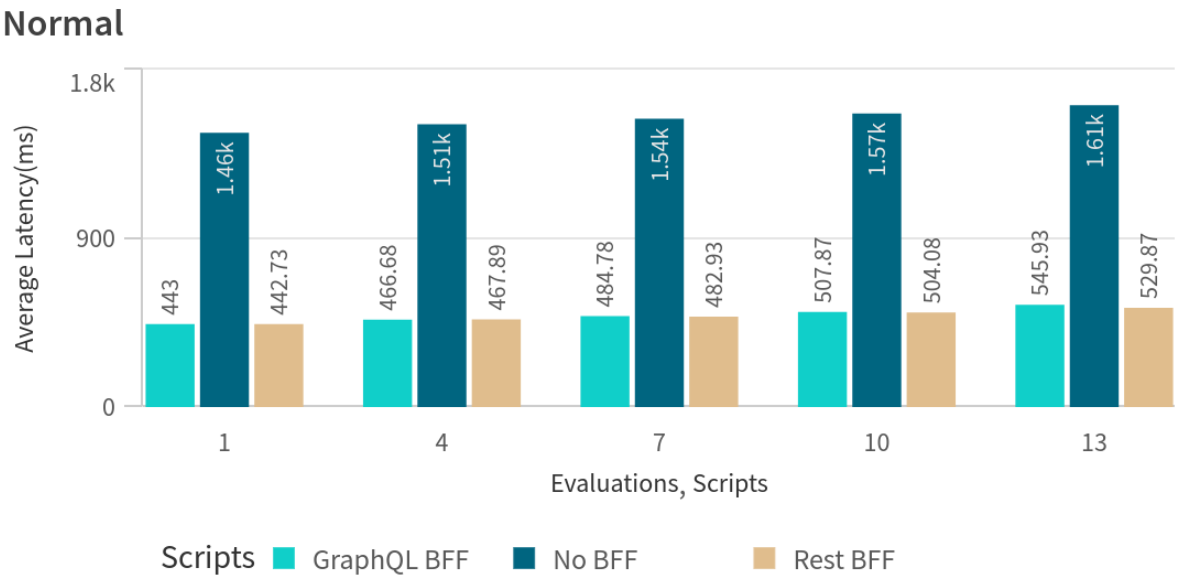

Figure 5.6: Latency in Normal Network

In this part, we can see the results that we obtained from fetching the histories in a 3G network. The tables for the different alerts can be found in appendix A (See the tables A.6,A.7,A.8,A.9,A.10).
The graph below shows the latencies of the different alerts' histories with the different scripts. We can see from the graph that the latency of the current solution increases when the number of evaluations increases. However, the number of evaluations does not significantly impact the latencies of the scripts that use the BFFs. When fetching the same alerts, the confidence intervals do not intersect for the different BFFs. Therefore, we can say that the REST BFF has lower latency than the GraphQL BFF.

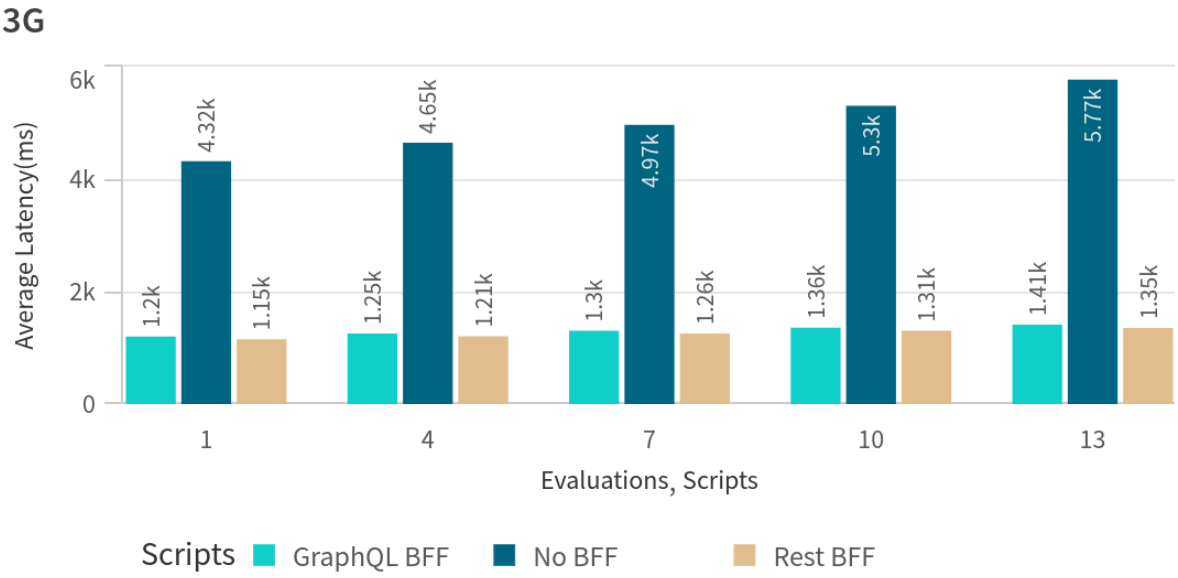

Figure 5.7: Latency in 3G Network

Finally, these are the results for the latencies in the edge network (See the tables in the appendix A A.11,A.12,A.13,A.14,A.15).
When fetching the same alert, the confidence intervals do not intersect for the different BFFs. Therefore, the REST BFF has lower latency than the GraphQL BFF. The graph shows the average latencies for the different solutions when fetching the different alerts' histories.

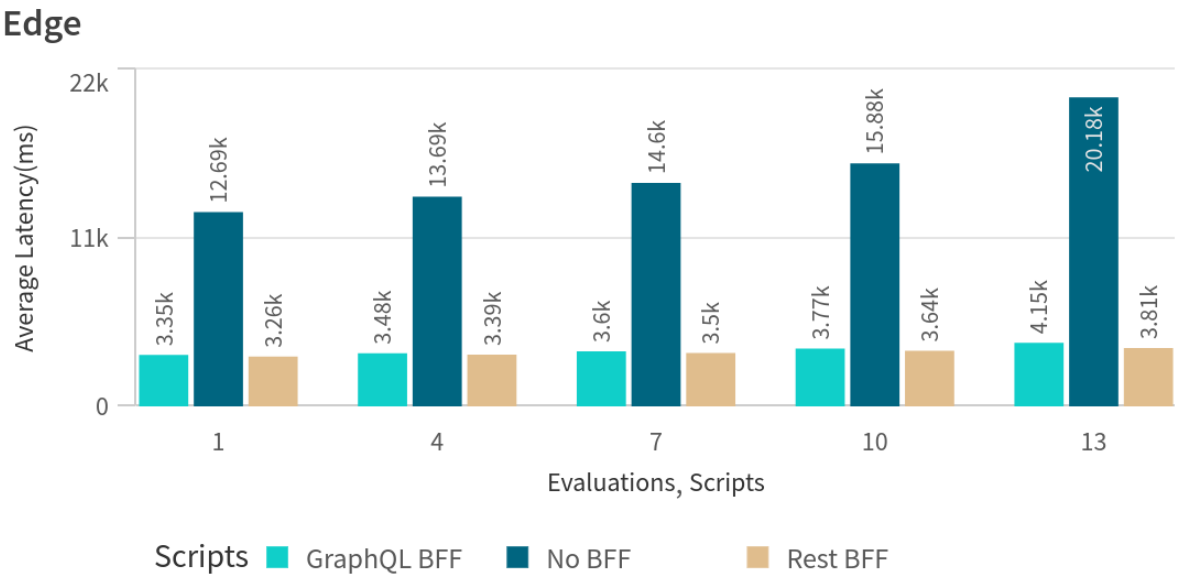

Figure 5.8: Latency in Edge network

Experiment 2

This subsection displays the results of fetching a specific user using the different scripts we have and in different network environments. The tables are available in appendix one under Experiment 4 A.16,A.17,A.18.
The graph below shows the latencies of fetching the user using the different scripts in the different network conditions. We can see from the graph that even with small requests that only require one API call to complete, the BFF does not introduce any significant latency to fetching

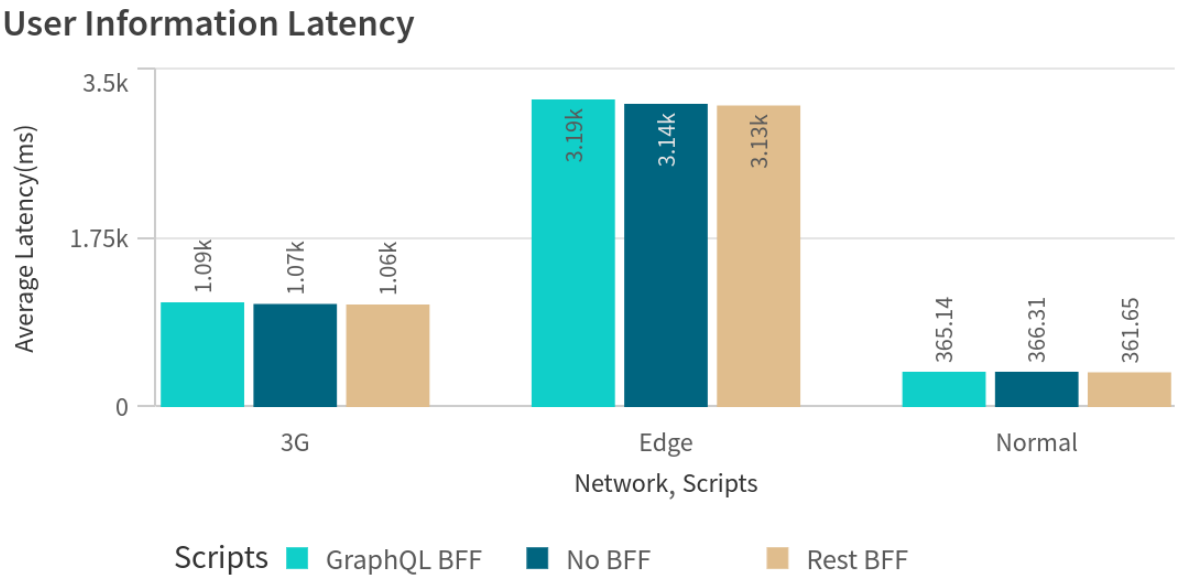

Figure 5.9: Latency for fetching user' information in different networks

Experiment 3

The results of measuring the amount of data the scripts send and receive when fetching the different alerts are available in the tables in appendix one under Experiment 5 A.19. The graph below plots the amount of data the different scripts used to get the information they needed to render the different alerts. We can see a significant reduction in the data sizes in the scripts that used the BFFs.

