Examensarbete

HÖGSKOLAN
I SKÖVDE

REST AND GRAPHQL: A COMPARATIVE ANALYSIS IN MOBILE NETWORK ENVIRONMENTS

REST OCH GRAPHQL: EN JÄMFÖRELSE ANALYS I MOBILA NÄTVERKS MILJÖER

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå nivå 30 Högskolepoäng
Vårtermin 2024

Johan Persson

Handledare: Manfred Jeusfeld
Examinator: Joeri van Laere

Summary

This study investigates the performance differences between REST and GraphQL, focusing on their efficiency in data fetching within different network environments, specifically in terms of WiFi and 4G mobile networks. As microservices and API architectures have evolved, REST has become a standard. However, GraphQL is a promising and interesting alternative, addressing REST issues such as over-fetching.

This study aims to answer two primary research questions: 1) How does GraphQL perform in relation to REST with regard to response times, throughput, and payload size in both regular WiFi and 4G settings? And 2) in what scenarios is GraphQL most efficient?

To answer these questions, a series of experiments was conducted using a locally developed GraphQL server and a REST API, both benchmarked under various user load scenarios using Apache JMeter (for the rest of the paper to be referred to as “JMeter”).

The findings indicate that GraphQL demonstrates an ability to reduce payload sizes and decrease response times and throughput, particularly in mobile network environments. It is also shown that it matters how GraphQL is implemented in terms of what type of source the data is fetched from, demonstrating significant differences between services such as Azure Cosmos DB and Elasticsearch.

Keywords: GraphQL, REST, microservices, API, data fetching, payload optimization

Table of contents

Summary 2
1 Introduction 1
1.1 Microservices 1
1.2 REST, Endpoints, HTTP & CRUD 1
1.3 Drawbacks of REST 2
1.4 GraphQL 2
1.5 Expected Results & Research Questions 3
2 Background 4
2.1 A Brief Background on Fetching Data 4
2.2 Previous Studies 4
2.3 REST 6
2.3.1 Adoption and brief history 6
2.3.2 Challenges 6
2.3.3 Over-fetching 6
2.3.4 Under-fetching 7
2.4 GraphQL 7
2.4.1 Challenges 9
2.5 The Future of APIs 9
3 Problem Areas 11
3.1 Existing Research and Gaps 11
3.2 Problem/Question 11
3.3 Network 11
3.3.1 Research Question 1 12
3.3.2 Research Question 2 12
3.4 Benchmarking APIs 12
3.5 How Will the Research Method Impact the Results? 12
3.5.1 Experiment 12
3.5.2 JMeter 13
3.5.4 The data 13
3.6 Delimitations 14
3.6.1 JMeter 14
3.6.2 Data fetching focus 14
3.6.3 Mutations / POST / UPDATE 14
3.6.4 Subscriptions 14
3.7 Expected Result 15
3.7.1 Ethical and societal connections 15
4 Method 16
4.1 Scientific Approach 16
4.1.1 Experiment 16
4.1.2 Hypothesis 16
4.1.3 Method RQ1 17
4.1.4 Method RQ2 18

4.2 Dependent Variables and Independent Variables 20
4.3 Ethical Considerations 21
5 Material Presentation 22
5.1 Response Times 22
5.1.1 GraphQL 22
5.1.2 GraphQL - Elasticsearch 23
5.1.3 GraphQL - Cosmos 24
5.1.4 REST 25
5.2 Over-Fetching 26
5.2.1 GraphQL 26
5.2.2 GraphQL - Elasticsearch 27
5.2.3 GraphQL - Cosmos DB 28
5.3 Throughput With Comparable Payload Size 29
5.3.1 GraphQL - REST 29
5.3.2 GraphQL - Elasticsearch 30
5.3.3 GraphQL - Cosmos DB 31
5.3.4 REST 32
5.4 Throughput Over-Fetching 33
5.4.1 GraphQL - REST 33
5.4.2 GraphQL - Elasticsearch 34
5.4.3 GraphQL - Cosmos DB 35
5.5 Payload 36
6 Analysis 37
6.1 500 Users 38
6.1.1 Response times -same payload size 38
6.1.2 Response times - over-fetching 39
6.1.3 Throughput 40
Test execution time 41
6.2 300 Users 42
6.2.1 Response times - same payload size 42
6.2.2 Response times - over-fetching 43
6.2.3 Throughput 44
Test execution time 45
6.3 100 Users 46
6.3.1 Response times - same payload size 46
6.3.2 Response times - over-fetching 47
6.3.3 Throughput 48
Test execution time 49
6.4 A Closer Look At Network, Over-Fetching and Response Times 50
7 Results 51
7.1 Research Question 1 51
7.1.1 Networks & Insights 51
7.1.2 Conclusion: Research Question 1 52
7.2 Research Question 2 52

7.2.1 GraphQL-REST **52**
7.2.2 GraphQL-Elastic **54**
7.2.3 GraphQL-Cosmos **55**
7.2.4 REST **55**
7.2.5 Conclusion: Research Question 2 **56**
8 Discussion **57**
8.1 Societal Aspects **57**
8.2 Ethical Aspects **58**
8.3 Environmental Aspects **58**
8.3.1 Threats to validity **58**
8.3.2 Reliability **59**
8.3.3 Future studies **59**
9 References **61**

1 Introduction

For a long time, software was designed, created and maintained within monolithic architectures, meaning that it was often large and complex and also somewhat chaotic when it came to things such as updates and patchwork (Adrio et al. 2023). Many of the parts doing one certain thing were dependent on other parts of the system running their course without issue, and if one segment failed, the whole system would go down. With many vital parts operating side by side and intertwined in a monolithic system, this often led to substantial down time and complex troubleshooting with unwanted costs as a byproduct. One way to combat these vulnerabilities was to break the monolith apart into clearly separated services which ran independent of one another so as not to affect each other completely should one system fail (Autili, Filippone and Massimo Tivoli, 2023).

1.1 Microservices

Microservices are a style of microservice architecture whereby a set of vital parts are identified and taken apart from each other to run separately (Mosleh, Dalili, and Heydari 2018). One of the most important components of microservices are their loose couplings to one another. These loose couplings are often implemented via REST (Representational State Transfer) APIs, which enable services to communicate with each other through data transfer (Rodriguez, 2008). This hinders one service from crashing if another goes down due to error. The architecture might still not function properly if it is dependent on data from the crashed service, however instead of searching for bugs in a whole monolithic system, there is now a narrower set of code to investigate due to a more apparent indication regarding which service has crashed.

1.2 REST, Endpoints, HTTP & CRUD

REST is used to expose data of certain types either publicly, over the Internet through endpoints, or privately within organizational domains. It is a very common way to have clients and servers communicate in a stateless manner. One of REST's strongest capabilities is that it doesn't care what type of service, application or system it is being called by. It easily transfers data from one application to another, regardless of the code language.

One of the two most valuable strengths of REST is its response formats. Most commonly used is JSON, but there are other formats available as well – for example, XML which, together with JSON, can be handled by all modern paradigms within development today. REST's other strength is its use of the HTTP protocol which empowers REST with exposure to the network either publicly or locally. REST requires HTTP 1.1 to perform CRUD operations, which means that almost all devices running a client that can access the 1.1 protocol can also access and use the power of REST technology (Vadlamani et al. 2021). CRUD is an acronym which stands for Create, Read, Update and Delete. These are the basic operations that any modern data fetching technology needs to be capable of

performing to fulfill rudimentary requirements of developers and organizations. These operations are what are used to retrieve and write data to and from a range of sources. Within the HTTP protocol, these operations are called GET, POST, UPDATE, DELETE. Regardless of their naming, they remain the same basic needed functionalities for successful data source manipulation and handling.

1.3 Drawbacks of REST

There are also drawbacks to REST, one of them being over-fetching (Brito and Valente, 2020). This is a phenomenon that the vast majority of developers working with REST APIs face everyday. When data is fetched, many times only a portion of it is needed. However, REST will deliver all of the data connected to the requested endpoint. This means that, often, bigger payloads than necessary are being sent over the network.

On the other hand, another drawback is under-fetching. Under-fetching is common in microservice environments where information is needed from different services via, for example, endpoints made available by REST. This cannot be done through the same endpoint; instead, several calls need to be made to separate services to collect data, followed by a certain amount of manipulation to join, wash and prepare the data for usage.

1.4 GraphQL

GraphQL is a query language created by Facebook in 2012. Back then, the organization was facing issues with delivering huge amounts of data in an efficient way for all the concurrent mobile users requesting it (Brito and Valente, 2020). To act on these issues, Facebook needed to slim down the data responses from their APIs to create faster applications (Vadlamani et al. 2021). GraphQL was designed to do just that. One of its main ideas is to allow the client to request only the data that is needed. Some studies show that GraphQL decreases payload sizes by up to 99% (Vadlamani et al. 2021). GraphQL can combine several sources and deliver them in a single request from a single endpoint. In contrast to REST's multiple endpoints, this also potentially makes development more streamlined.

GraphQL is also a strongly-typed language, meaning that each data transaction is matched against a set of predetermined types. These types can be, for example, strings (text), Ints (integers) and other common basic programming types such as lists and objects. This aids in making sure that the implementation is robust and strict in terms of what type of data is allowed to be processed by the server.

1.5 Expected Results & Research Questions

This paper aims to answer two questions related to the comparison of the two technologies REST and GraphQL. Firstly, how does GraphQL perform in the mobile network? Former studies have had results that vary in their outcome. Both paradigms have delivered similar results in regard to response times like in the study performed by Vadlamani et al. (2021), where REST performed better than GraphQL in one occurrence. The first question therefore is:

How does GraphQL perform in relation to REST in both a regular (WiFi) setting as well as a mobile setting (4G)?

To answer this question, tests running a locally-developed GraphQL server will be conducted alongside a locally-hosted implementation of a REST API to observe the changes in dependent variables such as throughput (efficiency), payload (byte size of response) and response time (time in milliseconds until response was received from server). Due to the nature of hardware and its widely-varying performance capacities, it is important that all involved servers are hosted on the same machine (or hardware) to eliminate the risk of faulty or irrelevant results. As such, it is crucial to eliminate independent variables that can bias the results in order to claim that the one independent variable being investigated (in this case, the network type) is the variable that makes the difference in what is being measured (Oates, Griffiths and McLean, 2022).

Expectations are that the results will show an increase in productivity in favor of GraphQL when the issue of over-fetching is being highlighted, due to its reduction in payload sizes. As REST endpoints keep delivering the same data size, GraphQL queries will request only a portion of that data, thus resulting in less usage of the network. In combination with a higher number of users, this study expects a higher efficiency for GraphQL queries which fetch precise data.

The second research question is:

In what setting is GraphQL most efficient?

Since the performance results of these two paradigms fluctuate in favor of both directions, it begs the question: is there a certain type of server implementation where GraphQL can outperform REST? To answer this question, performance measurements will be done with 3 different GraphQL servers implemented to fetch data from three different scenarios: 1) Against Elasticsearch, a document-based search engine; 2) Against Cosmos DB in Microsoft Azure's cloud environment; and 3) Against a layer of REST endpoint. By looking at the performance of GraphQL versus REST in both different network environments as well as architectural ones, future decision makers and data-driven organizations can gain a deeper understanding of when to choose which technology for their business objectives.

2 Background

2.1 A Brief Background on Fetching Data

Ever since the rise of complex computing in the 1960s and 1970s, the need for interconnectivity has increased rapidly. Due to this demand and the exponential growth of new users and units on the Internet, the methods of data fetching have evolved to meet these needs. In the 1940-1950s, one of the ways data got fetched was by direct access to the machine and its disks via command line tools. These were cumbersome methods that required a deep understanding of operating systems and their structure (Goda and Kitsuregawa, 2012). In the early 1970s, SQL (Structure Query Language) and relational databases were introduced as a way to query a datasource based on uniquely identifying keys. As data structures became more complex, SQL provided a flexible way to query that structure and combine data into new sets of data.

SOAP (Simple Object Access Protocol) came around in the early 1990s as a way to connect devices via web services, and could be implemented in both HTTP and SMTP protocols (Wennerström, Jespersen and Lundquist, 2002). Its result format was XML, which later also turned out to be one of its drawbacks, as XML can result in complex structures and big payloads. As the web and its utilization became more complex and faced more and more users, data fetching called for even more flexibility (Barbaglia, Murzilli and Cudini, 2016) and in the year 2000, Roy Fielding wrote his PhD dissertation defining REST that would quickly grow to become the web's most adopted API. REST is widely adopted and is an efficient way to handle CRUD operations on data. Many companies are using it whilst also looking at new technologies and the potential benefits of GraphQL as an approach to data fetching.

2.2 Previous Studies

Many studies performed in the area used test cases, mock data and students to answer their questions regarding the rise and adoption of GraphQL and its relation to REST. It also seems evident that the outcomes and conclusions beg for more research and data on the matter. For example, Vadlamani et al. (2021) based their study on single requests instead of a heavier load to observe performance as pressure on the system increased, and in Vohra and Manuaba (2022), a single type of data source was used, and there was no transparency about network type.

Therefore, this study aims to set up a performance-test scenario based on a real-world microservice e-commerce website creating a GraphQL implementation to be tested and measured alongside the current REST implementation. The goal is to answer questions regarding GraphQL and REST in different network environments as payload decreases and user load increases, and to look further into the findings of Marchuk, Dyyak and Makar (2023) where the authors concluded that different execution strategies potentially have a heavy influence on GraphQL's efficiency.

2.3 REST

2.3.1 Adoption and brief history

In the year 2000, Roy Thomas Fielding wrote his doctoral dissertation about the architecture of the WEB as it were and how it could develop in the future. In it, he described the REST protocol, which he defined as a method for coupling clients and servers using the HTTP protocol (Barbaglia, Murzilli and Cudini, 2016). This, to create statelessness between the client and the server. “Statelessness” refers to how a server does not need to keep any sessionstorage or information about previous client requests and connections, and therefore becomes less complex to design and implement (Vadlamani et al. 2021). To make this possible, the client needs to authorize itself for each request made, thus requiring extra data in its API calls such as tokens or keys to confirm identity.

Since Fielding's dissertation in 2000, REST has become widely adopted with upwards of 86% of WEB API developers using REST Architecture in their everyday work, making it the most adopted application programming interface to date, by far. It is notable, however, that REST’s domination in adoption has decreased by 6% in just a handful of years, from 92% to 86% percent (Xu, 2023), making room for up-and-comers such as GraphQL which have seen a slight increase in usage during the same time period.

2.3.2 Challenges

Two of the most prominent issues with REST are the so-called over-fetching and under-fetching phenomenons. Over-fetching is when an endpoint returns more data than the client needs for a given request (Vadlamani et al. 2021).

Let's imagine a microservice environment that has a specific endpoint meant to expose customer data for other internal services to consume, and investigate the impact of over-fetching below.

2.3.3 Over-fetching

Suppose a developer needs to fetch data from a theoretical specific endpoint. To solve their problem at hand, data regarding a customer's name, address and age is needed from the customer endpoint. However, the endpoint could also contain many other data fields. Marital status, SSN, member numbers and time of registrations are examples of other informative data fields held by the customer endpoint, they all have certain purposes and are important in their own right. For the current task at hand, however, let’s suppose the developer only needs name, address and age.

Employing REST, the endpoint will still deliver all those other data fields in the payload, creating an unnecessary amount of data being transferred over the network, and possibly also unnecessary exposure of data. This is why it is called over-fetching, because it delivers more than it has to, or should.

2.3.4 Under-fetching

Under-fetching, on the other hand, is when a specific endpoint does not deliver enough data (Vadlamani et al. 2021).

Let's expand upon the imaginary microservice to also include order history. This information is being exposed by a different endpoint called ‘orders’. The two separate endpoints can deliver information about a customer and its previous orders, but to retrieve the information it needs to request two separate endpoints, it would require repetitive round-trip requests fetching data from multiple sources to satisfy the complete request requirements. This scenario is an example of under-fetching, it basically means that the client needs to make round trip, repetitive requests to fulfill the query, and each request has the potential and likelihood to deliver over-fetched results. This creates somewhat of a bloated scenario where lots of bytes of data are being transported over the network, but potentially only a fraction of them are getting used to fulfill a purpose.