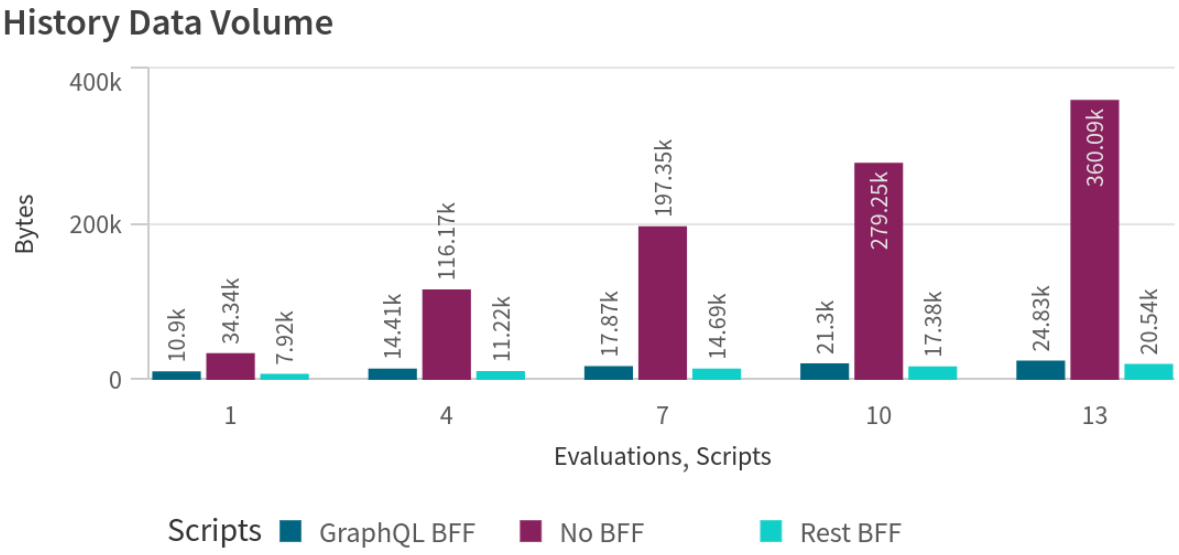

Figure 5.10: Amount of the used data to fetch alerts' histories (Bytes)

The following table A.20 and the graph below shows the amount of data the scripts used to fetch a certain user information. We can see from the graph that the script that uses REST script and the current solution use almost the same amount of data. On the other hand, we can see that the GraphQL script uses almost 20% more data than the other scripts.

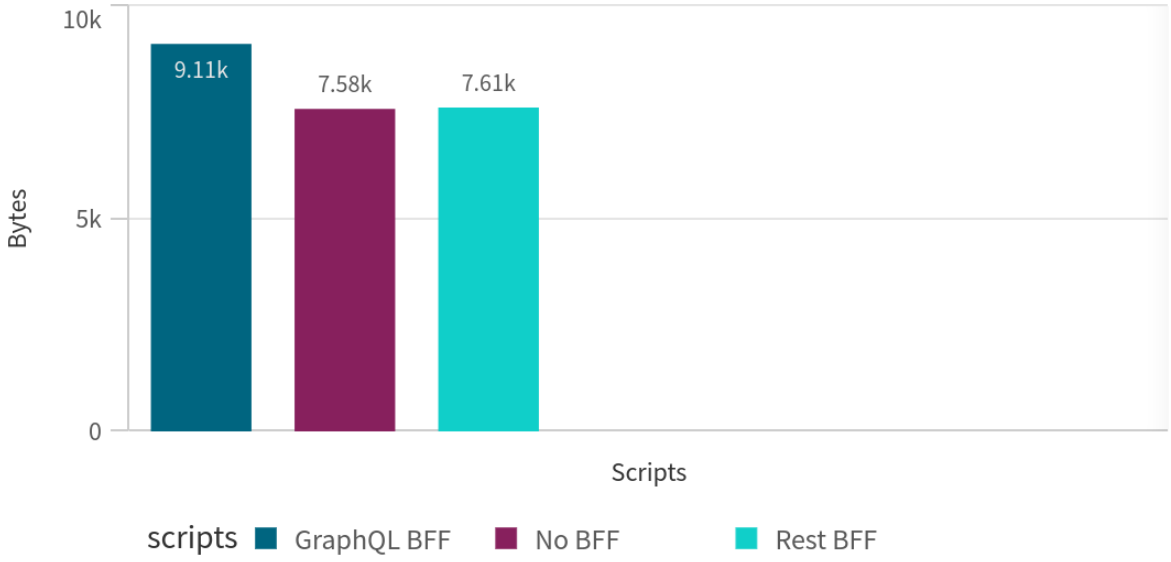

Figure 5.11: Amount of uploaded and downloaded data to fetch user information (Bytes)

Experiment 4

These are the results for fetching the different alerts' histories in a normal network condition (See the tables in appendix A:A.21,A.22,A.23,A.24,A.25)
We can see from the tables that the confidence intervals for all the different scripts overlap when fetching the same alerts. As a result, we can not see a significant difference in the latencies for the different scripts. The graph below shows the mean values of the latencies: 5.12

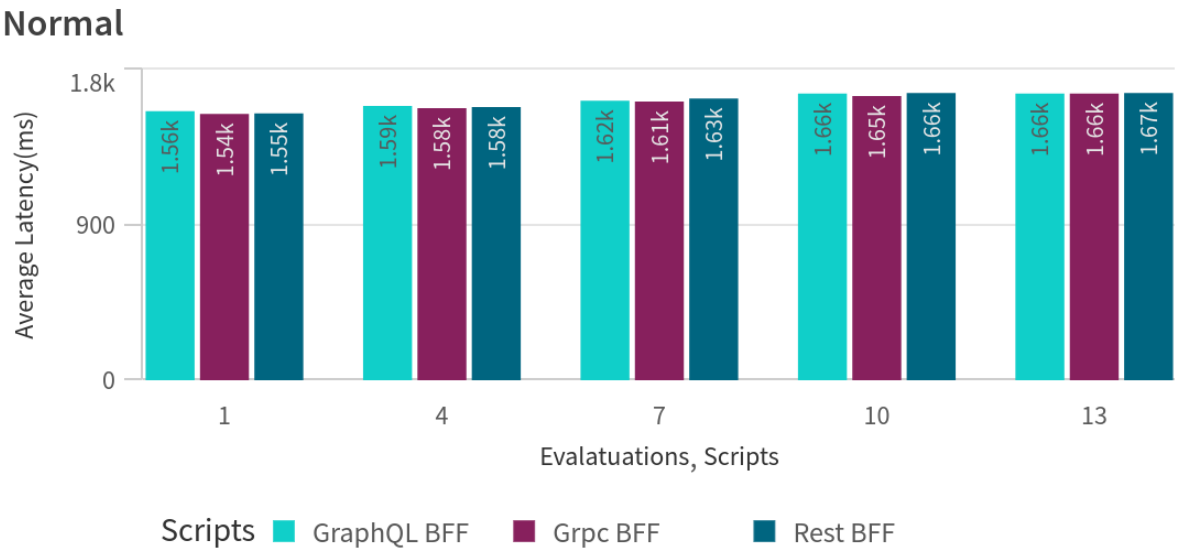

Figure 5.12: History Latency in Normal Network

Here we can see the results that we obtained from fetching the histories using the different scripts in a 3G network. The tables for the different alerts can be found in appendix A(See the tables A.26,A.27,A.28,A.29,A.30).
The confidence intervals for the different scripts do not overlap when fetching different alerts, except for the alert with four evaluations. However, the GraphQL BFF and the REST BFF overlap for that alert. The graph below shows the latencies of the different alerts' histories with the different scripts 5.13 Furthermore, we can see that the gRPC BFF is the fastest, followed by the REST BFF.

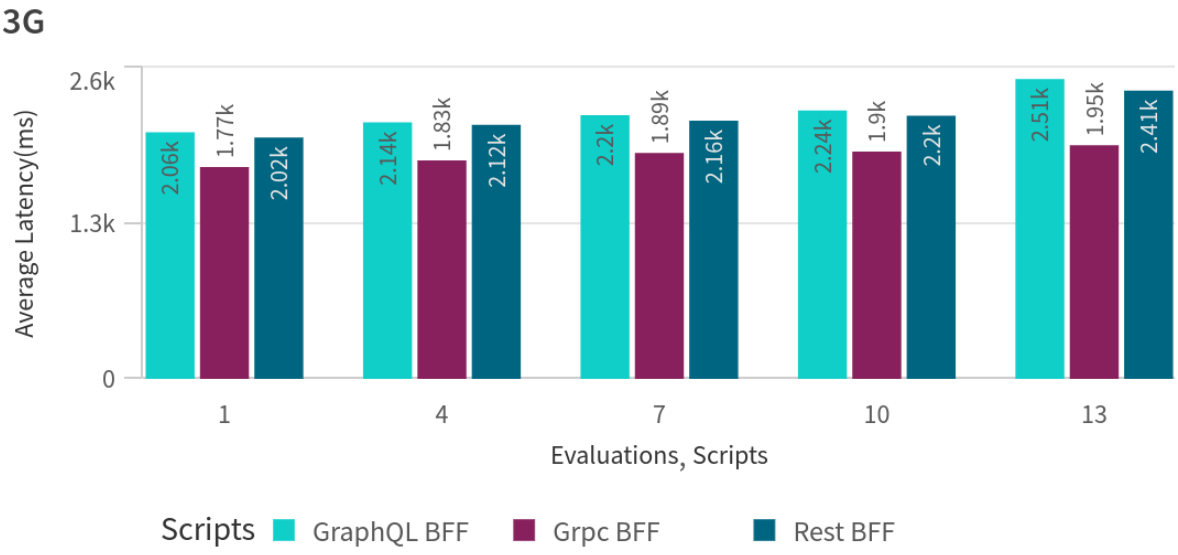

Figure 5.13: History Latency in 3G Network

Now we can see the results that we obtained from fetching the histories using the different scripts in an Edge network. The tables for the different alerts can be found in appendix A(See the tables A.31,A.32,A.33,A.34,A.35).
The confidence intervals for the different scripts do not overlap when fetching different alerts. The graph below shows the latencies of the different alerts' histories with the different scripts 5.14. We can see that the gRPC BFF is the fastest and the second fastest BFF is the REST BFF.

Figure 5.14: History Latency in Edge Network

Here we can see the latency results for fetching the user information using the three different BFFs. The tables are in Appendix A (See A.36, A.37A.38).
We can see in the tables that the confidence interval overlaps for the three scripts when fetching the user in a normal condition. Also, the confidence interval for GraphQL and REST overlap when fetching the information in the edge network. Therefore we can say that the performance of the GraphQL, REST, and gRPC BFFs are similar in a normal network condition. However, in the slower networks, we can see that the gRPC BFF outperforms the other BFFs. We can see the latencies in the graph below 5.15

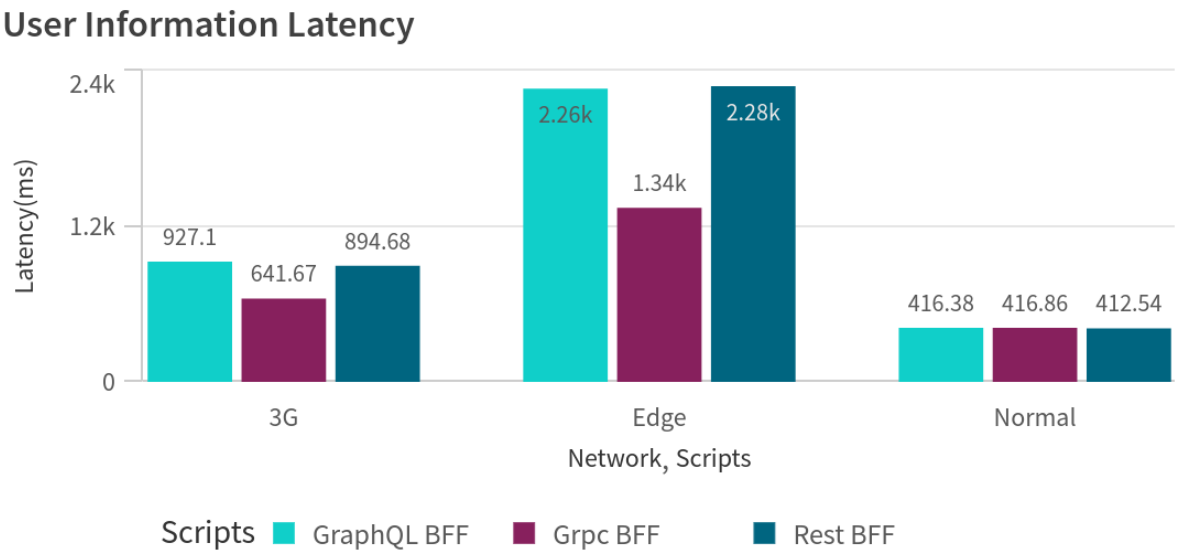

Figure 5.15: User Latency in all Networks

The table in Appendix A (see A.40) shows the amount of data that the scripts used to fetch the alerts' histories. The Graph below 5.16 visualizes the data in the table. We can see that the REST script uses less data than the GraphQL script and that the gRPC script uses the least amount of data compared to the other scripts.

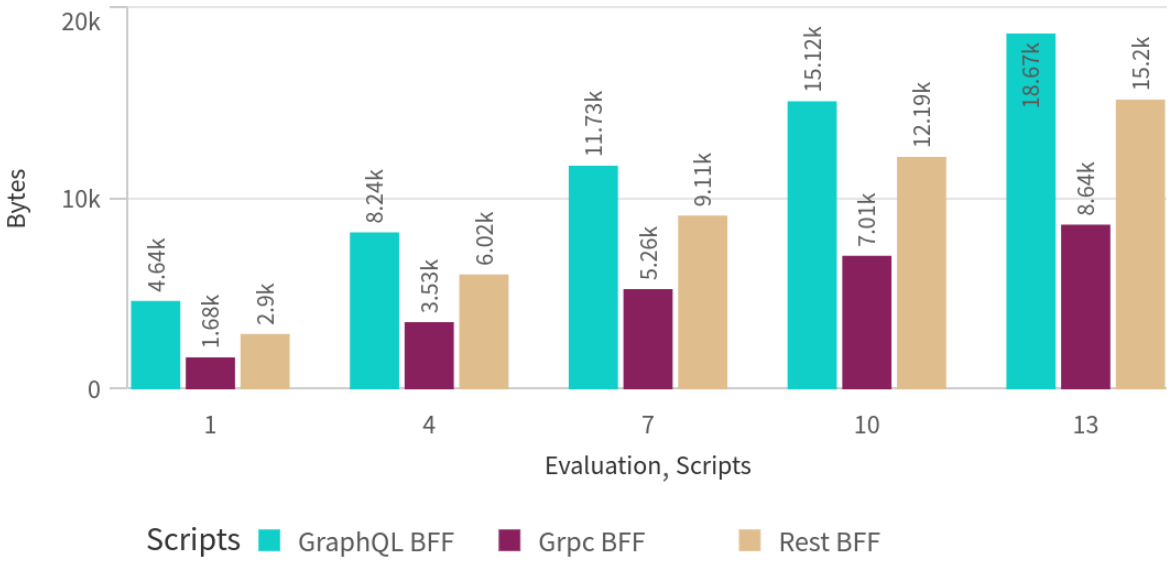

Figure 5.16: History Data Usage

The table in Appendix A (see A.39) shows the amount of data that the scripts used to fetch the user's information. The Graph below 5.17 visualizes the data in the table. We can see that the REST script uses less data than the GraphQL script and that the gRPC script uses the least amount of data among the three scripts.

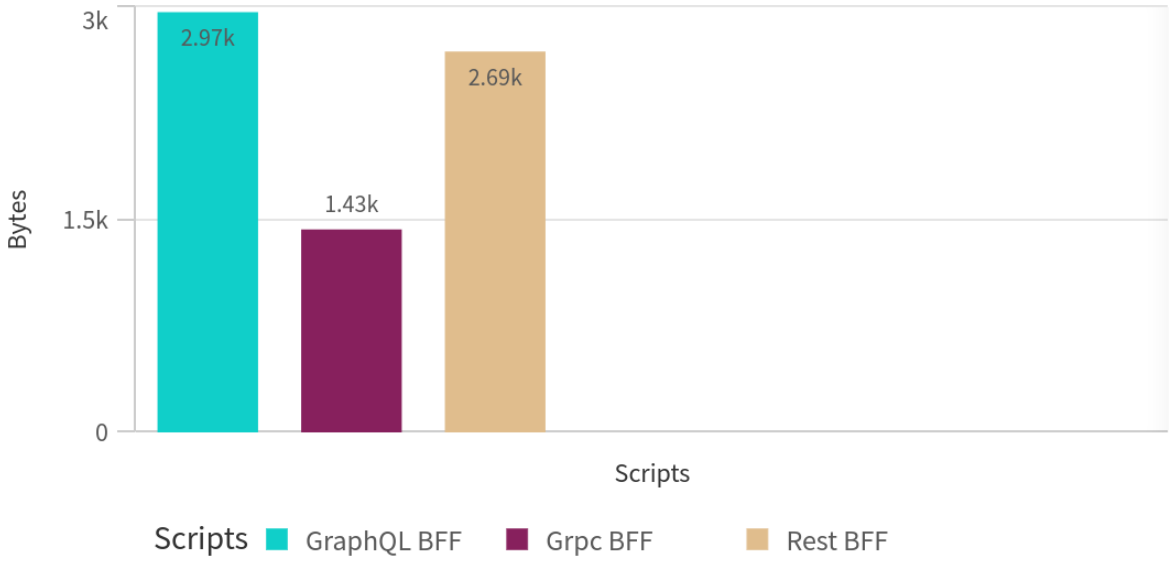

Figure 5.17: User Data Usage

5.4.3 Discussion

Experiment 1

We can see from the first experiment that all the BFF scripts outperform the current solution in all network conditions. However, if we compare the different scripts' latencies, we see that in the normal condition, there is no significant performance differences between the different scripts. As we mentioned earlier, the BFF scripts make one request to fetch the history data of an alert regardless of the number of evaluations that alert has. On the other hand, the current solution makes 2N +1 requests –where N is the number of evaluations– to fetch the data it needs. The current solution uses HTTP 1.2 without the keep-alive option, which means that every HTTP request starts a new TCP connection with a new TLS handshake (encryption), and as a result, each of these requests increases the overall latency of the communication.

In the 3G and the Edge network, we can see from the graphs 5.7 and 5.8 that the REST script has lower latency than the GraphQL script. We can explain this by looking at the graph 5.10 that shows the amount of data the client uses to fetch the alert history. We can see that the REST script uses less data, which means that it uses fewer packages, and in slower

networks, this can decrease the latency of the communication.

Experiment 2

In the second experiment, we measure the latency for fetching a user's information using the two scripts that use BFFs and the current solution. This experiment aims to see whether the BFFs add any extra latency when fetching resources that require only one request to fulfill. As we can see from the graph 5.9, in all network conditions, there is no significant difference in the latencies of the three different approaches, which means that the BFFs do not increase the latency significantly even when we use them to fetch data that require few numbers of requests to fulfill. In the current solution, the client makes one request to the users service directly, whereas in the BFF approach, the client makes one request to the BFF and the BFF makes another request to the users service. However, the experiment shows that the BFF does not add significant latency to the requests. This is because the BFF lives in the same cluster as the Users service, and they are on the same network. Therefore, the latency between the BFF and the users service is negligible compared to the latency between the client and the cluster.

Experiment 3

In the third experiment, we measure the amount of data the client uses to fetch the data it needs for the client and the different alert histories. In the alert history case, we can see in the graph 5.10 that the BFF scripts use significantly less data than the current solution. This can be explained by the fewer requests the BFF scripts make compared to the current solution. Also, the BFFs send back smaller JSON responses than the current solution because the BFFs customize the response to fit the client's need and only send back the relevant fields, which prevents over-fetching. On the other hand, we can see from the same graph 5.10 that the REST script uses less data compared to the GraphQL script, and this can be because the GraphQL client sends a relatively large query with every request to tell the BFF what fields the client needs.