2.4 GraphQL

GraphQL was created in 2012 as an internal experimental tool meant to address problems with massive amounts of concurrent users fetching data from multiple sources at the same time (Brito and Valente, 2020). Mainly on mobile devices, Facebook saw problems with delivering a seamless and responsive experience to their users, forcing them to rethink the way they fetch data and look for possible ways to mitigate their problems outside of REST and other standard technologies.

A few years later, in late 2015, GraphQL was open-sourced and made available outside of the organization, allowing it to be investigated and adopted by other organizations on the market. Today, significant platforms such as AirBnB and Github are running concurrent REST and GraphQL implementations (Brito and Valente, 2020).

A couple of GraphQL’s main strengths are that it helps the client and developers to fetch only the data that they need and ask for and nothing more, thus potentially reducing the payload substantially. Some studies have shown that GraphQL can decrease the payload by as much as 99% for certain types of queries, an example of this is in such a case where perhaps the only data needed from an endpoint is just a few fields out of several hundred.

Another strength is that a GraphQL implementation will expose only one endpoint for all queries defined. This means that in comparison to REST, which can expose an arbitrary amount of endpoints, GraphQL can handle all types of queries and sources through one single endpoint (Brito and Valente, 2020). In a GraphQL implementation, queries are defined and strongly typed through interfaces.

2.4.1 Challenges

According to previous studies, GraphQL can be hard to conceive (Vadlamani et al. 2021). It can take a long time to learn and grasp the concept well enough to use it correctly. It demands a more complex implementation and careful definition of queries, mutations and subscriptions which are the equivalents of REST's CRUD operations (GET, POST, PUT and DELETE). A GraphQL server would most likely need more time to develop in comparison to a traditional REST implementation. As it becomes more complex with many specific queries, the GraphQL server demands tedious structure so that developers can maintain and scale the server as client applications increase their request volumes. On the other hand, when this is done, the developer might find themselves doing less data processing in relation to fetching, as data is being delivered in a more readily available manner.

2.5 The Future of APIs

To understand what the future for APIs is, this study takes a brief look at what the developer's views and opinions are regarding the shortcomings and strengths of a good API. Developers can face many challenges as they learn new technologies, which can take time away from producing value for the organization. One of the key findings of Nawaz et al. (2022) indicates that if a technology is complex to learn, it might fall short due to the need and demand for fast solutions to the projects and requirements handed to employees by the business side of their organizations. Vadlamani et al. (2021) have found that since GraphQL is a fairly young technology, it might cause issues that are hard to resolve due to a lack of resources on problem solving. This, in contrast to a technology like REST which has been around for over 20 years and has countless sources of research and forums to address surfacing challenges.

Though these concerns are very real, there is also another side to the coin. Brito and Valente (2020) found that both students and developers find GraphQL easier to construct queries against once the server was set up compared to REST. This indicates that if a GraphQL server has been properly configured to handle the needs of an organization, the lead time for development might go down, and the learning curve for new frontend developers could flatten out to a certain degree. GraphQL is developer-friendly in the query construction aspect due to its JSON-mimicking structure, which most developers are familiar with today. APIs that are reliable, flexible and easy to understand and work with will continue to be used, and the ones unable to combat their weaknesses will eventually fall to the side. GraphQL shows promise in its query construction and its ability to slim down payload sizes. REST is a robust technology with a huge amount of learning sources and guides, and will stay at the top of the adoption list for many years to come. What is clear is that developers want to work with technologies that are easy to get into, require less boilerplate (extra code required to set up the environment) and that deliver data at fast response times with reliable results. If these requirements are met, then the technology will remain in the arena of APIs to be used by developers and organizations.

3 Problem Areas

3.1 Existing Research and Gaps

In studies performed on the comparison between the two data fetching paradigms, relevant real-world scenarios and data are lacking. Existing research such as in Vohra and Kerthyayana Manuaba (2022), a study about the “Implementation of REST API vs GraphQL in Microservice Architecture'', investigated test cases by implementing REST and GraphQL scenarios in a local environment using only a single RDBMS (relational database system). As the web becomes increasingly mobile-driven, and despite GraphQL being *specifically* designed to solve Facebook's mobile issues, the study authors did not seem to test GraphQL in a mobile setting.

Furthermore, other studies such as Hartina, Lawi, and Panggabean (2018) investigated the “Performance Analysis of GraphQL and RESTful API” in a single relational database management system (RDBMS), which does not reflect a real-world microservice implementation were often many more sources are used to store relevant data. Even further, in another study performed by Vadlamani et al. (2021), the performance testing was built on a single request query, also not reflecting a real-world scenario where REST’s under- and over-fetching issues were properly exposed alongside GraphQLs more complicated layers of abstractions. To properly measure performance differences between GraphQL and REST, benchmarking needs to be set up in a scenario that effectively mirrors a use case of an operational microservice environment.

3.2 Problem/Question

This study shall conduct a performance experiment through a setup based on a real-world use case utilizing Cosmos DB (Azure) and Elasticsearch as well as REST (swagger). Elasticsearch is a data-driven search engine database that is based on indexes and documents; Cosmos DB is a service that can run different types of data models, such as SQL, NoSQL and graph databases among others. This research uses the structure and data from an e-commerce web service handling thousands of concurrent users browsing the site, many of them in a mobile network environment fetching data about configurable products, which means a lot of fetching as the customer changes the product look. By setting up a GraphQL server in parallel to a REST implementation, the aim is to conduct an experiment to measure the performance differences between the two approaches to data fetching.

3.3 Network

Current studies lack the investigation needed in a mobile network environment. This study will measure performance and efficiency in a mobile setting and on a regular WiFi network. The mobile tests will be run via a mobile device as a network provider, inheriting all the properties of the mobile network. The regular WiFi (2.4GHz) network

will be used to run tests to show potential variations in the results between the two network types.

3.3.1 Research Question 1

How does GraphQL perform in terms of response times, throughput and payload size in relation to REST in mobile and desktop settings? Are there any notable differences as payload size reduces?

3.3.2 Research Question 2

In what setting is GraphQL most effective? In what type of environment do we see the best performance from a GraphQL server fetching data from services like Elasticsearch, Cosmos DB and REST?

3.4 Benchmarking APIs

The definition of benchmarking according to the Oxford Dictionary is to "Evaluate (something) by comparison with a standard". The standard can be an internal or external criteria. When benchmarking APIs, parameters such as throughput (transactions per second), payload (the size of the response in bytes) and response times (time it takes for an API to respond with the requested data) (Rui André Oliveira et al., 2020) are key elements of what constitutes the results to be analyzed. Benchmarking APIs is not just about ensuring that the API can handle expected load – which is the amount of concurrent users at any given time – but also to see how the API behaves during heavy load in general, to identify bottlenecks and latency issues so that proper measures can be taken to optimize the current technology, and to choose the best fit for the architecture and organizational goals.

3.5 How Will the Research Method Impact the Results?

3.5.1 Experiment

By conducting an experiment to measure performance and compare between several different servers, it is important to achieve a high internal and external validity (Oates, Griffiths and McLean, 2022). To achieve high internal validity means that one can say with certainty that it is a single independent variable change that is causing the effect on what is being measured. A researcher needs to eliminate as many independent variables as possible, and to succeed with this, this study will run all servers on the same machine. This way, the independent variable of different hardware capacity is eliminated. There are, however, two different server programming languages at play; C# for the REST server, and TypeScript for the GraphQL servers.

To reach high external validity, researchers should see to it that the results are generalizable (Oates, Griffiths and McLean, 2022). To ensure this, this study has run a

vast amount of tests to see that the results are reproducible and not unpredictably fluctuating from test run to test run. In addition, another approach to reach higher validity is to use representative test cases, which means running the tests in an environment that is representative to real-world use cases. For example, the REST implementation is based on C# and Dotnet, and is in use at the time of writing this paper, in a microservice system.

The TypeScript Apollo Server is also a well-adopted environment, and a recommendation from the GraphQL foundation. These are both well-adopted setups being utilized in real-world use cases, and should not provide any harm to the results in terms of external validity.

3.5.2 JMeter

Using JMeter to answer Research Question 1 has many benefits. It is a powerful tool when it comes to benchmarking. It does not have the capacity to emulate a mobile device's hardware, but can be configured to simulate network capacities and request headers as well as concurrent users, making it a great tool to measure different types of environments in fetching data. At the same time, it must be emphasized that the test results in this study are not based on an actual physical mobile environment running specific operating systems and hardware setups.

3.5.3 Local execution

To make accurate measurements in an experiment, it is important to decrease the amount of independent variables. One of those independent variables is the hardware on which the software is being run during tests and how powerful that hardware is. To take control of this, both the GraphQL and the REST servers will be run locally on a ThinkPad T14s with 32gb RAM and an Intel EVO i7 processor. To run applications locally means to eliminate the need for internet access to communicate between the client and server. All applications needed are hosted on, in this case, the same unit.

3.5.4 The data

This research is not based on test data but rather real in-use data being utilized by a company selling products through a complex web application. Its structure is complex due to the company's configurable products making many combinations possible, resulting in an exponential number of variants of a single base product. This concept is a great way to get more realistic test results compared to earlier studies, since it requires multiple sources and in terms of REST, multiple round trip requests, which are one of the main weaknesses in REST that this study is attempting to measure GraphQL performance against. The data in this study is based on this complex structure.

3.6 Delimitations

The GraphQL server will be set up using VisualCode, running TypeScript and Apollo server which is a well documented and used GraphQL gateway recommended by

GraphQL.org and the GraphQL foundation. The environment will be local due to the need for control of hardware knowledge. These tests can therefore not be run at scale and heavy load due to the limitations of the capacity of a single source node server running both test client and server implementations.

3.6.1 JMeter

JMeter can be configured as a proxy for a client, meaning that it is possible to run a complete environment, measuring very accurately all the request behavior of a client to server relationship while avoiding limitations that come with implementing a few queries directly into the jmeter interface. However, this is outside of the scope of this study, as the time and resources within this thesis are not enough to implement a complete system benchmarking setup. This is something that should be tested further in future studies.

3.6.2 Data fetching focus

This study will focus on the concept of fetching data, and will therefore not be measuring performance for writing data. In REST, one can typically use the POST method to send data in an object body over a network to a service which then writes the data to the source and or performs other operations. With GraphQL, the same purpose is filled by something called a mutation.

3.6.3 Mutations / POST / UPDATE

A mutation is the equivalent of writing data. A mutation works as queries do, but the other way around where it is possible to write over specific fields without concern for the rest of the data connected to the endpoint or source object. This has the potential to, in some cases, decrease the amount of data being sent in objects over the network in POST requests. This is, however, outside of the scope of this paper due to constraints in time and resources.

3.6.4 Subscriptions

Lastly, this study will not look at subscriptions, which are a third query type available in GraphQL. Subscriptions are like event listeners or triggers in a relational database that react when certain terms are met. This enables the possibility for live data and dynamic reactive applications.- However, this study will focus on reading data and, as such, mutations and subscriptions are therefore out of scope of this study.

3.7 Expected Result

The tests in this study are designed to compare performance between REST and GraphQL in different types of network environments. The performance is expected to be very even in its distribution between the two paradigms, and there should be specific settings where GraphQL can outperform REST consistently.

It is expected that GraphQL will outperform REST in a mobile setting in terms of efficiency as payload size decreases for the GraphQL queries. It will be the details that separate the two technologies, meaning that the choice between GraphQL and REST is very much an optimization question. That is the very reason this study also looks at three different possible implementations of GraphQL to see if there is a certain setting where REST will be clearly outperformed. Today, many organizations using GraphQL also run parallel REST implementations, and this might just be the way to go when adopting the new technology.

This study aims to broaden the knowledge of how and when to implement GraphQL, if at all. By utilizing a more complex data structure and adding the dimension of mobile networks, a greater understanding of the two technologies' drawbacks and powers can hopefully be reached.

By running several different tests for each one of the four implementations on both WiFi and 4G networks, and then reducing the payload while highlighting the over-fetching problem, expectations are that GraphQL queries will have a higher efficiency increase as the payload sizes decrease. This would suggest that GraphQL potentially can outperform REST in certain user-heavy scenarios by utilizing its capability to pick and choose certain data to respond to the client with.

3.7.1 Ethical and societal connections

Within the results of the tests in this study, performance data is generated and directly linked to power and energy. To be able to perform at a high level, lots of power is needed. In the age of AI and massive data centers, we have to ask ourselves what the performance is worth. Though not in the scope of this research, processors need a lot of energy to process queries and other operations performed within computers. Therefore, it is important that choices made within optimization are grounded in reason, meaning that decisions should be made by including factors as environmental effects.

4 Method

In the data subjected to testing, no personal information will be used whatsoever. Only raw data regarding products and product categories will be used in the queries created to run the tests.

4.1 Scientific Approach

4.1.1 Experiment

To conduct the research that is suggested in this paper, the scientific approach will be that of an experiment. An experiment in scientific terms is when observations and measurements are taken on dependent variables (Oates, Griffiths and McLean, 2022). In the case of Research Question 1, the dependent variables are response times, throughput and payload sizes. These are the elements we want to measure as change is applied to the independent variables, which are the network types in which GraphQL and REST will be tested. A regular WiFi network versus a mobile network will be the only change made to then measure the effect that change has on response times, throughput and payload sizes. The fewer independent variables possible, the better – this will have the added benefit of making a stronger case that surely the change of the independent variable is what has created the difference in the test results. As described in Oates, Griffiths and McLean (2022), the best case scenario is when only one independent variable is clearly isolated.

4.1.2 Hypothesis

In order to design and perform an experiment, a hypothesis should be formulated. A hypothesis is a statement that contains something that is measurable. In the case of Research Question 1, the hypothesis is:

GraphQL will outperform REST in certain cases when operating on a mobile network as payload size decreases.

This can be measured by designing an experiment in which the hypothesis can be tested. The following chapter on *Method RQ1* will describe the design.

4.1.3 Method RQ1

The first question will be a quantitative approach where performance tests and benchmarking procedures will be used to evaluate response times, throughput and payload size mainly in a simulated mobile environment. The purpose behind the mobile setting is that GraphQL was founded as a response to issues detected while delivering data to mobile devices with a large amount of concurrent users. To perform this quantitative analysis, a GraphQL implementation will be created alongside an already existing REST implementation. As a performance measuring tool, Apache's JMeter tool will be used since it is a well-renowned and well-established benchmarking tool for measuring response times, throughput and payload sizes.

JMeter is effective in simulations demanding concurrent users to effectively recreate a web service under heavy load and, thus, performing a so-called stress test regarding both data fetching paradigms. The results will be analyzed and translated into graphs and other visual datasets to simplify the presentation of results for future users of this research.

The queries will replicate the click and choice (of a user) to view a configurable product containing variants that is prone to deliver too much data, resulting in a replication of the over-fetching problem. JMeter will aid in simulating many concurrent users and by doing so, will test the technology against heavier usage (load).

The data itself is actual in-use data that a large corporation is using on their live site, and therefore is very well-suited for use in a real-world simulation approach testing REST and GraphQL. To be able to complete these queries, this study is dependent on many different objects carrying data, with some of these objects coming from separate sources. Because of this, both the over- and under-fetching issues can be viewed and analyzed directly during the test runs. The queries will be run in two scenarios: one where all data available is being fetched, and one where a portion of data is selected, enabling GraphQL's selective querying to be utilized.

4.1.4 Method RQ2

Hypothesis

The design of the server and settings for testing Research Question 1 will also be the foundation for answering Research Question 2, in which an optimal setting for GraphQL is to be identified among three different types of data sources. To formulate a hypothesis, a sort of prediction must be made (Oates, Griffiths and McLean, 2022).