In the users' case, we can see from the graph 5.11 that the current solution and the REST script use a similar amount of data, while the GraphQL script uses more data than the other two. Here we see that since all the three approaches make one request only to the backend, all three approaches open a single connection with the backend system. Therefore, the advantage that BFF scripts had with the alert histories does not exist with the users' service requests. Also, with the user information situation, the client consumes all the data that the users' service sends, which means that the JSON responses the BFFs send back and the ones the users' service sends back are almost the same. The increase in the GraphQL script data usage can be due to the GraphQL client including a query with every request to the BFF.

Experiment 4

We could not host the gRPC BFF in the staging environment because the staging environment is not configured to expose a gRPC service externally. Therefore, we created experiment number 4 to compare the REST and the GraphQL scripts with the gRPC script. In the first graph 5.12, we can see that there is no significant difference in the different scripts' latencies when fetching the different alerts in the normal network condition. In slower network conditions, i.e., 3G 5.13 and Edge 5.14, we can see that similar to the results from experiment 1, the REST script has lower latencies when fetching the histories of the different alerts. However, the gRPC script has even lower latencies than the REST script. The reason behind this latency reduction in the gRPC script's case can be that the gRPC client uses less data when fetching the data needed to render the alerts' histories 5.16. Consequently, the communication requires fewer packages, which can reduce the latency of the entire communication.

In the user's information case, we can see from the graph 5.15 that all three scripts have similar latencies when fetching the user's information in a normal network condition. However, in slower networks, i.e., 3G and Edge, the gRPC script has lower latency when fetching the user's information, and also this can be due to the lower data usage in the gRPC script's case, as we see in the following graph 5.17.

The data usage experiment shows that the REST script uses less data than the GraphQL script. However, the gRPC script uses even less data than the REST script. As mentioned earlier, gRPC uses HTTP 2.0 protocol to communicate compared to GraphQL and REST.HTTP 2.0 uses HPACK [32] that specifies how to compress HTTP headers consequently HTTP 2.0 eliminates redundant information for the header packets, and as a result, HTTP 2.0 uses less data[10].

When comparing the results from experiment 4 and the previous experiments, we can understand how the gRPC BFF would perform in the staging environment. The results deduct that the gRPC BFF would have lower latency than the GraphQL and the REST BFFs in slower network conditions. Moreover, the gRPC BFF would use less data when fetching users' information and alerts' histories.

5.4.4 Threat to Validity

Some confidence intervals intersect for the latencies, indicating that we can not determine which solution has lower latency. There are several ways to decrease the length of a confidence interval, one of which is increasing the number of iterations. In our experiments, we chose 300 as the number of iterations, and this is due to the long time and the large number of requests each iteration sends to the backend system. We have five alerts and combined they have 35 evaluations. Also, we have three different network conditions and three different solutions. This means that in each iteration, the backend makes

$$(35 \times 2 + 5) \times 3 \times 3 = 675 \quad \text{Requests}$$

So in 300 iterations the system makes

$$675 \times 300 = 202500 \quad \text{Requests}$$

We see that the first experiment only results in 202500 requests. The other experiments invoke many more requests in the backend system. Also, looking at alert history latency tables in the Appendix, we can see that fetching the history evaluation of an alert takes, on average 3.64 seconds. Therefore, fetching the history of five alerts using three different scripts in three different network conditions takes around

$$5 \times 3 \times 3 \times 3.64 = 49140 \quad \text{Seconds} \rightarrow 13.67 \quad \text{Hours}$$

After that, we need to conduct the second, third, and fourth experiments, which are also time-consuming. From all the above, we can see that the experiments are resource and time-consuming, and therefore we could not increase the number of iterations. Moreover, increasing the number of iterations has a small impact on the confidence interval's length, which means that the number of iterations must be increased significantly to significantly reduce the confidence interval, especially when the mean values are so near to each other. Moreover, we experimented in Qlik's staging environment. Therefore, more elements were involved in the process, i.e., a load balancer, authentication service, metrics, and tracings gathered from the BFFs, which we did not mention in this thesis. We wanted to test the BFF pattern in a real production environment where these elements are present in most microservices patterns. Since the staging environment mirrors the production environment, we experimented there. Also, we implemented tracing and metrics the same way for all the BFFs. Moreover, the GraphQL BFF and the REST BFF had the same configurations in the cluster to reduce the cluster's effect on the BFFs. For the gRPC BFF, we could not add the BFF to the cluster due to configuration issues, and therefore, we did a new experiment to compare it with the other BFFs.

5.5 Implementation Evaluation

In this section, we are interested in measuring the impact of the BFF design pattern on the coupling between the different components of the system, i.e., the backend services, the BFF, and the client. Also, we are interested in measuring the effect of the BFF design pattern on the client's code base. Therefore, we create two experiments to do the measurements.

5.5.1 Coupling Experiment

In this section, we are interested in measuring how the BFF design pattern affects the coupling of the system, i.e., between the microservices, BFFs, and the clients. We study the effects of some changes that can take place in similar systems and whether we need to change the codebases of the BFFs or the clients to adjust to those changes. The changes that we will investigate are the followings:

1. We assume that the address of the data-alerts service is changed.
2. We assume that the name of some fields in the users service response is changed.
3. We change the clients to request more fields about the users.
4. We change the client to request data about alert conditions. The data comes from a new service not implemented in the BFF.

5.5.2 Code Metrics Experiment

We want to study the code complexity impact of using different BFF technologies on the clients' code bases. Therefore, we apply software complexity metrics to each client's source code. These are the complexity metrics we are using:

Lines of Code(LOC)

The LOC metric is the lines of code in the source code of a program. It is easy to calculate and understand. However, the LOC metric does not consider the complexity of each line of code. For instance, LOC does not differentiate between "i=1" and "i = func(1+2,3)/func(5);" [39].

Halstead Complexity Metric (HCM)

HCM is calculated using the number of operators and operands in the source code. Operators are symbols used in expressions to manipulate the operands[39].
In this experiment, we calculate the metrics mentioned below for each function involved in the process of fetching the needed data.
HCM uses the following metrics:

- n_1 = the number of unique operators
- n_2 = the number of unique operands
- N_1 = the number of all operators
- N_2 = the number of all operands

to calculate the following metrics:

- Volume: $V = (N_1 + N_2) * log_2(n_1 + n_2)$
- The software's ideal volume: $V^* = (n_1 N_2 / 2n_2)(N_1 + N_2) log_2(n_1 + n_2)$
- Programming Difficulty: $D = V/V^*$
- Programming Effort : $E = V * D$
- Error Estimate: $B = V/S^*$
- S^* is the programmer's ability, Halstead sets it to 3000.

[39]

Cyclomatic Complexity Metric (CCM)

CCM represents the number of linearly independent paths in the control flow graph of a program's source code, thus, it indicates the minimum paths that the programmers should test. The formula to calculate CCM is $CCM = e - n + 2$ where e is the number of edges and n is the number of nodes in the control flow graph.[39]. According to [26] the control flow graph represents all the flow of control that may arise during the program's execution, and we can explain it with an example:

```
a = 0
b = 2
while a < b:
    b =- 2
print(b)
```

the control flow graph for the code above is the following 5.18:

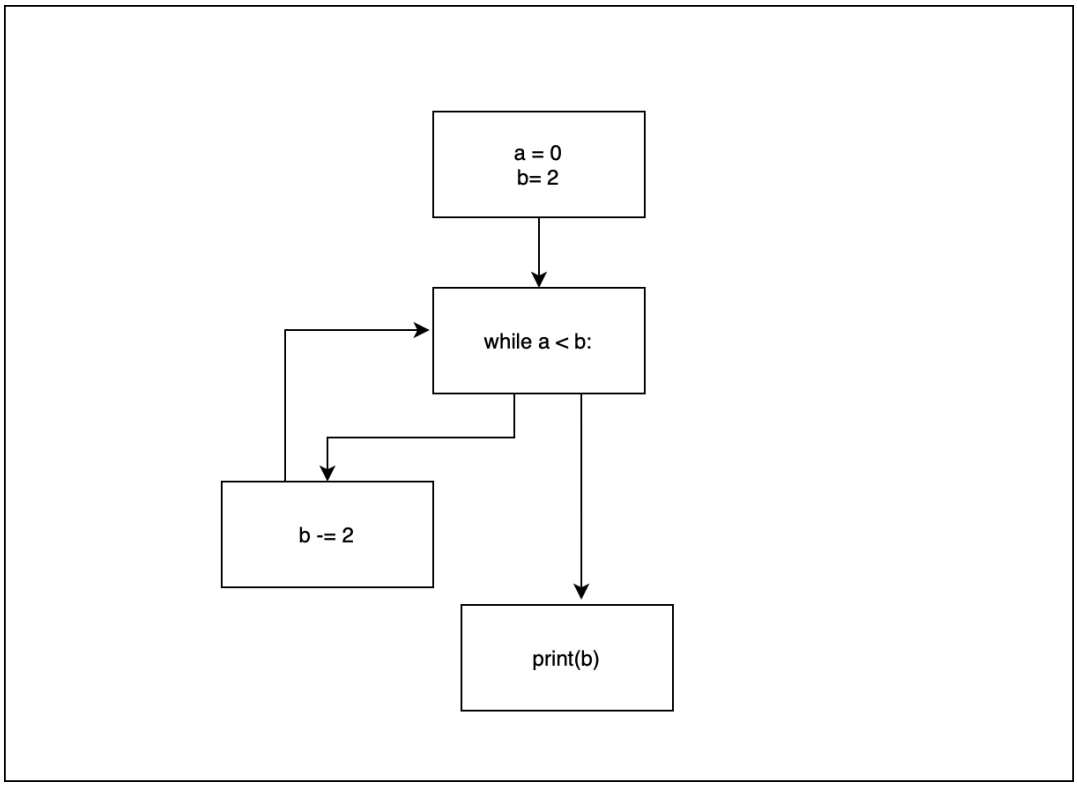

Figure 5.18: Control Flow Graph

5.5.3 Results

Coupling Experiment

Here we create a table to present the results of the coupling experiment. We put X next to the entity if the changes in the system require the developers to update the entity's codebase.

The changes:

1. The data alert service's address changes
2. We assume that the name of some fields in the users service response is changed
3. We change the clients to request more fields about the users
4. We change the client to request data about alert conditions. The data comes from a new service not implemented in the BFF.

Entity/ Changes	1	2	3	4
GraphQL BFF	X	X		X
REST BFF	X	X	X	X
gRPC BFF	X	X	X	X
No BFF Script	X	X	X	X
GraphQL Script			X	X
REST BFF Script			X	X
gRPC Script			X	X

Table 5.3: System Coupling

Code Metrics Experiment

The Lines of code table 5.4 shows the lines of code number involved in getting the history of an alert in each script. We can see that the current solution, i.e., the script that does not use the BFF, has the largest code base. The second-largest code base is the GraphQL script, which is due to the large query in the code.

Entity	LOC
No BFF	180
GraphQL BFF	101
REST BFF	42
gRPC BFF	23

Table 5.4: Alert Lines of Code Metrics

The table 5.5 shows the lines of code metrics for the script that fetches the user information. We can see that the largest code base is the GraphQL code base. This is due to the relatively large query that is sent in the post request.

Entity	LOC
No BFF	15
GraphQL BFF	51
REST BFF	22
gRPC BFF	20

Table 5.5: User Lines of Code Metrics

The table 5.6 shows the Halstead metrics for each alert history script where the lower the difficulty, programming effort, and estimated bug count, the better. We want to point out that the script that uses the current solution is the same code that Qlik uses in the mobile application, i.e., code in production. The script has eight different functions that get called to fetch an alert's history. The rest of the scripts use one function to get the required data. For simplicity, we refer to every helper function in the current solution as Function 1,2,3, and so on.

Type	Function	D	E	B
No BFF	Function 1	18.79	13806.74	0.191
No BFF	Function 2	2.37	266.87	0.0138
No BFF	Function 3	2.44	316.12	0.015
No BFF	Function 4	3.42	805.09	0.0288
No BFF	Function 5	4.16	594.26	0.023
No BFF	Function 6	9.21	3247.01	0.073
No BFF	Function 7	2.37	266.87	0.013
No BFF	getHistory	5.4	1218.35	0.038
GraphQL BFF	getHistory	2.88	392.30	0.017
REST BFF	getHistory	2.84	418.75	0.018
gRPC BFF	getHistory	0.593	116.809	0.007

Table 5.6: History Halstead Complexity

Entity	Function	D	E	B
No BFF	getUser	2.96	339.03	0.016
GraphQL BFF	getUser	2.58	295.57	0.014
REST BFF	getUser	2.33	177.33	0.010
gRPC BFF	getUser	0.625	89.974	0.006

Table 5.7: User Halstead Complexity

The table 5.8the cyclomatic complexities of the different scripts. If a script has more than one function, we summed the CCMs of all the functions in that script.

Entity	CCM
No BFF	35
GraphQL BFF	2
REST BFF	2
gRPC BFF	2

Table 5.8: History Cyclomatic Complexity

Entity	CCM
No BFF	2
GraphQL BFF	2
REST BFF	2
gRPC BFF	2

Table 5.9: User Cyclomatic Complexity

5.5.4 Discussion

Coupling Experiment

The first experiment shows that the GraphQL BFF requires the least number of releases in the four scenarios that we have suggested. The GraphQL BFF allows the client to specify the fields it needs in every request. As a result, if the GraphQL client needs different fields in a response, as long as the GraphQL BFF has the fields in its schema and the microservice still serves that field, there is no need to release a new version of the BFF. On the other hand, the same does not apply to gRPC and REST. because with those two, the fields that the BFFs send are hard-coded into the BFFs, and therefore, the BFFs have to be updated if the client decides to ask for more or even fewer fields.

For the clients, the experiment showed that the BFF design pattern reduces the coupling between the client and the backend system. This is because we moved all the logic dependent on the internal services into the BFF instead. As a consequence, the client became more independent from the internal services. This means that we can reduce the number of client releases related to changes in the backend system.

Code Metrics Experiment

The second experiment shows that for the alert history fetching scripts, the BFF design pattern reduces the code complexity of the client code. We also see that all the BFF scripts have fewer lines of codes compared to the current solution script(see table:5.4). Also, when we look at the Halstead complexity metrics in table 5.6, we can see that the current solution uses eight functions to fetch the alert history data. Each of those functions needs to be tested and maintained. Each BFF script has only one function to fetch the needed data. If we compare the Halstead metrics (See table: 5.6) for the BFF scripts functions and the current solution

function, we see that the BFFs' source codes are easier to maintain and have on average lower estimated bugs number than the current solution code. On the other hand, for the scripts that fetch the user's information, we can see that the BFFs do increase the length of clients' source code compared to the current solution (See table 5.5) because the BFFs and current solution clients only send one request to the backend. The current solution does not need to format the response. However, the increase in the code lines does not necessarily mean an increase in code complexity. If we look at the Halstead metrics for the user code in the table 5.7, we see that the difficulty, programming effort, and estimated bug count are lower for the BFFs clients compared to the current solution client. This can indicate that the BFF clients would be easier to maintain compared to the current solution client code.

5.5.5 Threat to Validity

The Halstead metrics, lines of code, and Cyclomatic complexities guide programmers to write more maintainable and readable software. They do not give objective results that can be used to compare to methods because programmers can refactor any piece of code to alter those results. However, to minimize the effect of the programming style in the metrics, one programmer wrote the code of all the scripts, and that programmer followed the same coding style and design pattern in all the scripts.

5.6 BFF Security Evaluation

We conduct the security evaluation by doing a threat analysis on the BFFs. After that, we recommend security requirements to mitigate the effects of the threats we find.

5.6.1 Threat Analysis

First, we create a Data flow diagram (DFD) to visualize all the processes and entities in the system and how data flows between them, and then we set our different trust boundaries. Next, we use the STRIDE[35] method to conduct our threat modeling on all the data that flows across different trust boundaries.
STRIDE is an abbreviation for:

- Spoofing: is an attack where an adversary impersonates a legitimate user in the system
- Tampering: is an unauthorized modification of information or processes in the system.
- Repudiation: is to dismiss responsibility of action.
- Information Disclosure: is when the attacker gets access to confidential information.
- Denial of Service: This attack overwhelms the system with illegitimate requests or requests to prevent the system from responding to legitimate ones.
- Privilege Escalation: is when a user can access more information or services than they are allowed.

To conduct threat enumeration using STRIDE per interaction that considers tuples of (origin, destination, interaction) and enumerates threats against them [37]. First, we identify all the interactions that cross the trust boundary in DFD5.19. After we identify all the interactions, we create a stride table to identify the S.T.R.I.D.E threats applicable to each interaction, and we put a checkbox for each valid S.T.R.I.D.E risk for every interaction as we did in this table 5.10. For example the interaction "BFF has inbound data flow from a client." is susceptible to Spoofing, Denial of service and Privilege Escalation, So the table for that interaction would be:

#	Element	Interaction	S	T	R	I	D	E
1	BFF	BFF has inbound data flow from a client	X				X	X

After creating the STRIDE table, we create a new table where we place a threat per checkbox (see table: 5.11).

Security Assumptions

The readers need to understand why we avoid analyzing specific system parts, i.e., the internal services and Ingress. Therefore, we state the reasons to avoid those analyses in the security assumptions we have regarding the system.

1. Ingress only forwards the requests and responses between the client to the BFF, and therefore, we treat every data flow between the client and the BFF that goes through Ingress as a data flow between the client and the BFF directly.

2. The entire cluster is hosted on a secure network. Therefore we do not consider attacks from the microservices or attacks on the data flow inside the cluster.

Data Flow Diagram

We represent every service and entity that we do not have control over as a rectangle, process that we can control as a circle, and data stores as parallel lines.

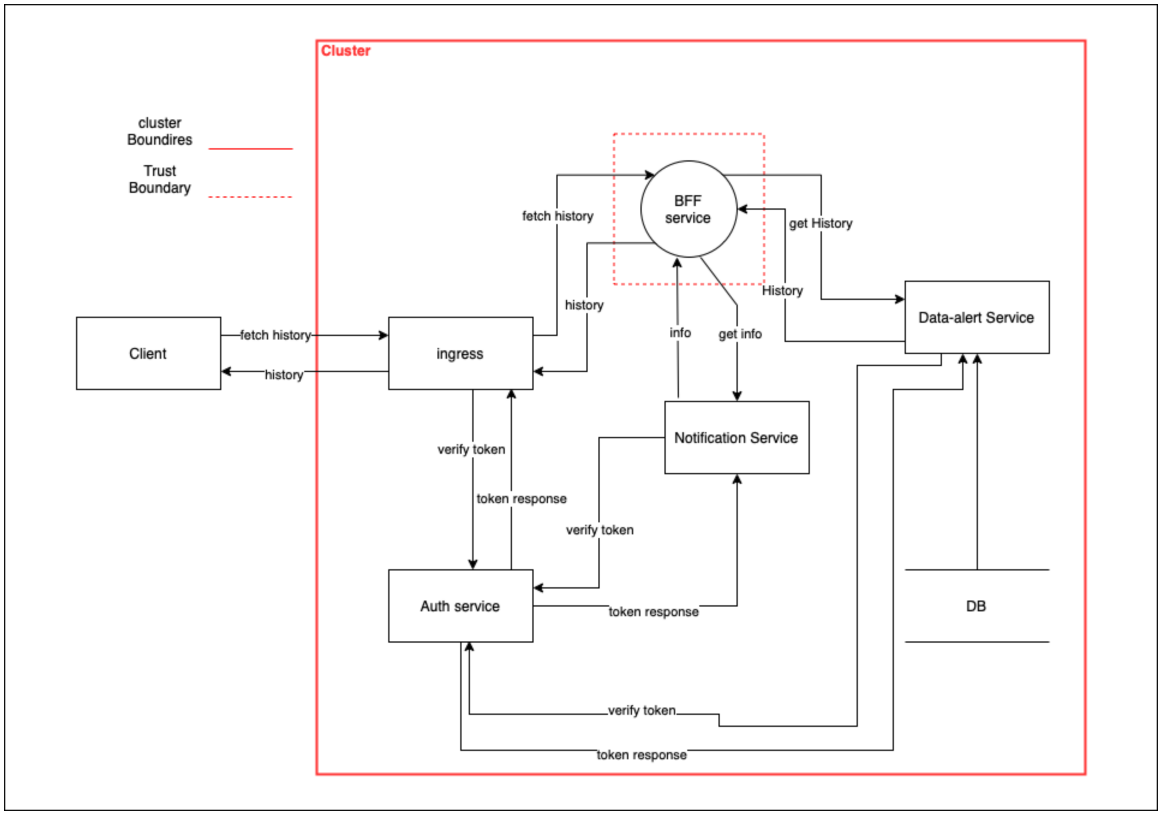

Figure 5.19: Data Flow Diagram for fetching the history

5.6.2 STRIDE Table

When creating a STRIDE table we are only interested in interactions that cross trust boundaries.
Elements are the different entities in the system.