GraphQL will have an optimal architecture placement in the landscape and show clear performance differences when compared to fetching data from Elasticsearch, Cosmos DB and REST (swagger) API endpoints.

The second research question is about finding the setting where GraphQL can be utilized most efficiently. To determine the result of this question, GraphQL will be benchmarked individually, and set up in three different scenarios. The goal is to measure anomalies in performance as GraphQL takes on separate positions in the architectural design.

Scenario 1:

GraphQL is implemented against Elasticsearch, working on one source with several different indexes to fetch and write to.

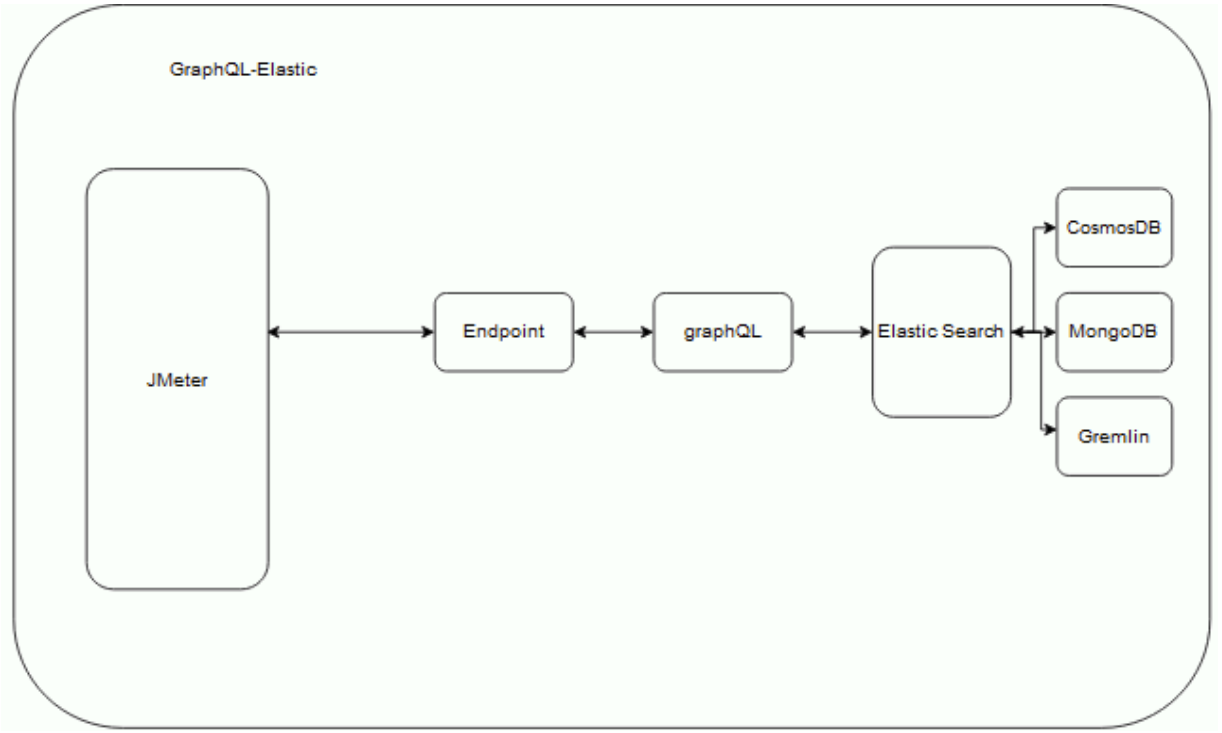


Figure 1: Setup for GraphQL-Elastic Server.

Scenario 2:

GraphQL is implemented to fetch data from a Cosmo DB running in the Microsoft Azure Cloud.

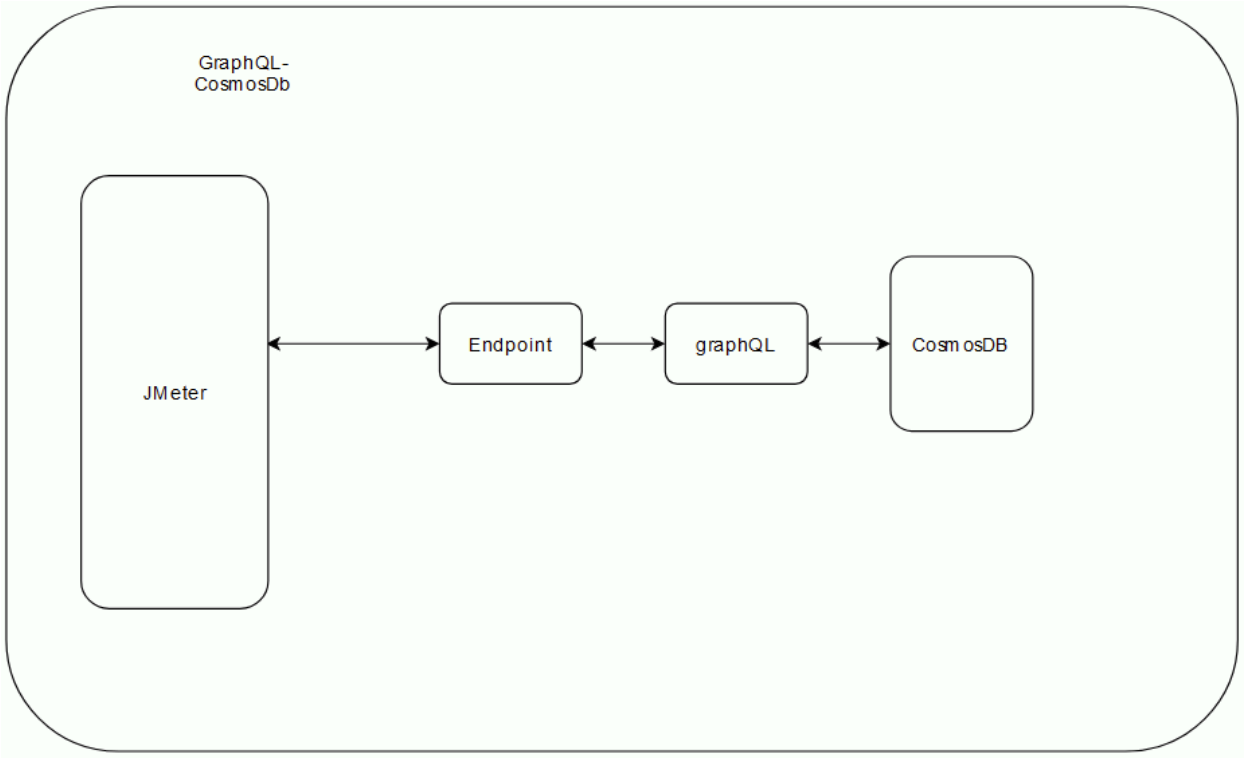


Figure 2: Setup for GraphQL-Cosmos Server.

Scenario 3:

GraphQL implemented on top of a layer of REST API endpoints to make the task of fetching data from REST more specific and streamlined through GraphQL queries, thus eliminating over-fetching and under-fetching issues.

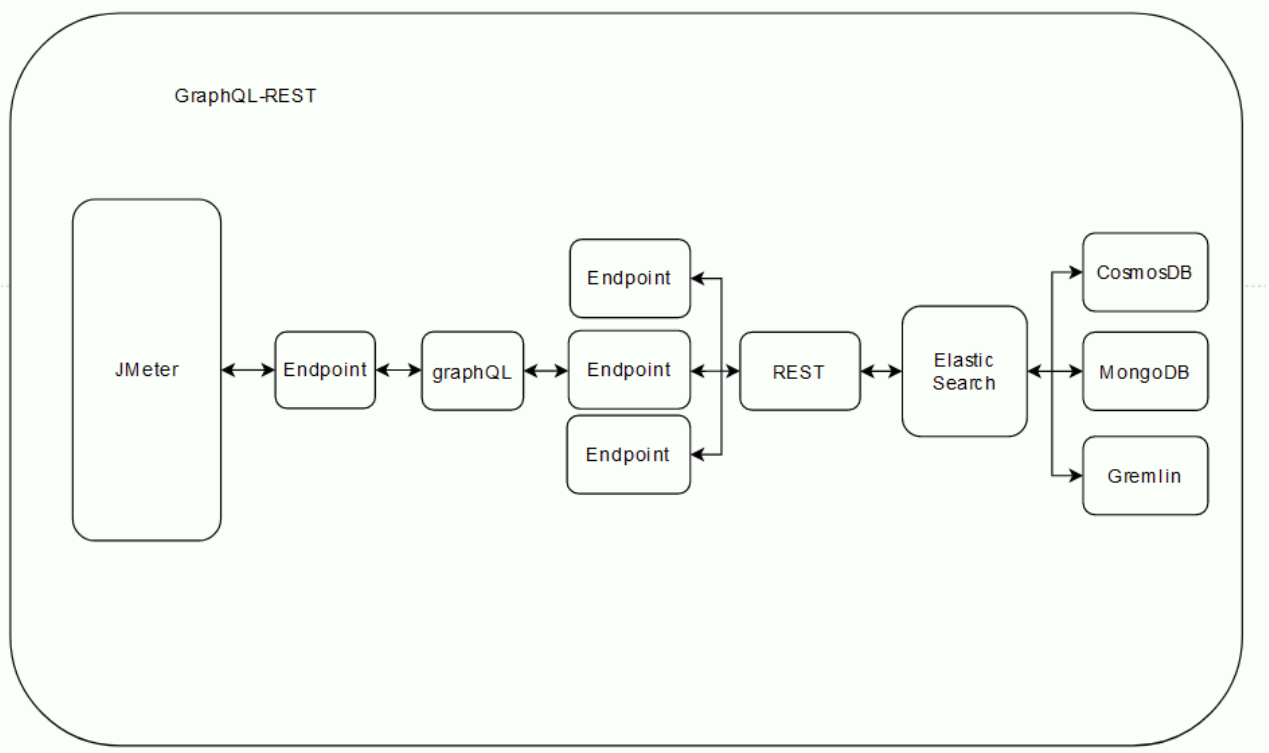


Figure 3: Setup for GraphQL-REST Server.

4.2 Dependent Variables and Independent Variables

Also in this case, the dependent variables are the response times, throughput and payload sizes of requests made to fetch data from a source. The difference this time is that REST doesn't have a place in this. Now, a performance test is run with only GraphQL as the subject. The independent variables in this case are instead the utilization of GraphQL. The server implementation will be fetching data from the three different sources and, as such, the source of the data will be the independent variables affecting the dependent ones in response times, throughput and payload sizes.

4.3 Ethical Considerations

Ethical aspects have affected the methods and research due to their view on confidentiality and the participants' rights to their own data. Such data can be personal information like social security or personal identical numbers, names or email addresses. With that in mind, this study has been designed to not use any data of such a nature in order to completely remove the issue of accidental personal information exposure. In addition, the company which owns and hosts the microservice that this study conducts its test on has been anonymized, and the text has been formulated in a general way such that it is applicable for others to use and draw insights from, without exposing what company the study has been performed within.

This research will not handle any sensitive information and will therefore not be violating any parts of GDPR, yet there are still other ethical aspects of performance research. This study aims to figure out not just the performance when it comes to faster responses and more efficient data handling, but also what technology can be set up in the most resource friendly way. This could assist further work and decisions regarding what technique to implement in a sense that, perhaps, it is not always the fastest response times that constitute the foundation of such decisions, but also resource utilization. If one paradigm shows to be more efficient than the other, can resource utilization be a more important factor? Can this also be an environmental question – meaning, how can we build more environmental systems that, in addition to user experience and efficient problem solving, also focus on being sustainable? Researchers have the responsibility to consider not just greater performance but also smarter resource utilization and less energy consumption through less computational power. Could this be one of the ways that system development contributes to such societal and environmental issues?

5 Material Presentation

5.1 Response Times

These are the response times in milliseconds (ms) measured while all four different implementations deliver a payload at around 13600 bytes. These graphs show user load with 500 concurrent users within a 5-second test period, then down through 300 users and on to 100 concurrent users during a 5-second test period.

5.1.1 GraphQL

WiFi

In this setting, the GraphQL server is configured to fetch data from REST endpoints. Here, average response times for the WiFi network are at 2872 ms for 500 concurrent users, while at 300 users there is an average response time at around 1295 ms, representing a drop of about 55%. When this implementation strategy gets tested for 100 concurrent users, the response times average at 382 ms, another sharp drop-off, this time by 70%.

4G

When tested via the mobile 4G network we see average response times at 6000 ms for 500 concurrent users, then a steep drop of about 70% for 300 users, down to 1790 ms. While running 100 users the response times bounced back up to 6800 ms.

WiFi 2.4Ghz

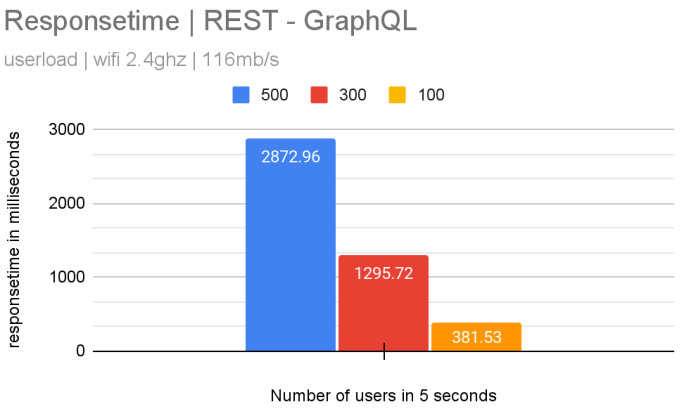


Chart 1: Response time during different user load on WiFi.

4G Mobile

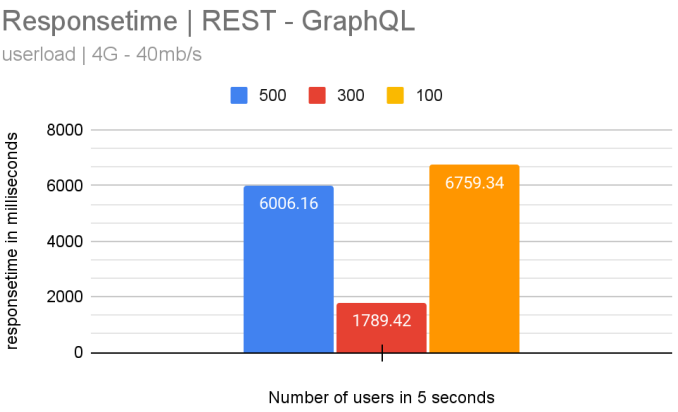


Chart 2: Response time during different user load on 4G.

5.1.2 GraphQL - Elasticsearch

WiFi

GraphQL fetching data from Elasticsearch. In this type of environment, GraphQL delivers average response times of about 70 ms regardless of user load. There is a slight drop of about 2 ms from 500 and 300 to 100 users but this is an insignificant drop within this context.

4G

While on the mobile network, Elasticsearch and GraphQL in combination have a 1411 ms average if 500 concurrent users fetch data in a 5-second period. When 300 users request data there is a steep drop of 83% to 239 ms, followed by around 125 ms when 100 users are tested.

WiFi 2.4Ghz

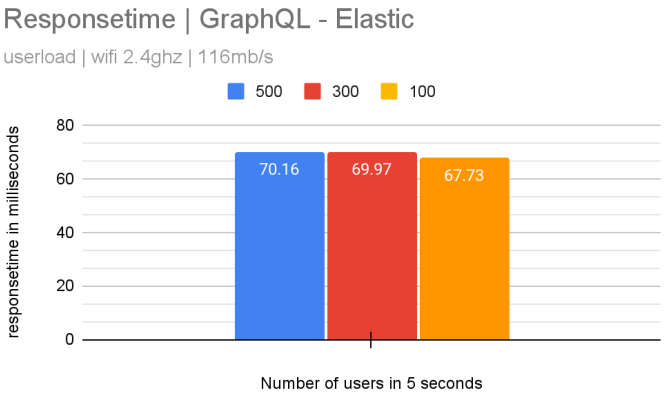


Chart 3: Response time during different user load on WiFi.