#	Element	Interaction	S	T	R	I	D	E
1	Process(BFF)	BFF has inbound data flow from a client	X				X	X
2		BFF has outbound data flow to the client	X		X	X		
3		BFF has inbound data flow from Data Alert service	X				X	X
4		BFF has outbound data flow to Data Alert service	X		X	X		
5		BFF has inbound data flow from Notification service	X				X	X
6		BFF has outbound data flow to Notification service	X		X	X		
7	External Interactor (Client)	Client has inbound data flow from the BFF through Ingress	X					
8		Client has outbound data flow to BFF through Ingress	X		X			
9	Data Flows	Data flow between the client and Ingress		X		X	X	
10		Data flow between the BFF and Data Alert service		X		X	X	
11		Data flow between the BFF and Notification service		X		X	X	
12	External Interactor (Data Alert Service)	Service has inbound data flow from the BFF	X				X	X
13		Service has outbound data flow from the BFF	X		X	X		
14	External Interactor (Notification Service)	Service has inbound data flow from the BFF	X				X	X
15		Service has outbound data flow from the BFF	X		X	X		

Table 5.10: Stride Table

#	S	T	R	I	D	E
1	BFF not sure of client's identity				BFF might crash or Client overwhelms the BFF with requests	Client sends malicious request to force BFF to request internal services with BFF privileges
2	Attacker pretends to be the user		Client dismisses receiving any thing from the BFF	BFF sends data to unauthorised client		
3 . . 6	Not in the scope of this paper					
7	Attacker pretends to be the BFF					
8	Client not sure about the identity of the BFF		The BFF dismisses receiving anything from client			
9		Attacker can tamper with packets		Attacker can sniff packets	Attacker can interact with the TCP packets to stop the communication	
10 . . 15	Not in the scope of this paper					

Table 5.11: Stride Table

5.6.3 Threat Mitigations

In this section we provide some security requirements for the BFF and the entire system, and in the security requirements we use the certain keywords to indicate requirement levels according to RFC 2119 [5] and these keywords are the following:

- **Must** means that the definition is an absolute requirement of the specification
- **Must not** means that the definition is an absolute prohibition of the specification
- **Should** means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- **Should not** means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

1. TLS

Transport Layer Security a.k.a TLS [20] encrypts the communication channels between the cluster and the client. We **should use** TLS 1.3 and if that is not applicable, TLS 1.2 **must** be used instead. Also, for the client would be able to authenticate the cluster, a trustworthy certificate authority (a.k.a CA) must sign the server's certificate.

2. Authentication Tokens

The client **must** send an authentication token (a.k.a auth token) with every request to the cluster. When the request reaches the cluster, the authentication service **must** verify the token before it reaches the BFF, and then the BFF **must** forward the token to the internal services. The internal services **must** also send the token to the authentication service to verify the token. The BFF will have no privileges that an attacker can abuse.

3. Logging

The BFF **must** log every event that the client takes. Also, the BFF **must** log every request and response. The logs are helpful to prove any action the users make, and it helps developers to find bugs. Moreover, extensive logging can help security experts determine an adversary's actions in case of a security breach. We **must not** log any data that the General Data Protection Regulation a.k.a GDPR [23] classifies as a personally identifiable information (a.k.a PII) e.g., usernames, emails or personal addresses.

4. Using cluster management tools and monitoring

The cluster management tools can automatically restart services if they crash without any human intervention. Also, service monitoring can flag any potential crashes of the BFF to the developers so that they can fix the bugs that caused the crashes. Therefore, we **should** use cluster management tool and monitoring.

5. Scaling and Cloud Hosting

Services **should** be hosted in the cloud, because cloud hosting makes it easier to scale the services both vertically by increasing the computing power for each server and horizontally by creating multiple replicas of the BFF. In addition, scaling allows the BFF to handle more traffic.

6. Time-outs to drop slow connections

If a request takes more time than a time limit, the BFF **must** drops the request, freeing up resources to handle other requests. Also, this can mitigate some attacks where attackers open many slow communication channels with the BFF, which can affect the service's capabilities to handle legitimate requests.

7. Restricting the IP addresses that the BFF can send a request to

The programmers **must** specify the IP addresses that the BFF can interact with to prevent attackers from using the BFF to contact any internal or external services and endpoints.

8. Not allowing the client to specify the services' addresses to the BFF

The BFF **must** either use a discovery service or a list of hard-coded IP addresses of the services it needs to contact and the BFF **must not** wait for the client to provide these addresses. Otherwise, the BFF would be vulnerable to Server Side Request Forgery attacks.

5.6.4 Threat-Mitigation Table

The 5.12 table shows us which recommendation mitigates which threat. The numbers in the mitigation row map to the threat mitigations in 5.6.3 and the numbers in the threats column map to the numbers in this table 5.11

Evaluate the BFF design pattern from a security perspective and provide recommendations that make the entire design pattern more secure.

Threats/ Mitigations	1	2	3	4	5	6	7	8
1.Spoofing		X						
1.Denial of Service				X	X	X		
1.Privilege Escalation							X	X
2.Spoofing	X							
2.Repudiation			X					
2.Information Disclosure		X						
7.Spoofing	X							
8.Spoofing	X							
8.Repudiation			X					
9.Tampering	X							
9.Information Disclosure	X							
9.Denial of Service	X							

Table 5.12: Threat-Mitigation Table

Chapter 6
Related Work

There are already research papers and articles investigating different ways of integrating frontend technologies into a backend consisting of several microservices. For example, Brown and Woolf [6] propose the BFF design pattern as a possible solution to connect different frontend application types such as mobile, desktop, and web application clients. According to the authors, the BFF design pattern can orchestrate several API calls that result from a single client action. Moreover, the BFF pattern can translate the microservices' responses so they fit the clients' needs, and it also can filter away data that the clients do not need[6]. Also Brown and Woolf state, that in most cases, to increase the development efficiency, the team responsible for the frontend client application is also responsible for the BFF because the client team does not have to waste time collaborating with other teams to add some changes to the BFF. Moreover, the client team knows best how to optimize the BFF to fit their client's needs. Hamrs, Rogowski, and Lo Iacono [17] compare different methods of implementing frontend architectures that are connected to a microservices backend. One of the methods they examine is integrating microservices using the BFF design pattern. According to the paper, the BFF design pattern can separate the development of the microservices and the client applications, and this makes it easier to modify different releases of the client applications[17]. Furthermore, we saw in our results 5.5.3 that the BFF design pattern decreases the dependencies between the client and the backend services, which makes it easier to introduce specific changes to the services without impacting the client. Sound Cloud implemented the BFF design pattern into their system back in 2013, and they wrote a blog [8] about their experience with the BFF pattern. The blog informs about the advantages and disadvantages of using BFFs. According to them, BFFs provide autonomy because having different BFFs for different client types enables the developers to optimize each BFF to fit its clients' needs. Furthermore, BFFs provide resilience in the system because a bad deployment can bring down a BFF, but it should not bring down the whole platform. As a consequence, high autonomy and lower risk increase the development speed. The blog points out that feature integration tends to end up in the BFFs in some cases. Consequently, the code for the feature gets duplicated in the different BFFs, which leads to inconsistent implementations that diverge with

time. Moreover, the blog states that there should be a strategy to determine the maximum number of BFFs a system can have because it is tempting to create too many BFFs, which increases the autonomy. However, creating too many BFFs increases the maintenance and operational overhead[8]. The following paper [29] discusses different methods to make the microservices design pattern more secure. It highlights that the authentication token must be verified at the microservices level. On the other hand, verifying the access token at the API gateway level makes the system vulnerable if the API gateway gets compromised. This is called the confused deputy problem. Since BFFs are very similar to API gateways, we can extrapolate this reasoning to apply to the BFF design pattern. Similarly, in our threat analysis 5.6, we state that the microservices must check the token validity when getting requests from the BFF, which would prevent an attacker from using the BFF to access services that the attacker does not have access to. Moreover, we suggest more measures that improve the BFF's security as a service and the entire infrastructure that the BFF is part of. All these papers contain important information about the BFF design pattern, and they inspect elements of interest to us in our thesis. However, in our thesis, we go into detail of what technologies to use when implementing a BFF. Moreover, we study the BFF's impact on the maintainability of the frontend and the BFF service. Furthermore, we create a threat model for the BFF design pattern to give a solid understanding of how to secure the BFF and mitigate the cyber security threats when implementing a BFF.

Chapter 7
Conclusion

This thesis evaluates the BFF design pattern as a possible solution to the performance-overhead the microservices pattern introduces. We found that the BFF design pattern mitigates the microservices overhead. It decreased the dependencies between the backend services and the clients because the client does not have to know about the different services and endpoints in the backend system. Also, in our case that the BFF reduced the client's code complexity because we moved any processing or formatting of the incoming responses from the client to the BFF. Moreover, we migrated the code that makes several requests to several services to the BFF. As a result, the client code had fewer potential bugs according to the Halstead metrics. Moreover, any changes in the shape of the requests or responses did not affect the client. As a result, the frontend team's productivity increased, and the number of releases related to those changes reduced.
When comparing the different BFFs and their effect on the client-backend dependencies 5.3, we saw that the GraphQL BFF had the least dependency among the three BFFs. However, when looking at the client code metrics (5.6, 5.7), we saw that the gRPC client has the least programming effort, difficulty, and estimated bug counts.

We show that the BFF design pattern significantly decreased the latency between the clients and the backend when fetching data requiring several backend requests, especially for clients with slower internet connections. When looking at the results for the latency of the different BFFs' clients in different network conditions, we found that for complex requests, in fast network conditions, the REST, GraphQL, and gRPC BFFs had similar latencies. However, we found that the gRPC client had the least latency in slower network conditions, followed by the REST and GraphQL clients. However, for data that do not require several requests to fulfill, we found that the gRPC client had the least latency 5.15. On the other hand, the current solution, REST and GraphQL clients, had similar latency results 5.9.