4G Mobile

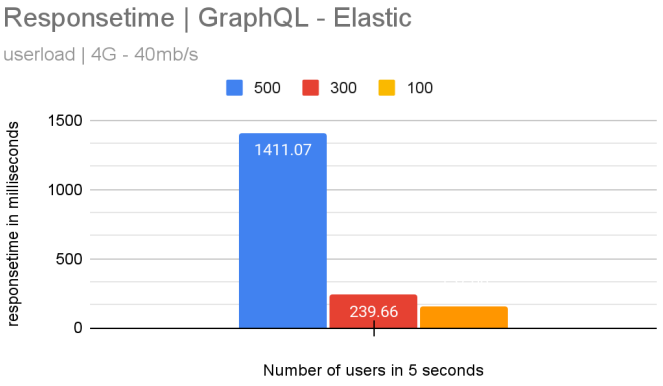


Chart 4: Response time during different user load on 4G.

5.1.3 *GraphQL - Cosmos*

WiFi

GraphQL and Cosmos DB have an average response time of around 8700 ms if 500 users request data concurrently. When tested with 300 users, almost a 50% drop appears down to 4700 ms followed by 1000 ms when 100 users are tested against this implementation.

4G

This implementation had the longest response times across the board while running on the 4G mobile network. Average response time for 500 concurrent users ended up at 72451 ms. At 300 users an average response time of 23100 ms was measured and at 100 users the measurements showed average times at 3000 ms.

WiFi 2.4Ghz

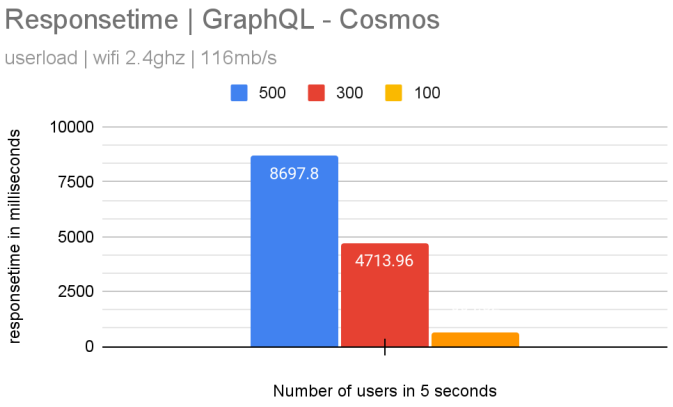


Chart 5: Response time during different user load on WiFi.

4G Mobile

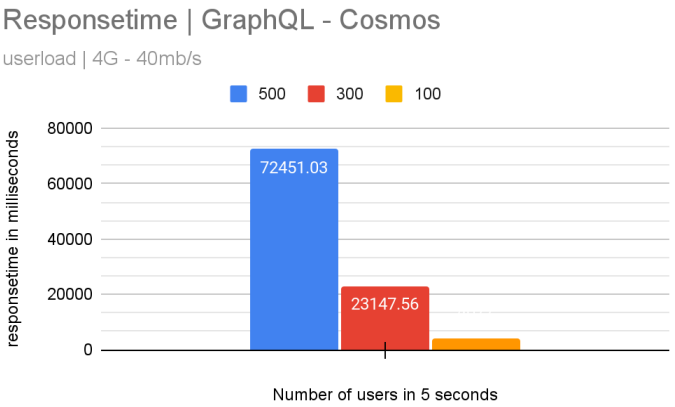


Chart 6: Response time during different user load on 4G.

5.1.4 REST

WiFi

While fetching data from traditional REST (swagger) endpoints, the average WiFi response times during a 500-user test landed at 4237 ms, followed by 941 ms for 300 users, a drop of 78%. This configuration delivers about 300 ms response times while 100 users are requesting data.

4G

The 4G mobile network delivers response times at 3344 ms for 500 users and then drops down by a little more than 50% to 1490 ms when 300 users are tested. For the 100-user test, the response times ended up at about 450 ms. The system is clearly able to handle the requests with better performance as user load decreases in regard to both networks.

WiFi 2.4Ghz

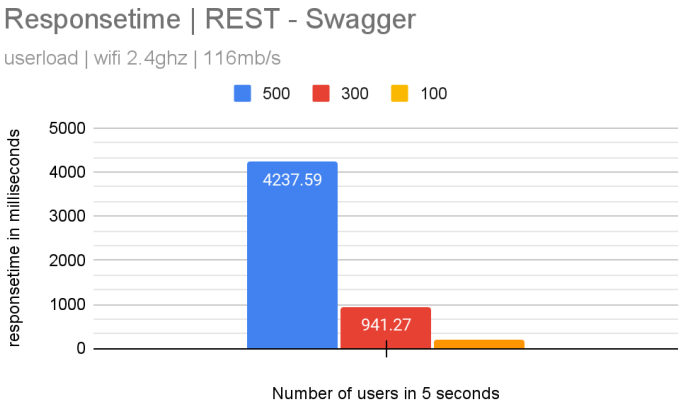


Chart 7: Response time during different user load on WiFi.

4G Mobile

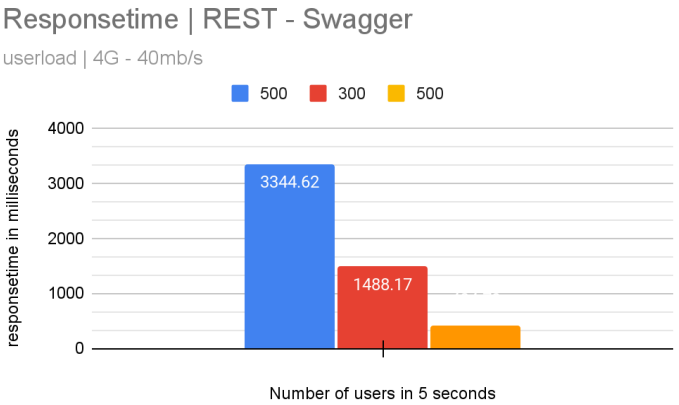


Chart 8: Response time during different user load on 4G.

5.2 Over-Fetching

These graphs show response times for all four different implementations while simulating the over-fetching issue with REST. A REST endpoint always deliver the same payload size. With GraphQL, there is the approach of choosing only the data you need, resulting in a smaller payload.

In this scenario, REST *will not be tested* as it cannot reduce its payload in the same way as GraphQL. The REST payload remains the same at 13600 bytes for all tests. This is where the claim that GraphQL can decrease the response times as a smaller payload is being transferred to the client.

5.2.1 GraphQL

WiFi

While fetching data from REST (swagger) endpoints, not including the GraphQL server, the average WiFi response times during a 500-user test landed at 4237 ms, followed by 941 ms for 300 users, a drop of 78%. This configuration delivers about 300 ms response times while 100 users are requesting data.

4G

The 4G mobile network delivers response times at 3344 ms for 500 users and then drops down with a little more than 50% to 1490 ms when 300 users are tested. For the 100-user test, the response times ended up at about 450 ms.

WiFi 2.4Ghz

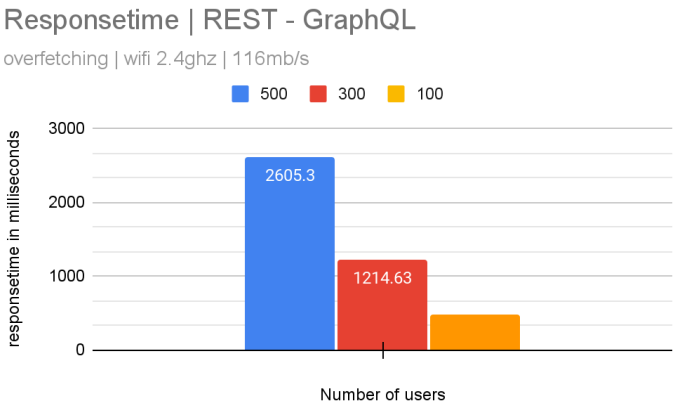


Chart 9: Response time while over-fetching during different user loads on WiFi.

4G Mobile

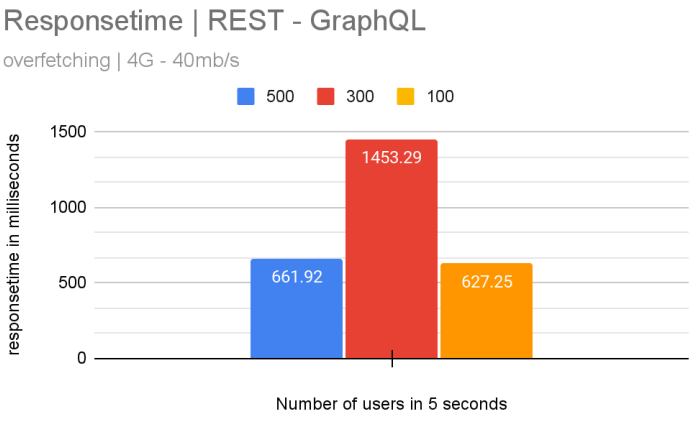


Chart 10: Response time while over-fetching during different user loads on 4G.

5.2.2 GraphQL - Elasticsearch

WiFi

GraphQL and Elasticsearch in a setting where the payload has been reduced from 13600 bytes down to 6000 bytes. On the WiFi network, the response times average out at around 70 ms regardless of the amount of concurrent users.

4G

In the mobile environment, Elasticsearch and GraphQL appear to average out at 282 ms whilst 500 users request data concurrently, and 159 ms for 300 users. When testing with 100 users, the response times stay at 134 ms.

WiFi 2.4Ghz

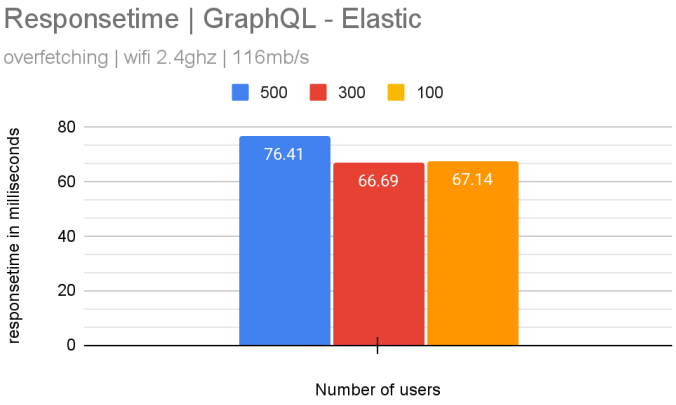


Chart 11: Response time while over-fetching during different user loads on WiFi.

4G Mobile

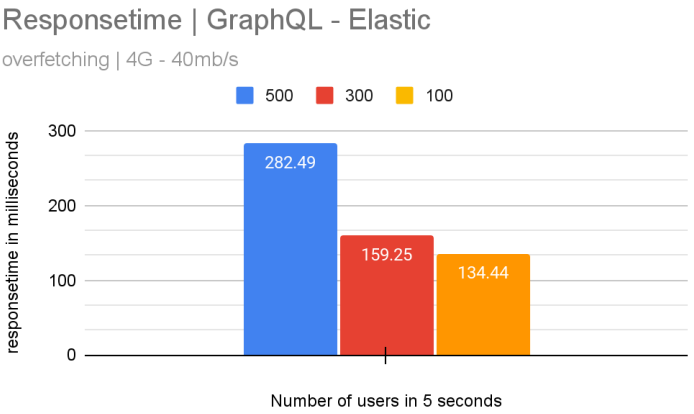


Chart 12: Response time while over-fetching during different user loads on 4G.

5.2.3 GraphQL - Cosmos DB

WiFi

Cosmos DB and GraphQL together have an average response time of 9500 ms during a 500-user test and a reduced payload. For 300 users, the response times dropped about 60% down to 3871 ms and then further down to about 400 ms.

4G

In this setting there is a very long response time at 86500 ms for 500 users. Meanwhile at 300 users the drop is massive down to 17900 ms, still a high number but much reduced from the 500-user test. At 100 users, the response times averaged at about 4200 ms.

WiFi 2.4Ghz

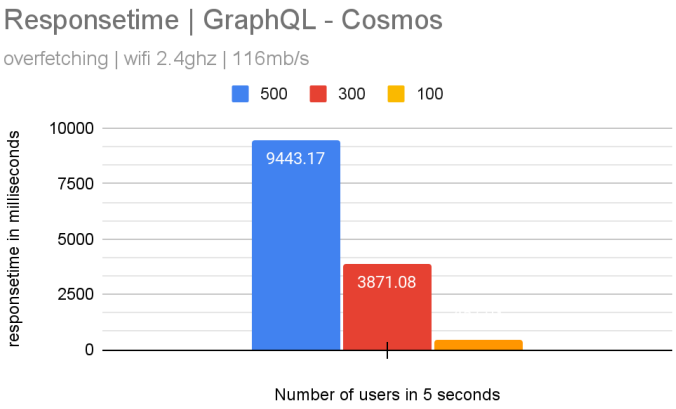


Chart 13: Response time while over-fetching during different user loads on WiFi.

4G Mobile

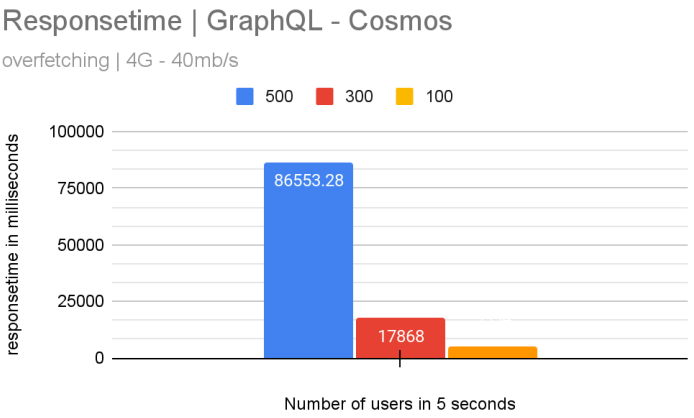


Chart 14: Response time while over-fetching during different user loads on 4G.

5.3 Throughput With Comparable Payload Size

Throughput in this context is transactions per second. The test time for these measurements has been set at 5 seconds of execution. Several tests have run for longer than that, and this will be discussed further in Chapter 6: Analysis. The throughput can be seen as the efficiency of the implementation but is not necessarily directly linked to the data source. Rather, the type of server implementation and the hardware it is running on.

An important factor in these charts is the designed test execution time of 5 seconds.

5.3.1 GraphQL - REST

WiFi

At 56 transactions per second, the 500-user test took just under 9 seconds to complete, and at 45 transactions per second, the 300-user case went down to a test execution time of 6.5 seconds. When hitting 100 users, the execution time decreased to 5.3 seconds.

4G

The mobile setting indicates that there are stable response times for both 500 users and 300 users at 43 transactions per second (11.67 seconds for 500 users and 7 seconds for 300 users). For 100 users, the measurement is 19 transactions per second and 5.2 seconds total for the test.

WiFi 2.4Ghz

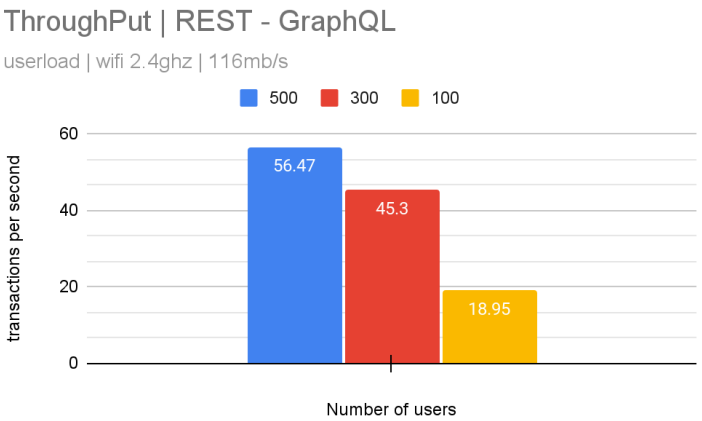


Chart 15: Throughput for the user load scenarios on WiFi.

4G Mobile

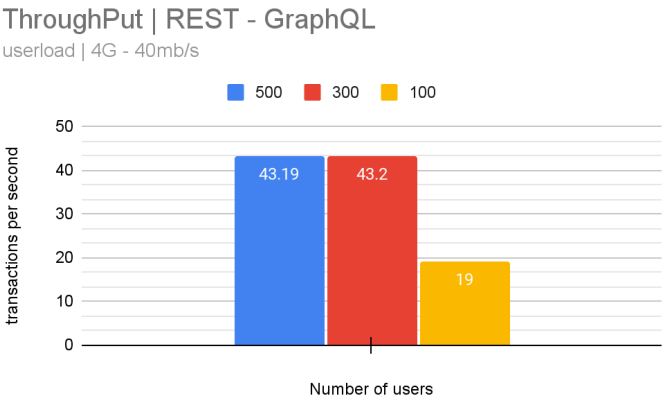


Chart 16: Throughput for the user load scenarios on 4G.

5.3.2 GraphQL - Elasticsearch

WiFi

Elasticsearch and GraphQL's efficiency on a WiFi network is showing 102 transactions per second for 500 concurrent users (4.9 seconds), for 300 users the throughput is at 60 transactions per second (5 seconds test run) and for 100 users, the results show another 5-second test at 20 transactions per second.

4G

For the mobile setting, the 500-user test turned back a result of 73 transactions per second, signifying an almost-7 second test, later dropping to 5 seconds for the 300-user test and another 5-second test at 20 transactions per second for the 100-user test.

WiFi 2.4Ghz

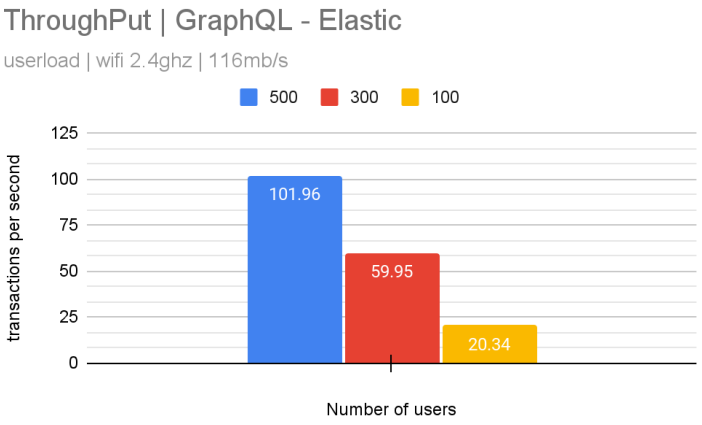


Chart 17: Throughput for the user load scenarios on WiFi.

4G Mobile

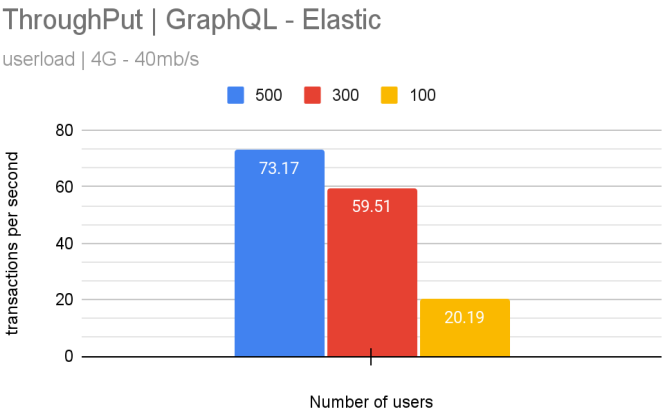


Chart 18: Throughput for the user load scenarios on 4G.

5.3.3 GraphQL - Cosmos DB

WiFi

For 500 users, while GraphQL utilizes Cosmos DB via Azure, a transactions per second measurement of 18 was observed, meaning a test time of 29 seconds. Then, while running 300 users, the test time dropped down to 13 seconds, less than half of the 500 users tested. Lastly, the 100-user test came back with data showing a test run at 6.5 seconds.

4G

Cosmos DB and GraphQL showed low throughput at 500 users with only 3.7 transactions per second. This test took 134 seconds, or a little over 2 minutes to complete. For 300 users, efficiency went up to 6 transactions per second resulting in a test run of 50 seconds, and for the 100-user test, efficiency went up a bit more, to 7.6 transactions per second and a total time of just about 13 seconds.

WiFi 2.4Ghz

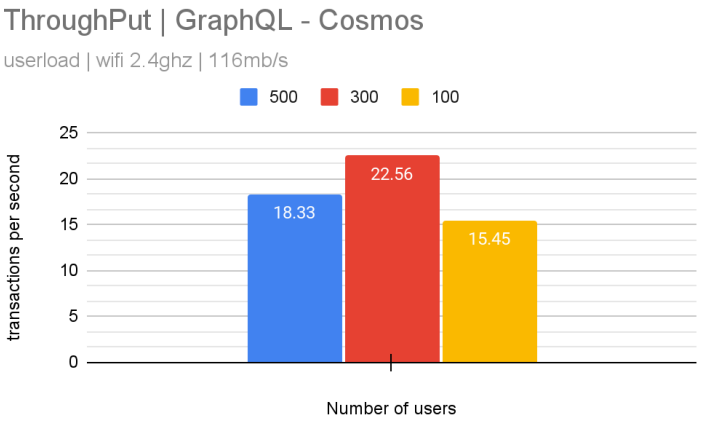
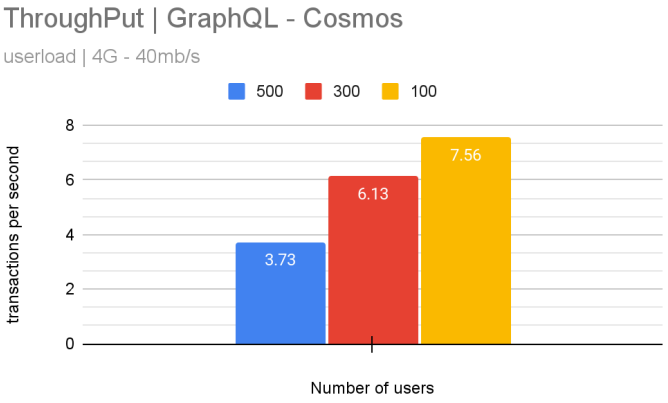


Chart 19: Throughput for the user load scenarios on WiFi.

4G Mobile



fChart 20: Throughput for the user load scenarios on 4G.

5.3.4 REST

WiFi

REST via swagger has a throughput of 42.5 transactions per second while being subjected to 500 requests. This means an almost 12-second test run at heavy load. For the smaller load of 300 users, REST throughput was observed at around 54.5 transactions per second and a 5.5-second test.

4G

In the mobile setting, the REST implementation shows throughputs at 54.75 transactions per second for 500 users, a 9-second test run, and then 43.71 transactions per second for 300 users, resulting in a test run just shy of 7 seconds. When measured against 100 users, the throughput was 20 transactions per second and, thus, the test run time was 5 seconds.

WiFi 2.4Ghz

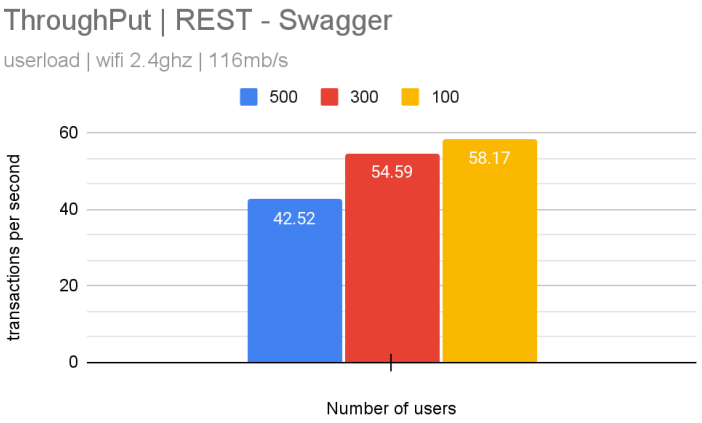


Chart 21: *Throughput for the user load scenarios on WiFi.*

4G Mobile

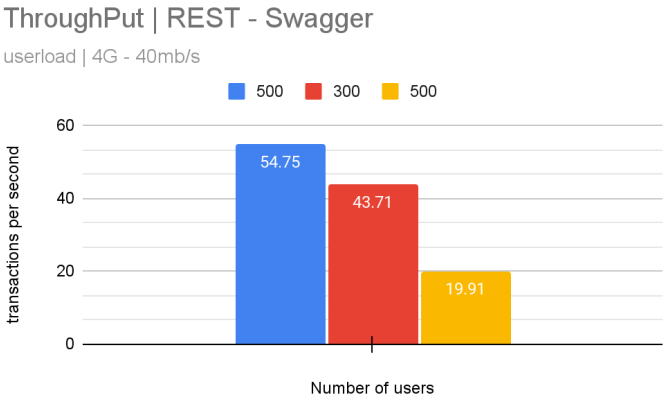


Chart 22: Throughput for the user load scenarios on 4G.

5.4 Throughput Over-Fetching

In these graphs, there are visualizations of the efficiency of the three different GraphQL implementations as they utilize their capacity to select precisely what is asked for in terms of data from their respective sources. A REST implementation has a fixed byte size to each endpoint and, therefore, it lacks the capability to reduce its payload size dynamically.

5.4.1 GraphQL - REST

WiFi

The GraphQL setting, which fetches data from a layer of REST endpoints, has an average transactions-per-second measurement of 55.6. This equates to a test run of 9 seconds. The version with 300 users shows a throughput of 44.8 transactions per second and a 6.7-second test run. When the 100-user test was executed, the results showed 19 transactions per second, or a 5.2-second test.

4G

In this setting, the efficiency goes from 18.5 transactions per second at 500 users, to 47.2 at 300 users and then back down to 18.3 for 100 users. This equates to test runs being carried out in this type of environment of 27, 6.3 and 5.4 seconds, respectively.

WiFi 2.4Ghz

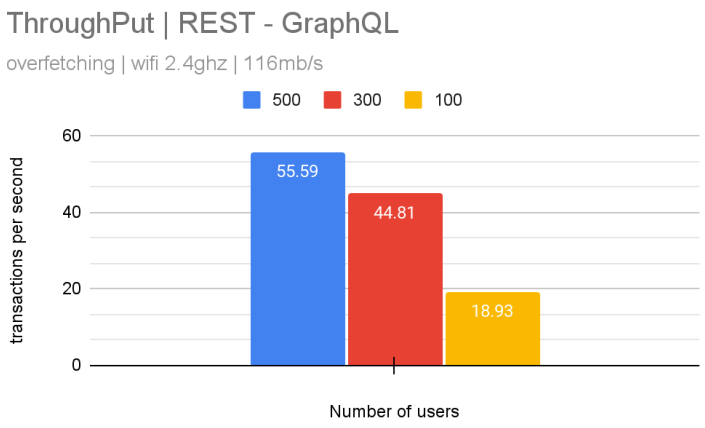


Chart 23: Throughput for over-fetching during different user load scenarios on WiFi.

4G Mobile

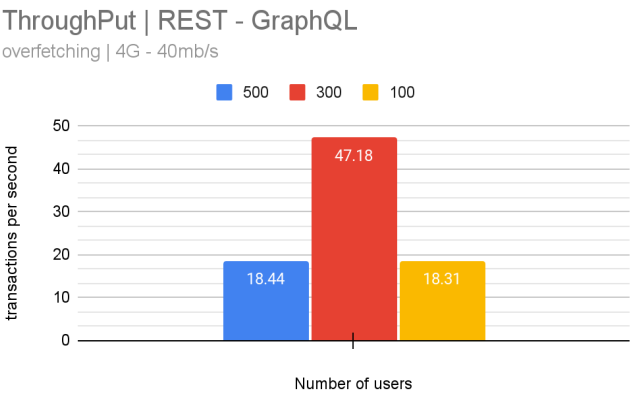


Chart 24: Throughput for over-fetching during different user load scenarios on 4G.

5.4.2 GraphQL - Elasticsearch

WiFi & 4G

This type of environment where GraphQL fetches data from Elasticsearch shows very similar results in 101-102 transactions per second regarding the 500-user test, which means 4.9-second tests at that level. For 300 users, the results show transactions at 59 per second, this too resulting in a 5-second test for both types of network. The 100 users tests are similar as well, as they both result in 20 transactions per second meaning 5-second tests.

WiFi 2.4Ghz

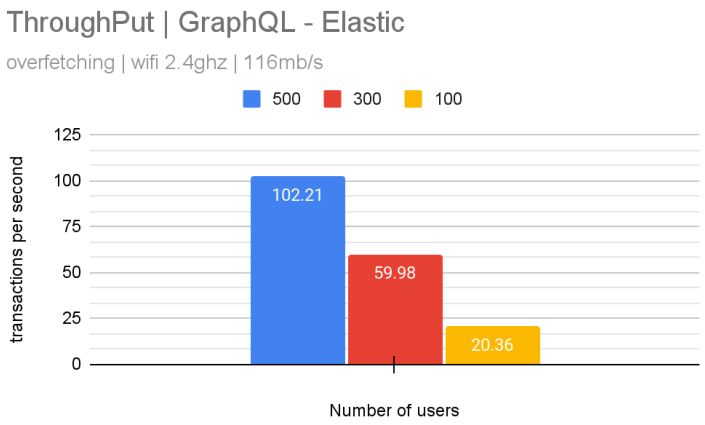


Chart 25: Throughput for over-fetching during different user load scenarios on WiFi.

4G Mobile

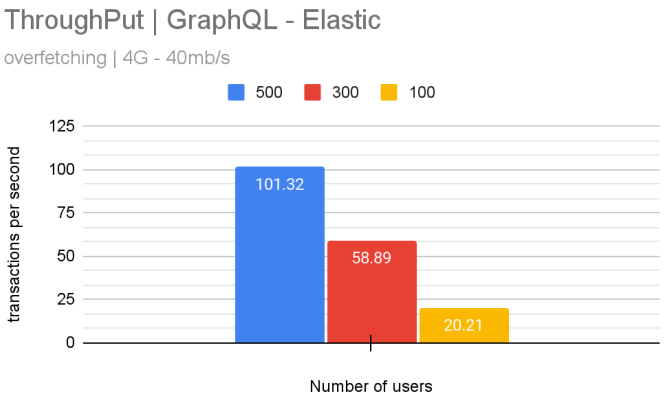


Chart 26: Throughput for over-fetching during different user load scenarios on 4G.

5.4.3 GraphQL - Cosmos DB

WiFi

Cosmos DB in combination with GraphQL shows a throughput at 16.5 transactions per second, and a more than 30 seconds long test run. For 300 users, transactions were at 21 per second and a 14-second test run was executed. Lastly, at 100 users, the test time stopped at 5.3 seconds and the throughput was observed at almost 19 transactions per second.

4G

Cosmos DB and GraphQL in a smaller payload scenario resulted in, for the 500 users test, transactions per second at 3.5, meaning a 111-second long test run. As for the 300 users case, the results showed 31.5 seconds, a steep increase in efficiency. Then at 100 users, the efficiency dropped down to 6.6 transactions per second and a 15-second test execution.