For the data usage, when fetching complex data that require several requests to fulfill, we

found that the BFFs significantly reduced the amount of data the client sends and receives 5.11 compared to the current solution where no BFFs are used. This reduces the data usage costs for clients operating on relatively expensive mobile networks. After comparing the data usage for fetching complex data 5.16, we found that the gRPC client used the least amount of data, then followed by the REST client and finally the GraphQL client. When comparing the data usage between the BFFs' scripts and the current solution when fetching data that does not require several requests, we found that the REST client uses a similar amount of data to the current solution. However, the GraphQL client used more data than the current solution and the REST clients. However, the gRPC client used fewer data to fetch the needed data. We also found that the BFF design pattern introduces security risks that need to be mitigated to make it harder for malicious users to abuse the system. In conclusion, the BFFs reduce latency, data usage, client-backend dependencies, and client codebase complexity.

References

[1] Akhan Akbulut and Harry G Perros. Performance analysis of microservice design patterns. *IEEE Internet Computing*, 23(6):19–27, 2019.

[2] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[3] Apollo. Introduction to apollo server. Available at `https://www.apollographql.com/docs/apollo-server/` (). Last Access 2022/1/24.

[4] Apple. App review. Available at `https://developer.apple.com/app-store/review/` (). Last Access 2022/2/14.

[5] Scott Bradner. Key words for use in rfcs to indicate requirement levels. Available at `https://datatracker.ietf.org/doc/html/rfc2119` (1997/03). Last Access 2022/5/11.

[6] Kyle Brown and Bobby Woolf. Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, pages 1–35, 2016.

[7] Romuald Corbel, Emile Stephan, and Nathalie Omnes. Http/1.1 pipelining vs http2 in-the-clear: Performance comparison. In *2016 13th International Conference on New Technologies for Distributed Systems (NOTERE)*, pages 1–6. IEEE, 2016.

[8] Jorge Creixell. Service architecture at soundcloud — part 1: Backends for frontends. Available at `https://developers.soundcloud.com/blog/service-architecture-1` (2021/07/29). Last Access 2022/1/17.

[9] Francisco Curbera, William Nagy, and Sanjiva Weerawarana. Web services: Why and how. In *Workshop on Object-Oriented Web Services-OOPSLA*, volume 2001. Citeseer, 2001.

[10] Hugues de Saxcé, Iuniana Oprescu, and Yiping Chen. Is http/2 really faster than http/1.1? In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 293–299, 2015.

[11] Sang Gyun Du, Jong Won Lee, and Keecheon Kim. Proposal of grpc as a new northbound api for application layer communication efficiency in sdn. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*, pages 1–6, 2018.

[12] Google. About app review. Available at `https://developers.google.com/workspace/marketplace/about-app-review` (). Last Access 2022/2/14.

[13] GooglePlay. Qlik sense saas. Available at `https://play.google.com/store/apps/details?id=com.qlik.qsm&gl=SE` (). Last Access 2022/2/14.

[14] graphql.org. Frequently asked questions (faq). Available at `https://graphql.org/faq/#is-graphql-owned-by-facebook` (). Last Access 2022/2/6.

[15] graphql.org. A query language for your api. Available at `https://graphql.org` (). Last Access 2022/2/6.

[16] grpc.io. Introduction to grpc. Available at `https://grpc.io/docs/what-is-grpc/introduction/` (2022/8/11). Last Access 2022/1/27.

[17] Holger Harms, Collin Rogowski, and Luigi Lo Iacono. Guidelines for adopting frontend architectures and patterns in microservices-based systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 902–907, 2017.

[18] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. Performance analysis of restful api and rabbitmq for microservice web application. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 257–259. IEEE, 2018.

[19] IETF. Hypertext transfer protocol – http/1.1. `https://www.ietf.org/rfc/rfc2616.txt`.

[20] Internet Engineering Task Force (IETF). The transport layer security (tls) protocol version 1.3. Available at `https://datatracker.ietf.org/doc/html/rfc8446` (). Last Access 2022/5/11.

[21] Docker Inc. Docker. Available at `https://www.docker.com`. Last Access 2022/1/10.

[22] Docker Inc. Use containers to build, share and run your applications. Available at `https://www.docker.com/resources/what-container`. Last Access 2022/1/10.

[23] intersoft consulting services AG. General data protection regulation gdpr. Available at `https://gdpr-info.eu` (). Last Access 2022/5/11.

[24] Kubernetes. What is kubernetes? Available at `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/` (2021/07/23). Last Access 2022/1/11.

[25] Li Li and Wu Chou. Design and describe rest api without violating rest: A petri net based approach. In *2011 IEEE International Conference on Web Services*, pages 508–515, 2011.

[26] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE Access*, 7:21235–21245, 2019.

[27] Mattt. Network link conditioner. Available at `https://nshipster.com/network-link-conditioner/` (2019/07/29). Last Access 2022/1/13.

[28] Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531. IEEE, 2017.

[29] Antonio Nehme, Vitor Jesus, Khaled Mahbub, and Ali Abdallah. Securing microservices. *IT Professional*, 21(1):42–49, 2019.

[30] Oracle. What is a server-side endpoint? Available at `https://docs.oracle.com/cd/E17802_01/webservices/webservices/reference/tutorials/wsit/doc/Initialization2.html`. Last Access 2022/1/10.

[31] Qlik. Meet the qlik active intelligence platform. Available at `https://www.qlik.com/us/` (). Last Access 2022/2/14.

[32] H. Ruellan R. Peon. Hpack: Header compression for http/2. Available at `https://www.hjp.at/doc/rfc/rfc7541.html` (). Last Access 2022/4/20.

[33] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Differences between model-driven development of service-oriented and microservice architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 38–45, 2017.

[34] Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33:18, 2008.

[35] Danish Sattar, Alireza Hosseini Vasoukolaei, Pat Crysdale, and Ashraf Matrawy. A stride threat model for 5g core slicing. In *2021 IEEE 4th 5G World Forum (5GWF)*, pages 247–252. IEEE, 2021.

[36] Korakit Seemakhupt and Krerk Piromsopa. When should we use http2 multiplexed stream? In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–4. IEEE, 2016.

[37] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[38] App Store. Qlik sense saas. Available at `https://apps.apple.com/us/app/qlik-sense-saas/id1523006591` (). Last Access 2022/2/14.

[39] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 352–356, 2010.

Appendices

Appendix A
Tables

A.1 Experiment 1

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	443.002	16.111	[441.179 , 444.825]	300
No BFF	1463.167	36.709	[1459.013 , 1467.321]	300
REST BFF	442.731	20.039	[440.463 , 444.999]	300

Table A.1: Latency results for alert with 1 evaluation

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	466.68	15.607	[464.914 , 468.446]	300
No BFF	1509.392	72.754	[1501.159 , 1517.625]	300
REST BFF	467.888	32.718	[464.186 , 471.59]	300

Table A.2: Latency results for alert with 4 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	484.775	20.826	[482.418 , 487.132]	300
No BFF	1537.509	31.626	[1533.93 , 1541.088]	300
REST BFF	482.925	21.636	[480.477 , 485.373]	300

Table A.3: Latency results for alert with 7 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	507.874	31.429	[504.317 , 511.431]	300
No BFF	1566.174	49.609	[1560.56 , 1571.788]	300
REST BFF	504.077	32.751	[500.371 , 507.783]	300

Table A.4: Latency results for alert with 10 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	545.932	52.838	[539.953 , 551.911]	300
No BFF	1610.774	192.896	[1588.946 , 1632.602]	300
REST BFF	529.869	43.986	[524.892 , 534.846]	300

Table A.5: Latency results for alert with 13 evaluations

A.1.1 History Latency in 3G Network

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1202.828	38.392	[1198.484 , 1207.172]	300
No BFF	4319.291	114.443	[4306.341 , 4332.241]	300
REST BFF	1151.847	47.395	[1146.484 , 1157.21]	300

Table A.6: Latency results for alert with 1 evaluation

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1253.851	35.103	[1249.879 , 1257.823]	300
No BFF	4649.397	128.276	[4634.881 , 4663.913]	300
REST BFF	1207.364	40.829	[1202.744 , 1211.984]	300

Table A.7: Latency results for alert with 4 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1304.631	37.098	[1300.433 , 1308.829]	300
No BFF	4966.262	127.305	[4951.856 , 4980.668]	300
REST BFF	1256.858	36.445	[1252.734 , 1260.982]	300

Table A.8: Latency results for alert with 7 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1359.337	33.574	[1355.538 , 1363.136]	300
No BFF	5302.108	174.608	[5282.349 , 5321.867]	300
REST BFF	1306.057	49.87	[1300.414 , 1311.7]	300

Table A.9: Latency results for alert with 10 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1409.747	39.253	[1405.305 , 1414.189]	300
No BFF	5769.792	160.431	[5751.638 , 5787.946]	300
REST BFF	1352.106	43.698	[1347.161 , 1357.051]	300

Table A.10: Latency results for alert with 13 evaluations

A.1.2 History Latency in Edge Network

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3353.031	48.3	[3347.565 , 3358.497]	300
No BFF	12686.576	230.015	[12660.547 , 12712.605]	300
REST BFF	3258.266	75.919	[3249.675 , 3266.857]	300

Table A.11: Latency results for alert with 1 evaluation

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3475.488	85.174	[3465.85 , 3485.126]	300
No BFF	13687.583	221.591	[13662.508 , 13712.658]	300
REST BFF	3386.001	139.406	[3370.226 , 3401.776]	300

Table A.12: Latency results for alert with 4 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3601.621	36.979	[3597.436 , 3605.806]	300
No BFF	14597.061	264.408	[14567.14 , 14626.982]	300
REST BFF	3495.012	102.417	[3483.422 , 3506.602]	300

Table A.13: Latency results for alert with 7 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3768.41	220.581	[3743.449 , 3793.371]	300
No BFF	15876.536	685.995	[15798.908 , 15954.164]	300
REST BFF	3638.098	109.776	[3625.676 , 3650.52]	300

Table A.14: Latency results for alert with 10 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	4148.87	171.016	[4129.518 , 4168.222]	300
No BFF	20181.909	3271.295	[19811.727 , 20552.091]	300
REST BFF	3807.324	179.263	[3787.038 , 3827.61]	300

Table A.15: Latency results for alert with 13 evaluations

A.2 Experiment 2

Type	Mean	Std	95% CI	#Iterations
GraphQL BFF	365.137	11.419	[363.845 , 366.429]	300
No BFF	366.314	10.679	[365.106 , 367.522]	300
REST BFF	361.651	13.798	[360.09 , 363.212]	300