WiFi 2.4Ghz

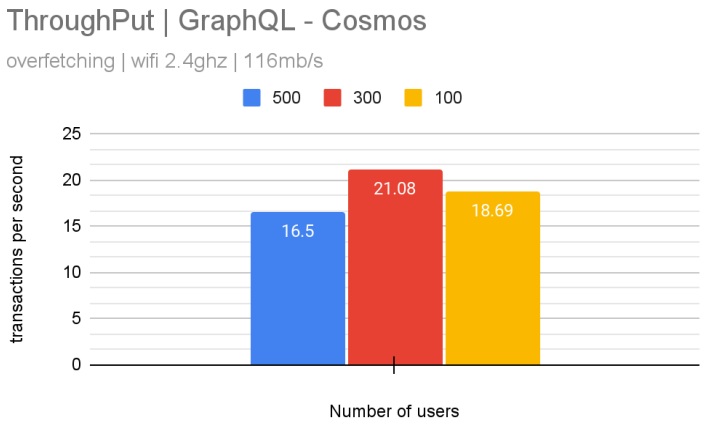


Chart 27: Throughput for over-fetching during different user load scenarios on WiFi.

4G Mobile

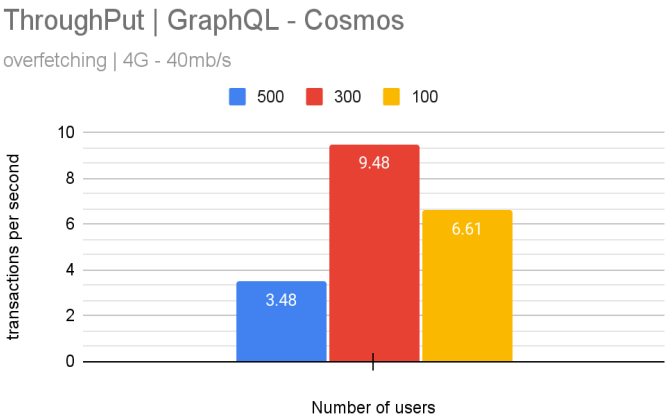


Chart 28: Throughput for over-fetching during different user load scenarios on 4G.

5.5 Payload

The following charts represent payload changes in the regular and the over-fetching scenarios. In the user load chart, each implementation delivers about 13600 bytes. This is the baseline due to the fact that it is the size of the REST implementation's payload size. Then, in the over-fetching case, the GraphQL server versions are all utilizing their capacity of picking and choosing and by doing so, have selected less than 50% of what the REST version delivers. This simulates the over-fetching situation, where a response will only carry what has been asked of it in terms of specific fields and objects.

User Load

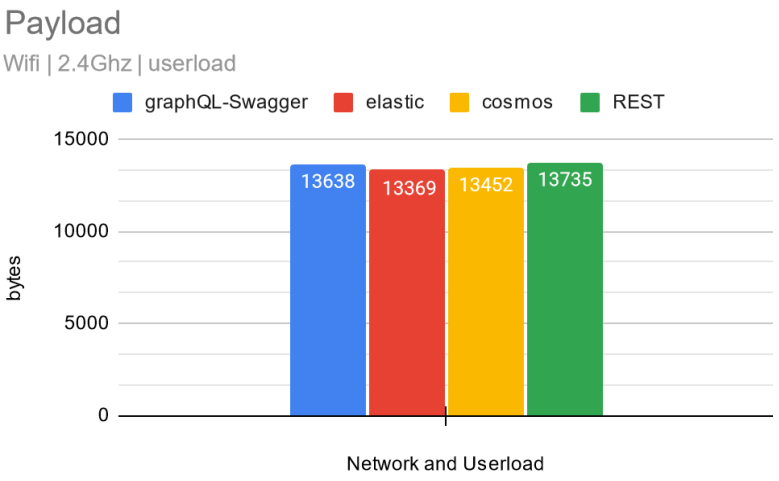


Chart 29: Payload sizes with a spread within a range of 366 bytes.

Over-Fetching

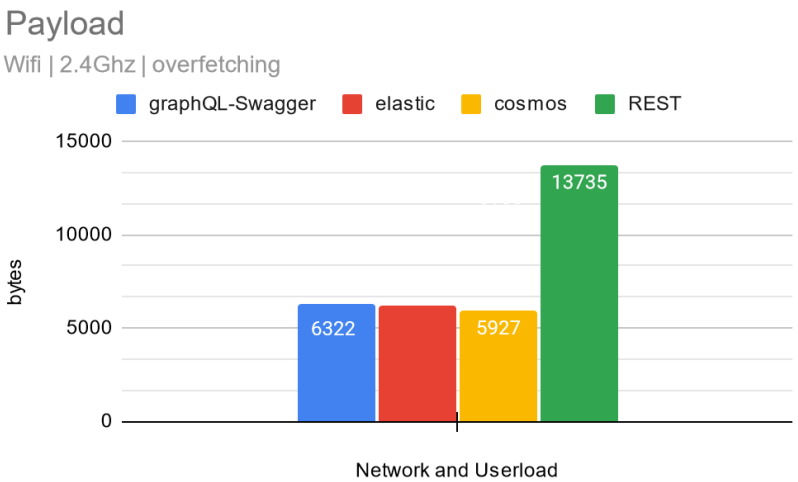


Chart 30: Payload sizes change as GraphQL utilizes its functionality for choosing specific data instead of all of it by default.

6 Analysis

To analyze the data, each network type and user load setting will be briefly discussed. Then, they will be put into comparison with relevant others to show patterns found in the datasets. Since the WiFi network has a download speed of 116 mb/s and the mobile 4G network has an average speed of 40 mb/s, it is not possible to definitely compare response times in between the two networks. Instead, efficiency changes relative to payload size decreases within each implementation will be observed.

6.1 500 Users

6.1.1 Response times -same payload size

WiFi 2.4Ghz

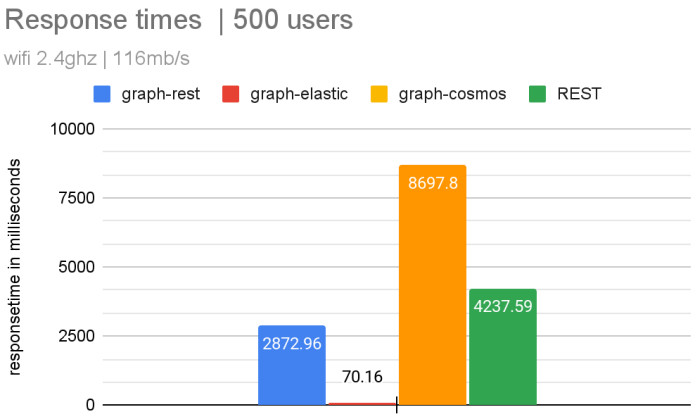


Chart 31: Response times for each implementation for 500 concurrent users on the WiFi network.

4G Mobile

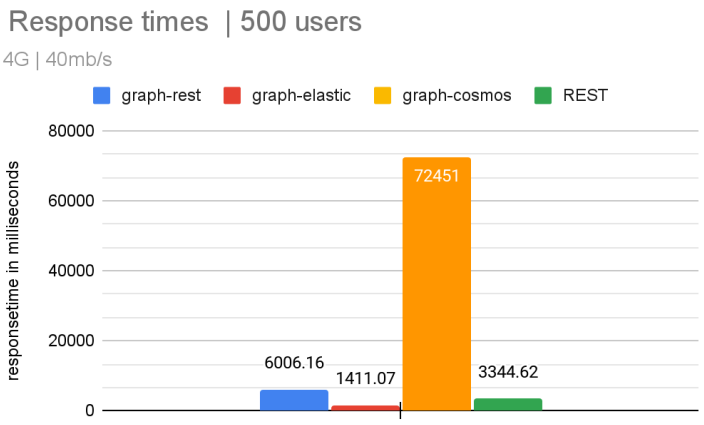


Chart 32: Response times for each implementation for 500 concurrent users on the 4G network.

The charts above depict the response times for each of the four server versions. The 500-user test is the heaviest test where 500 requests are made within a 5-second period. At these high levels, the results show some volatility in response times especially regarding the GraphQL-Cosmos DB implementation where response times are particularly high. An interesting observation is that in the 4G network, the response times skyrocket to many times longer compared to the WiFi version, suggesting anomalies in request handling that is either network- or server logic-driven. Potential fluctuations in the mobile network could also be a reason, although there are similar results in Cosmos DB's efficiency throughout the testing suggesting that this type of implementation is extra heavy to process server-side.

There are other patterns to be observed, as well. It can be seen that the profile of both networks are very similar, meaning that the charts have the same characteristics in terms of the order of the highest and lowest response times measured. When comparing to what Vohra and Manuaba (2022) observed, which is that REST had better response times when compared to GraphQL running with a RDBMS implementing SQL queries, there are evidence here to suggest that REST does in fact have better response times than two of the three GraphQL servers. However, there are also implementations where GraphQL can outperform REST, such as the GraphQL-Elastic server of this study.

The 4G network has much higher response times, but it is important to keep in mind the differences in WiFi download speeds where the WiFi version has 65% stronger downloading speed of 116 mb/s compared to 4G with 40 mb/s. Because of this, it is

important to not make conclusions based only on comparison between the two networks' response times, but rather on efficiency and how it changes as payload changes.

6.1.2 *Response times - over-fetching*

WiFi 2.4Ghz

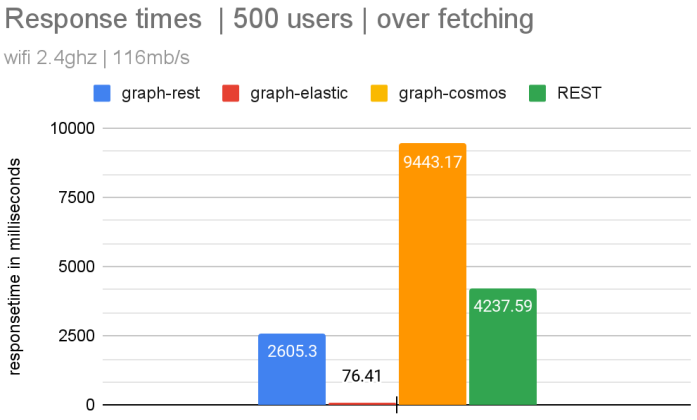


Chart 33: Response times for each implementation for 500 concurrent users on the WiFi network.

4G Mobile

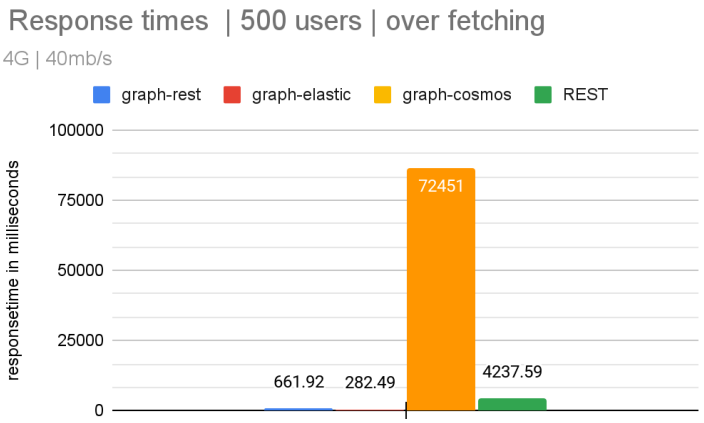


Chart 34: Response times for each implementation for 500 concurrent users on the mobile network.

In the charts above, the results for the over-fetching scenario are displayed. There is a slight decrease in response times from 2872 to 2600 ms regarding the GraphQL-REST server when the payload size shrunk by almost 50%.

For Elasticsearch, the response times went up by about 9% from 70 to 77.5 ms. Cosmos DB also increased its response times in the over-fetching scenario. Therefore, as the GraphQL servers utilize the functionality of selecting and filtering data, 2 out of 3 server implementations increase their response times. This could potentially be due to the findings of Marchuk, Dyyak and Makar (2023), where the authors concluded that GraphQL queries were heavier to process, which supports the findings of this study where it seems clear that the performance of GraphQL heavily depends on what type of data source is being queried.

6.1.3 Throughput

WiFi 2.4Ghz

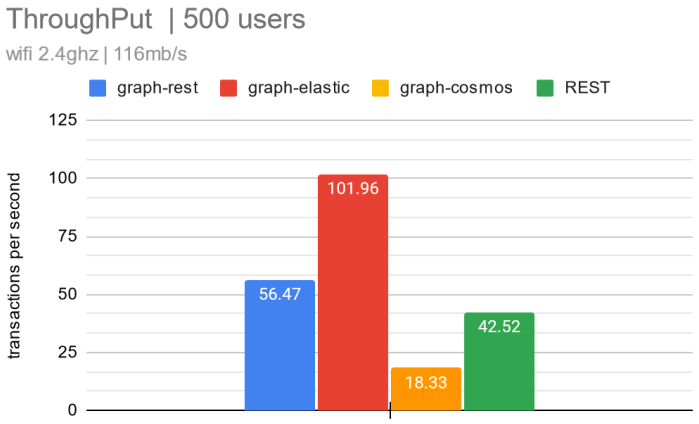


Chart 35: Throughput for each server version during the 500-user test on the WiFi network.

4G Mobile

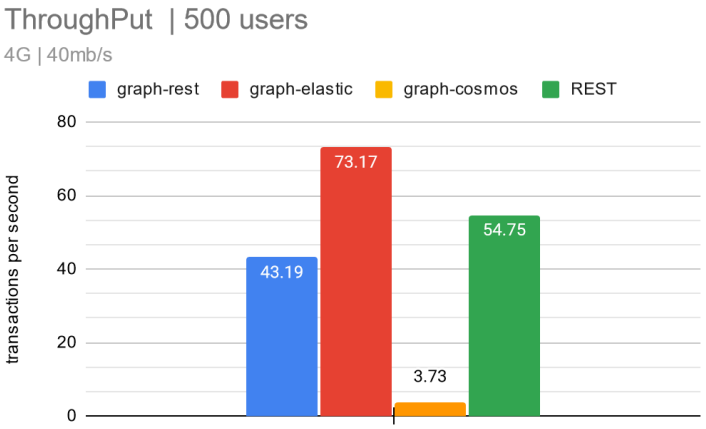


Chart 36: Throughput for each server version during the 500-user test on the 4G network.

It is important to keep in mind that the test is designed to fire 500 requests in a 5-second period. It is then up to the server to handle and process them. Since this is a 500-user test during 5 seconds, for a server to have successfully handled all queries within that period, a throughput of 100 transactions per second are needed. During the WiFi network test, the GraphQL-Elastic implementation was the only variant to handle this task within the set time, followed secondly by the GraphQL-REST server with 56.5 transactions per second.

Elasticsearch is the most efficient on the mobile network environment also, but it didn't quite manage to handle all 500 requests within the given timeframe. Considering the 65% drop in downloading speed, it is interesting to observe that the drop in throughput is much less across the board.

Test execution time

To put the efficiency into perspective, the table below converts the throughput to seconds. This metric shows the total time it took to execute the test. Each test run has been designed to send 500 requests in 5 seconds, and the server can handle the queue and process the requests within the server's capacity for that particular implementation. The test execution times are an effective way to get information regarding which network is most efficient since it tells us whether the server could handle the test or if it needed more time.

Table 1: User Load Scenario | 500 Users

Server version | Network	WiFi| Execution time	4G | Execution time
GraphQL-REST	8.85 seconds	12 seconds
GraphQL-Elastic	4.90 seconds	6.83 seconds
GraphQL-Cosmos	27.28 seconds	134 seconds
REST	35.28 seconds	27.40 seconds

Table 2: Over-Fetching Scenario | 500 Users

Server version | Network	WiFi | Execution time	4G | Execution time
GraphQL-REST	8.99 seconds	27.11 seconds
GraphQL-Elastic	4.89 seconds	4.93 seconds
GraphQL-Cosmos	90.91 seconds	143.68 seconds

Summary

The user test level where 500 requests were sent to each of the server versions appears to have been tough for the servers to handle within the set time of 5 seconds. Overall, Elasticsearch cleared 3 out 4 tests within that time and Cosmos DB had the toughest time handling this type of volume with some tests even passing 2 minutes of run time. This aligns with findings that GraphQL implementations can be resource-intensive and may require additional processing time compared to traditional REST APIs (Vohra and Manuaba, 2022).

6.2 300 Users

6.2.1 Response times - same payload size

WiFi 2.4Ghz

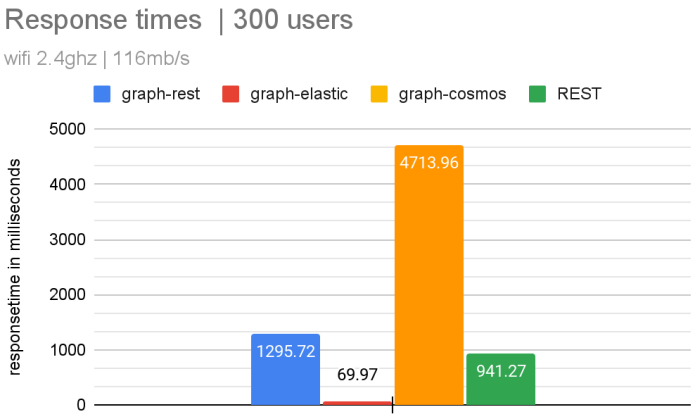


Chart 37: Response times for each implementation for 300 concurrent users on the WiFi network.

4G Mobile

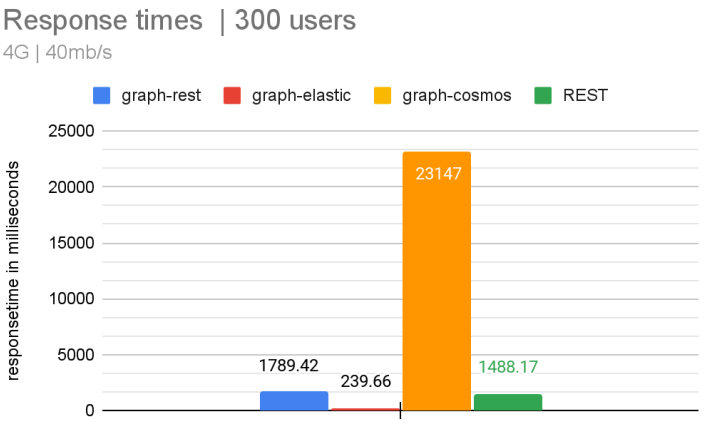


Chart 38: Response times for each implementation for 300 concurrent users on the 4G network.

In this version, where the tests are configured to run 300 requests in a 5-second period, response times drop across the board yet the characteristics of the charts remain similar to the 500-user case. GraphQL-Cosmos had the worst response times in both network environments and GraphQL-Elastic was the fastest version to respond with its response time of 70 ms on WiFi and 240 ms on the mobile network. Noteworthy here is that Elastic takes 4x longer to respond on the 4G network compared to WiFi, while GraphQL-REST only sees an increase in response times of about 38% when tested on the 4G mobile network.

Response times for REST increase by just above 500 ms when on 4G settings, with Cosmos DB still remaining the outlier in response times of over 23000 ms.

6.2.2 *Response times - over-fetching*

WiFi 2.4Ghz

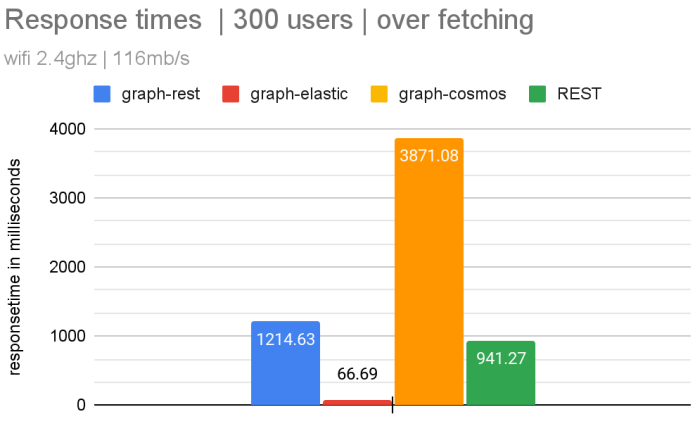


Chart 39: Response times for each implementation for 300 concurrent users on the WiFi network.

4G Mobile

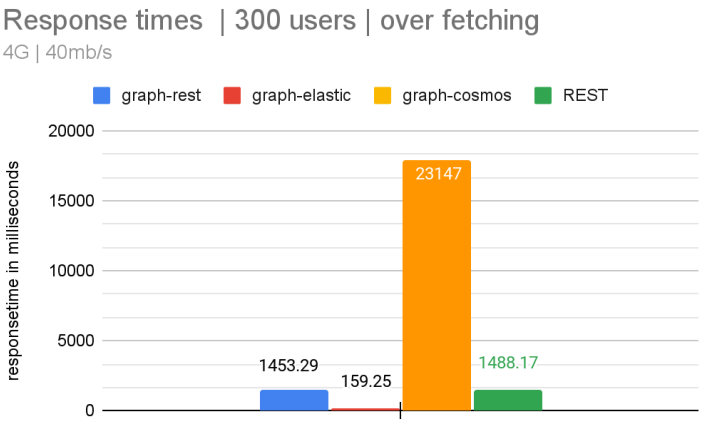


Chart 40: Response times for each implementation for 300 concurrent users on the 4G network.

For the over-fetching scenario during the 300-user test, GraphQL-Cosmos comes down a bit compared to its user load version with almost 900 ms, while GraphQL-REST version shows an insignificant change of 80 ms between user load and over-fetching scenarios. This indicates that the payload size decrease didn't seem to have had a massive impact on response times for this type of server implementation.

Elasticsearch stays around 65-70 ms on WiFi settings, and then shoots up to 160 ms while on the 4G mobile network. During the user load test, Elasticsearch had response times of 240 ms which might indicate that there was a higher efficiency observed when the payload size was decreased. In a previous study, it was stated that the performance of GraphQL heavily depends on its implementation and execution strategy (Roksela, Konieczny, and Zielinski, 2020). This has been shown accurately throughout the results of this paper as GraphQL-Elastic has consistently had superior results, while GraphQL-Cosmos has been performing very poorly compared to other versions of GraphQL servers and also the transitional REST server.

6.2.3 Throughput

WiFi 2.4Ghz

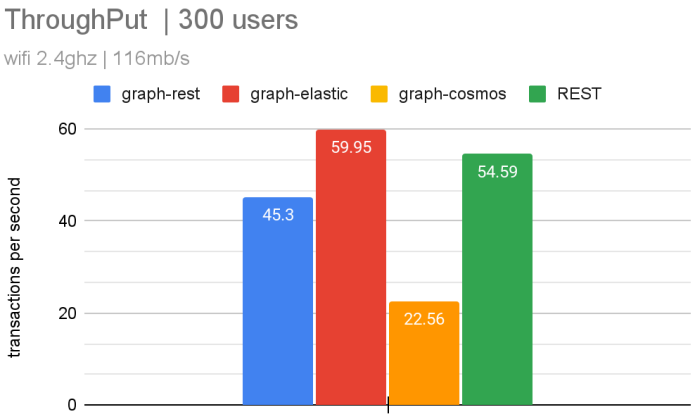


Chart 41: Throughput for each server version during the 300 users test on WiFi network.

4G Mobile

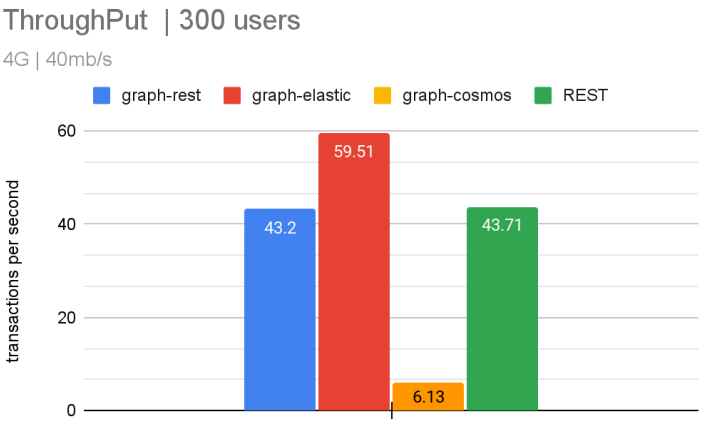


Chart 42: Throughput for each server version during the 300 users test on 4G network.

The throughput measurements for the 300 requests case show that on WiFi settings, both GraphQL-Elastic and the REST implementation had similar results in terms of efficiency even though GraphQL-Elastic came out on top with almost 60 transactions per second, or a 5-second test. This remains the same for the 4G setting where Elasticsearch alone managed to handle the 5-second test on both networks.

REST reaches its best results on the WiFi setting with a throughput of almost 55 transactions per second while on 4G, REST response times are observed at 44 transactions per second.

Cosmos DB yet again appears to have had the worst efficiency at 22.5 transactions per second on WiFi and only 6 transactions per second in the mobile setting while GraphQL-REST performed equally in both networks.

Test execution time

At the 300-user level, we see more stable results in terms of servers managing to handle the amount of requests being sent to a greater extent. Elasticsearch still shows the most stable results by managing 5-second test runs in all settings be it user load, over-fetching or network. GraphQL-REST now seems to stabilize at just around 7-second test runs, indicating that the server now can handle the computation needed to run these types of implementations.

The GraphQL-Cosmos server sticks out with its slow response times and poor ability to complete the test runs in time. With that said, there is a steep increase in efficiency in terms of GraphQL-Cosmos when comparing between 500 and 300 users.

Table 3: User Load Scenario | 300 Users

Server version | Network	**WiFi| Execution time**	**4G | Execution time**
GraphQL-REST	6.62 seconds	6.94 seconds
GraphQL-Elastic	5 seconds	5.04 seconds
GraphQL-Cosmos	13.30 seconds	48.94 seconds
REST	16.49 seconds	20.59 seconds

Over Fetching Scenario | 300 Users

Server version | Network	**WiFi| Execution time**	**4G | Execution time**
GraphQL-REST	6.69 seconds	6.36 seconds
GraphQL-Elastic	5 seconds	5.09 seconds
GraphQL-Cosmos	14.23 seconds	31.65 seconds

Summary

These servers now seem to be able to handle the request volumes and test run time constraints to a greater extent, yet there are some interesting deviations such as fetching data from Cosmos DB via GraphQL that appears to be heavy to process. This could be due to the logic in the GraphQL server not being properly streamlined, but it can also indicate that processing Cosmos DB queries are heavier work than Elasticsearch or REST requests, which has been previously discussed in this paper, as well as in the findings of other studies such as Marchuk, Dyyak and Makar (2023), where it was noted that GraphQL queries seem to be heavier in logic execution than REST requests.

6.3 100 Users

6.3.1 Response times - same payload size

WiFi 2.4Ghz

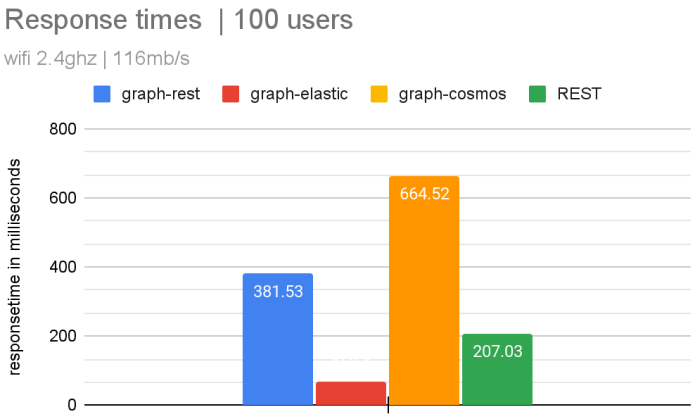


Chart 43: Response times for each implementation for 100 concurrent users on the WiFi network.

4G Mobile

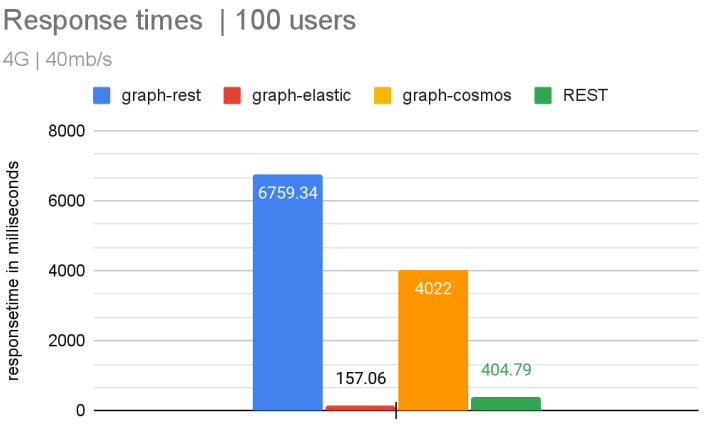


Chart 44: Response times for each implementation for 100 concurrent users on the 4G network.

During these types of settings, when 100 requests are being sent to the servers, mostly on the WiFi network configuration, there are what seems to be pretty stable levels of response times for all server types. Elasticsearch still has by far the lowest response times at 68 ms on WiFi and 157 ms on the 4G network. Right behind GraphQL-Elastic, the REST server responds after 207 ms on WiFi and 405 ms in 4G network.

There is an anomaly in the data regarding GraphQL-REST response times on the 4G network – response times shoot up to almost 7000 ms. This could be due to fluctuations in the mobile network, though tests were run on several different occasions and similar results were observed time and time again. It could also be due to the fact that servers and clients are being run on the same machine, risking resource exhaustion of the hardware capabilities.

6.3.2 Response times - over-fetching

WiFi 2.4Ghz

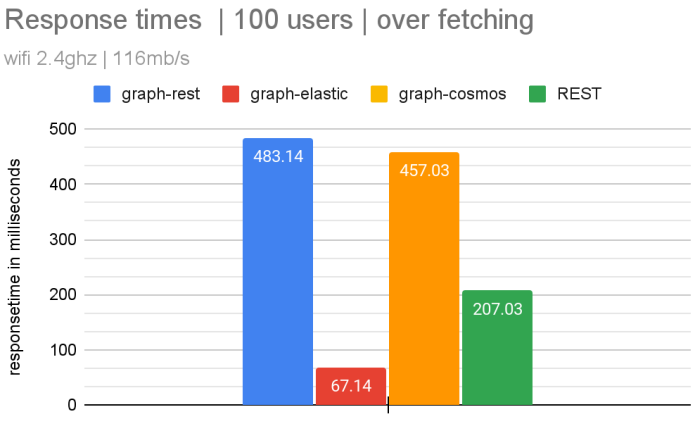


Chart 45: Response times for each implementation for 100 concurrent users on the WiFi network.

4G Mobile

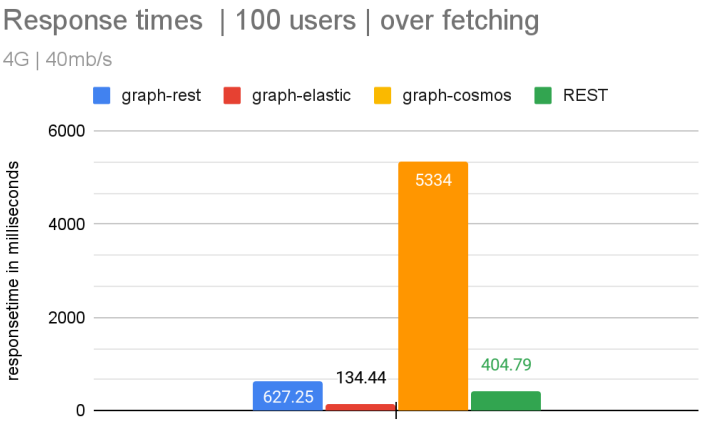


Chart 46: Response times for each implementation for 100 concurrent users on the 4G network.

In this over-fetching scenario, GraphQL-REST goes up 100 ms compared to the user load scenario on the previous page. GraphQL-Elastic remains the fastest in terms of response times and doesn't change much at all even though the payload size has decreased substantially. With GraphQL-Cosmos, the expected characteristics of being one of the slowest performers was evident. On the WiFi network, Cosmos DB does alright with response times around 457 ms; however, looking at 4G, response times are above 5000 ms. The REST implementation stays rather stable on both networks with 207 ms on WiFi and 405 ms on 4G.