Table A.16: Latency results for user in a normal network

Type	Mean	Std	95% CI	#Iterations
GraphQL BFF	1086.986	18.183	[1084.928 , 1089.044]	300
No BFF	1069.618	31.027	[1066.107 , 1073.129]	300
REST BFF	1063.208	25.487	[1060.324 , 1066.092]	300

Table A.17: Latency results for user in a 3G network

Type	Mean	Std	95% CI	#Iterations
GraphQL BFF	3190.003	34.673	[3186.079 , 3193.927]	300
No BFF	3144.898	89.7	[3134.747 , 3155.049]	300
REST BFF	3128.694	66.448	[3121.175 , 3136.213]	300

Table A.18: Latency results for user in an edge network

A.2.1 Experiment 3

Script	#Evaluations	Mean(Size)	#Iterations
No BFF	1	34341.183	300
REST BFF	1	7917.286	300
GraphQL BFF	1	10896.263	300
No BFF	4	116168.753	300
REST BFF	4	11221.313	300
GraphQL BFF	4	14405.83	300
No BFF	7	,197346.643	300
REST BFF	7	14691.086	300
GraphQL BFF	7	17866.44	300
No BFF	10	279253.006	300
REST BFF	10	17382.13	300
GraphQL BFF	10	21295.04	300
No BFF	13	360091.3	300
REST BFF	13	20543.92	300
GraphQL BFF	13	24826.913	300

Table A.19: Amount of uploaded and downloaded data by the client fetching histories (Bytes)

Script	Mean(Size)	#Iterations
No BFF	7581.596	300
REST BFF	7614.98	300
GraphQL BFF	9114.26	300

Table A.20: Amount of data downloaded and uploaded to fetch user information (Bytes)

A.3 Experiment 4

A.3.1 Alert Normal Network

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1558.621	114.921	[1545.616 , 1571.626]	300
Grpc BFF	1542.89	115.138	[1529.861 , 1555.919]	300
REST BFF	1546.328	116.996	[1533.089 , 1559.567]	300

Table A.21: Latency results for alert with 1 evaluation

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1589.592	132.07	[1574.647 , 1604.537]	300
Grpc BFF	1576.938	126.039	[1562.675 , 1591.201]	300
REST BFF	1582.886	125.526	[1568.681 , 1597.091]	300

Table A.22: Latency results for alert with 4 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1619.54	120.368	[1605.919 , 1633.161]	300
Grpc BFF	1614.886	137.96	[1599.274 , 1630.498]	300
REST BFF	1632.144	151.323	[1615.02 , 1649.268]	300

Table A.23: Latency results for alert with 7 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1660.537	142.0	[1644.468 , 1676.606]	300
Grpc BFF	1646.015	134.58	[1630.786 , 1661.244]	300
REST BFF	1663.583	155.532	[1645.983 , 1681.183]	300

Table A.24: Latency results for alert with 10 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	1661.119	155.406	[1643.533 , 1678.705]	300
Grpc BFF	1661.249	154.428	[1643.774 , 1678.724]	300
REST BFF	1665.079	130.652	[1650.294 , 1679.864]	300

Table A.25: Latency results for alert with 13 evaluations

A.3.2 Alert 3G Network

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	2061.459	119.981	[2047.882 , 2075.036]	300
Grpc BFF	1769.485	125.314	[1755.304 , 1783.666]	300
REST BFF	2017.906	126.802	[2003.557 , 2032.255]	300

Table A.26: Latency results for alert with 1 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	2143.421	134.271	[2128.227 , 2158.615]	300
Grpc BFF	1825.534	123.622	[1811.545 , 1839.523]	300
REST BFF	2123.889	142.106	[2107.808 , 2139.97]	300

Table A.27: Latency results for alert with 4 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	2204.064	177.738	[2183.951 , 2224.177]	300
Grpc BFF	1887.707	132.547	[1872.708 , 1902.706]	300
REST BFF	2157.753	168.03	[2138.739 , 2176.767]	300

Table A.28: Latency results for alert with 7 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	2242.946	209.323	[2219.259 , 2266.633]	300
Grpc BFF	1899.231	149.507	[1882.313 , 1916.149]	300
REST BFF	2199.991	125.657	[2185.772 , 2214.21]	300

Table A.29: Latency results for alert with 10 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	2505.991	186.47	[2484.89 , 2527.092]	300
Grpc BFF	1952.467	142.69	[1936.32 , 1968.614]	300
REST BFF	2408.924	165.655	[2390.178 , 2427.67]	300

Table A.30: Latency results for alert with 13 evaluations

A.3.3 Alert Edge Network

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3455.312	137.904	[3439.707 , 3470.917]	300
Grpc BFF	2454.965	155.562	[2437.362 , 2472.568]	300
REST BFF	3404.498	145.786	[3388.001 , 3420.995]	300

Table A.31: Latency results for alert with 1 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3596.961	148.939	[3580.107 , 3613.815]	300
Grpc BFF	2576.41	164.306	[2557.817 , 2595.003]	300
REST BFF	3553.451	140.732	[3537.526 , 3569.376]	300

Table A.32: Latency results for alert with 4 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3696.988	139.501	[3681.202 , 3712.774]	300
Grpc BFF	2665.46	147.276	[2648.794 , 2682.126]	300
REST BFF	3633.697	156.163	[3616.025 , 3651.369]	300

Table A.33: Latency results for alert with 7 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	3861.509	156.405	[3843.81 , 3879.208]	300
Grpc BFF	2759.331	210.841	[2735.472 , 2783.19]	300
REST BFF	3777.865	163.546	[3759.358 , 3796.372]	300

Table A.34: Latency results for alert with 10 evaluations

Script	Mean	Std	95% CI	#Iterations
GraphQL BFF	4707.094	227.652	[4681.333 , 4732.855]	300
Grpc BFF	2834.404	233.643	[2807.965 , 2860.843]	300
REST BFF	4505.178	181.524	[4484.637 , 4525.719]	300

Table A.35: Latency results for alert with 13 evaluations

A.3.4 User Normal Network

Type	Mean	Std	95% CI	#Iterations
GraphQL BFF	416.379	52.558	[410.432 , 422.326]	300
Grpc BFF	416.864	50.123	[411.192 , 422.536]	300
REST BFF	412.543	50.686	[406.807 , 418.279]	300

Table A.36: Latency results for User in the Normal Network

A.3.5 User 3G Network

Type	Mean	Std	95% CI	#Iterations
GraphQL BFF	927.096	77.613	[918.313 , 935.879]	300
Grpc BFF	641.675	64.884	[634.333 , 649.017]	300
REST BFF	894.685	86.12	[884.94 , 904.43]	300

Table A.37: Latency results for User in a 3G Network

A.3.6 User Edge Network

Type	Mean	Std	95% CI	#Iterations
GraphQL BFF	2261.67	74.331	[2253.259 , 2270.081]	300
Grpc BFF	1343.304	83.546	[1333.85 , 1352.758]	300
REST BFF	2280.573	229.419	[2254.612 , 2306.534]	300

Table A.38: Latency results for User in an Edge Network

A.3.7 User Data Usage

Script	Mean(Size)	#Iterations
GraphQL BFF	2965.52	300
gRPC BFF	1432.296	300
REST BFF	2686.82	300

Table A.39: Amount of data downloaded and uploaded to fetch user information (Bytes)

A.3.8 History Data Usage

Script	#Evaluations	Mean(Size)	#Iterations
gRPC BFF	1	1680.263	300
REST BFF	1	2904.06	300
GraphQL BFF	1	4636.11	300
gRPC BFF	4	3526.05	300
REST BFF	4	6024.24	300
GraphQL BFF	4	8243.18	300
gRPC BFF	7	,5262.2033	300
REST BFF	7	9111.32	300
GraphQL BFF	7	11729.3	300
gRPC BFF	10	7006.103	300
REST BFF	10	12191.52	300
GraphQL BFF	10	15119.18	300
gRPC BFF	13	8636.126	300
REST BFF	13	15198.433	300
GraphQL BFF	13	18668.946	300

Table A.40: Amount of uploaded and downloaded data by the client fetching histories (Bytes)

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-08

EXAMENSARBETE The Evaluation of Using Backend-For-Frontend (BFF) in a Microservices Environment
STUDENT Samer Alkhodary
HANDLEDARE Alfred Åkesson (LTH), Paul Ericsson (Qlik), Johan Enell (Qlik)
EXAMINATOR Niklas Fors (LTH)

Everyone Needs a BFF, Even our Mobile Applications

POPULÄRVETENSKAPLIG SAMMANFATTNING **Samer Alkhodary**

Mobile and web applications have become significant parts of people's lives nowadays, especially with the boom of smartphones and tablets. With the competitive market of mobile applications, software and service providers need to build more reliable and faster applications to compete for users' attention. This thesis shows that the Backend-for-Frontend design pattern (BFF) improves the communication between the clients and backend services and reduces the coding complexity of the clients' codebase.

The majority of us use smartphones for almost everything, for example, to get the latest news and discounts and stay connected to friends and loved ones. However, we hate seeing the never-ending loading screens that feel like an eternity. Also, we get irritated when we see that a specific mobile application uses tons of mobile data, which costs us more money every month. In this thesis, we tested a new way for mobile apps to communicate with the cloud and get us the information we want to see. The experiments showed that the new approach, which is called Backend-For-Frontend, a.k.a BFF decreased the time an application needs to get information up to 4 times. It also reduced the data usage of that application by a whopping 17 times. This means that applications using the new solution are much faster. As a result, they save us, the users, more mobile data, reducing our mobile bills. The BFF solution helps the programmers spend less time fixing bugs which gives them more time to develop new exciting features that can improve the user experience. Moreover, we specified some essential cyber security recommendations to help programmers keep the new approach safe. This way, we can use our favorite apps safely, knowing that our data and private information are secure and tucked away from prying eyes. We conducted the experiments with the new cloud approach using Qlik's mobile application. We tested the impact of the BFF on the time the application needs to get the information it needs from the cloud and the amount of mobile data it uses while doing so. Moreover, we tested the effect of the BFF design on the coding process of the application. We also tried implementing the BFF design with different communication technologies. We found that the different technologies had various advantages that programmers can utilize to tweak their applications to give the users the best possible performance.