6.3.3 Throughput

WiFi 2.4Ghz

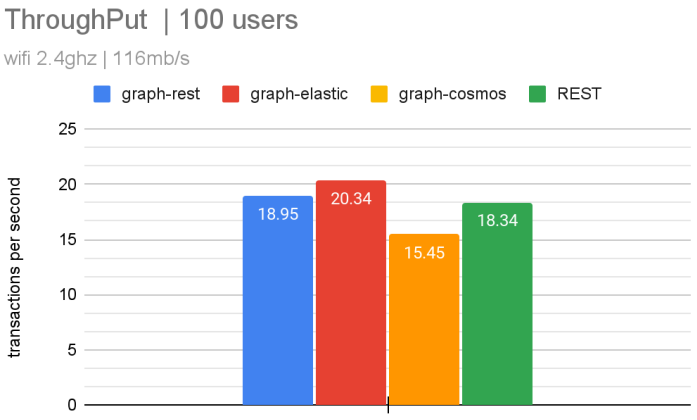


Chart 47: Throughput for each server version during the 100 users test on WiFi network.

4G Mobile

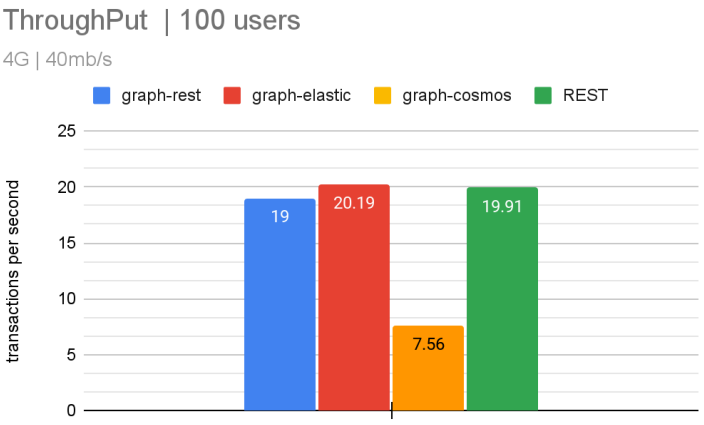


Chart 48: Throughput for each server version during the 100 users test on 4G network.

In terms of stability, it is at the 100 requests level that we see comparable efficiency, in particular for the WiFi network where transactions per seconds ranged from 15.45 to 20.34 transactions per second. All servers stay stable also on the 4G network, though Cosmos DB does drop down by about 7 transactions to 7.5 transactions per second. This again supports the work of Roksela, Konieczny, and Zielinski (2020), where it was concluded that GraphQL is heavily influenced by execution strategy, further strengthening the notion that GraphQL implementation needs to be well-fitted depending on the circumstances and that it is an optimization question requiring careful consideration before being acted upon.

Test execution time

At this level it is clear that almost all configurations can handle the incoming requests within the set time frame of 5 seconds. REST and GraphQL-Cosmos show slower execution times while tested on the 4G network. On the WiFi network, there is stability with servers handling the requests in both the user load and over-fetching scenarios.

Table 5: User Load Scenario | 100 Users

Server version | network	WiFi| Execution time	4G | Execution time
GraphQL-REST	5.28 seconds	5.26 seconds
GraphQL-Elastic	4.92 seconds	4.95 seconds
Graphql-Cosmos	6.47 seconds	13.23 seconds
REST	5.16 seconds	15.07 seconds

Table 6: Over-fetching Scenario | 100 Users

Server version | network	WiFi| Execution time	4G | Execution time
GraphQL-REST	5.28 seconds	5.46 seconds
GraphQL-Elastic	4.91 seconds	4.95 seconds
Graphql-Cosmos	5.35 seconds	15.13 seconds

6.4 A Closer Look At Network, Over-Fetching and Response Times

In Table 7 below, an average efficiency increase for each server and network version as the payload goes from 13600 bytes down to 6000 bytes is shown. On the WiFi setting, the change is rather small and it would seem that the smaller payload does not have that much to do with response times on that type of network environment. It is a different case on the 4G network though, where both GraphQL-REST and GraphQL-Elastic increase their efficiency with 69% and 83%, respectively. Looking at these numbers, it would appear that the smaller payload does have a potentially notable effect on the 4G mobile network in terms of response times. At the same time, Cosmos DB decreases efficiency by 10%, so it is not always so that the 4G network will increase the efficiency of the response times.

One possible reason for the outcome we see in Table 7 below is that the WiFi network is often more stable. It usually doesn't serve an arbitrary amount of users, but is more static in that sense. Meanwhile, the 4G mobile network fluctuates in capacity due to signal and available bandwidth at the time. Similar network-related performance issues were discussed in studies like Brito and Valente (2020). So, as the 4G network loses capacity, it would make sense that a smaller payload could make it more efficient. This study has not looked into fluctuating levels of network download speed and are not able to draw any direct conclusions based on that.

Table 7: Efficiency Changes

Server version | network	**User load response times**	**Over-fetching response times**	**Efficiency change**
GraphQL-REST WiFi (ms)	1517	1434	+ 6%
GraphQL-Elastic WiFi (ms)	69	70	- 1.5%
GraphQL-Cosmos WiFi (ms)	4692	4590	+ 2 %
GraphQL-REST 4G (ms)	4852	914	+ 83 %
GraphQL-Elastic 4G (ms)	603	192	+ 69 %
GraphQL-Cosmos 4G (ms)	33207	36585	- 10%

Server and network	**REST WiFi**	**REST 4G**	**Efficiency change**
Response times (ms)	1795	1746	+ 3 %

7 Results

7.1 Research Question 1

"*How does GraphQL perform in terms of response times, throughput and payload size in relation to REST in a mobile and desktop setting? - Are there any notable differences as payload size reduces?"*

Hypothesis

GraphQL will outperform REST in certain cases when operating on a mobile network as payload size decreases.

7.1.1 Networks & Insights

This study set out to look into two main questions, one being the performance differences between GraphQL and REST implementations on a regular 2.4Ghz WiFi network vs. a 4G mobile network. Are there any major differences to be seen in the two different network environments in terms of response times and throughput as payload size changes?

As seen in the results of the different network environments, the WiFi network has shown to be the most stable and reliable option. It has also been observed that payload size does not seem to have had a big impact on response times on the WiFi network, which potentially could change as user load becomes even higher. On the other hand, 4G has been shown to carry some expected volatility due to the nature of the type of network. But it has also shown potential signs of benefiting to a much greater extent as the payload decreases compared to the WiFi network where only minor changes were observed. There is also the aspect of download speed; while WiFi has an average measured speed of 116 mb/s and 4G has 40 mb/s in the test environment, one has to consider that there is a potential hidden pattern here that has been outside the scope of this paper – namely, the comparison of WiFi network vs. mobile networks and their data-fetching performance as the download speeds are synchronized to the same limit.

The most important insights of this paper in regards to differences and characteristics of the two different network types as data fetching occurs, is that over both WiFi and 4G networks, a traditional REST implementation seems to be providing reasonable response times and reliable results across these two types of network. It is a well-proven technique with world wide adoption and support to be found all over the Internet. At the same time, in the 4G network, GraphQL has shown to speed up response times substantially as payload was able to shrink.

Perhaps, if the user segment is mostly using mobile networks, in such a case the decision could be the difference between fast or slow response times and experiences for the end user.

7.1.2 Conclusion: Research Question 1

There are differences in the two network types during load testing with requests to 4 different server implementations. The WiFi network has shown to be stable, and to have faster response times than the 4G network, but since the network speeds are so different, the results of the throughput for each setting matters more since it is a measurement of efficiency.

WiFi is on average almost 1.75x more efficient in terms of transactions per second than the 4G network, meaning it takes on average less time to execute the test on WiFi settings. This is true for all different implementations in this study except the GraphQL-Elastic version, which has proven to be stable in execution times and throughput across both payload size and network type.

Lastly, the 4G network has shown signs of being a greater beneficiary of smaller payloads and this is one of the reasons that GraphQL was developed in the first place. The efficiency rate changed by 69-83% for GraphQL-REST and GraphQL-Elastic servers respectively, but it also went down by 10% for GraphQL-Cosmos. Again, this indicates that this server implementation type either is more heavy to process, or has been set up in a way that doesn't provide the environment which will allow for efficient data transactions.

7.2 Research Question 2

"*In what setting is GraphQL most effective? In what type of environment do we see the best performance from a GraphQL server fetching data from services like Elasticsearch, Cosmos DB and REST?*"

The second question is whether there is a type of GraphQL implementation that can consistently outperform REST. To answer this question, extensive testing and analyzing has been done on three setups of a GraphQL server. By gathering information regarding test execution time, response time and the changes between the two different network types (WiFi vs. 4G), the results show that there are major differences in performance depending on what type of set up an organization needs for its GraphQL purposes.

7.2.1 GraphQL-REST

This type of implementation has shown to increase rapidly in response times as the usage goes up. This is true for both network environments – it has managed to deliver test execution times of as low as 5.28 seconds observed on the WiFi network, and as high as 27 seconds on the 4G network. This type of implementation has throughout the testing been competing with the traditional REST server in terms of throughput and response times, the average response time for this server being 1517 ms for user load and 1434 ms for over-fetching, whereas the traditional REST server had 1795 ms in average response times. Another interesting thing to look at is that this type of server

has consistently had much faster test execution times than the traditional REST implementation which averages at about 20 seconds per test.

Summarized in Table 8 below are the results for the 100 requests case as this has proven to be a scenario that all four different server versions could handle the requests within the given time frames on the hardware available for test execution.

Table 8: Response and Test Times: 100 Users / GraphQL-REST

	Response time (WiFi)	**Response time (4G)**	**Test time (WiFi)**	**Test time (4G)**
User load: 100 Users	381 ms	6759 ms	5.28 s	5.26 s
Over-fetching: 100 Users	483 ms	627 ms	5.28 s	5.46 s

7.2.2 GraphQL-Elastic

This type of GraphQL server has consistently throughout the tests shown to be stable, efficient and lightweight. As the other versions showed struggle with scaling up, the GraphQL-Elastic server delivered test times of 4.9 seconds on average. Even as user load was 500 requests in 5 seconds, Elasticsearch has been able to process all requests within the given time frame, and has had the best and most consistent response times by far. Especially stable observations have been made on the WiFi network, and even though response times went up on 4G, the test time remained under 5 seconds, indicating that all the requests were being handled within the test time limit. Furthermore, in terms of response time changes as payloads decreased, the 4G network showed a clear reduction in the response times, which could suggest that the 4G network does benefit more from smaller payloads than the WiFi network does.

Table 9: Response and Test Times: 100 Users / GraphQL-Elastic

	Response time (WiFi)	**Response time (4G)**	**Test time (WiFi)**	**Test time (4G)**
User load: 100 Users	67.73 ms	157 ms	4.92 s	4.95 s
Over-fetching: 100 Users	67.14 ms	134.44 ms	4.91 s	4.95 s

One of Elastic's strengths is that it is defined as a search engine, even though it is by default a document-oriented database. It provides very fast search functionality and when configured so that the GraphQL server only needs to query one document and then let Elastic filter in the queries, Elastic's strengths get utilized together with GraphQL's capacity to give the requester a dynamic and easy way to fetch data.

7.2.3 GraphQL-Cosmos

The GraphQL-Cosmos server has consistently proven to be heavy to process. It has had some very high response times and lengthy test times – some of which going longer than 2 minutes – indicating that there is a bottleneck somewhere as the test was getting completed, making it take a very long time in comparison to the other server implementations. Even when the request rate was 100 in 5 seconds, the GraphQL-Cosmos server struggled to keep test times down.

Continuously throughout each user load and network setting, GraphQL-Cosmos has been the slowest across the board. This could be due to the possibility that querying Cosmos DB is a heavier process. This study has not looked into processing needs on the source side of computation, so no conclusion on that matter can be drawn. It could also be due to bad connectivity to Azure, as this implementation is dependent on Azure services to be up, running and efficient.

Table 10: Response and Test Times: 100 Users / GraphQL-Cosmos

	Response time (WiFi)	**Response time (4G)**	**Test time (WiFi)**	**Test time (4G)**
User load: 100 Users	664 ms	4022 ms	6.47 s	48.94 s
Over-fetching: 100 Users	457 ms	5334 ms	5.35 s	15.13 s

7.2.4 REST

The traditional REST implementation has during these tests shown to be reliable in terms of response times for both WiFi and 4G networks. When looking at its performance in a stable environment (all server types managed the user load volume), for the 100 requests case, their response times at around 207 ms for WiFi and 404 ms for 4G, meaning the test time takes longer than both GraphQL-Elastic and GraphQL-REST servers. This seems to indicate that the GraphQL Apollo Server is more efficient per second than the REST server implemented in C# (dotNet).

Table 11: Response and Test Times: 100 Users / REST

	Response time (WiFi)	**Response time (4G)**	**Test time (WiFi)**	**Test time (4G)**
User load: 100 Users	207 ms	404 ms	16.36 s	15.07 s

7.2.5 Conclusion: Research Question 2

"In what setting is GraphQL most effective? In what type of environment do we see the best performance from the GraphQL server fetching data from a service like Elasticsearch, Cosmos DB or REST API endpoints?"

In the observations made during the tests, it is clear that GraphQL in combination with Elasticsearch is the most stable, reliable and efficient GraphQL implementation among all three different versions tested in this paper. GraphQL-Elastic has had response times many times lower than the other server types.

The GraphQL-Elastic server has also been able to deliver efficiency despite scaling up, completing its test runs within the 5 second test time. None of the other implementations have been able to deliver response times at around 70 ms and also execute to completion in time.

The GraphQL-Elastic server has in the 100-request case shown:

- 82% faster response times than GraphQL-REST on WiFi
- 86% faster response times thanGraphQL-REST on 4G

- 90% faster response times than GraphQL-Cosmos on WiFi
- 85% faster response times than GraphQL-Cosmos on 4G

The answer to Research Question 2 therefore becomes; GraphQL has shown to be working better with certain data source services compared to others. The results show that the most efficient and reliable data source of the three proposed in this paper is Elasticsearch which has shown drastic performance differences that other server implementations just cannot compare to.

8 Discussion

This has been a challenging and rewarding project to work on. It has given me many insights regarding the different paradigms of data fetching, how it works, why it is important and many associated challenges. It has also been a great opportunity to increase my knowledge of server development. It has deepened my understanding of the importance of knowing all the independent and dependent variables. I found myself at many occasions, thinking I had the data to write the analysis but I kept noticing issues in the results indicating, for example, hardware limitations.

Initially, the test design was made to run at 500 users as the lower limit and then 1500 followed by 2500 users in a 5-second period. I started seeing very volatile data that was inconsistent and untrustworthy. I saw that the higher the volume of requests went, the more failed requests were returned with error codes indicating that it was unable to contact the localhost:3000/api route to fetch data, meaning that the server had most likely crashed due to resource exhaustion. In other words, a DDOS attack due to too many requests and incapacity to handle the sheer volume in an orderly fashion.

I started reducing the volume of requests in the test scenarios and found that the levels of 100, 300 and 500 requests all successfully handled the request without any type of failing on a consistent basis.

It has been fun and challenging to drive this project from start to finish, having to plan and execute all parts, develop the servers, design the experiment, and ensure the results were comparable by eliminating independent variables.

8.1 Method

To work with an experiment as a method was an obvious way to go due to its nature regarding formulating a hypothesis and then designing a test to prove that hypothesis, or disprove it. but also due to the nature of the research questions which both were formulated in comparative ways. It has worked very well and the course book “Researching Information Systems and Computing” (Oates, Griffiths and McLean, 2022), has been a great guide to perform the experiment and what to consider.

One thing that has been challenging is to make sure that the tests are designed to generate as relevant results as possible, this meant that I had to run the tests over and over again depending on realizations made during attempts to analyze the data. For example I had to run the test several times, trying to find a level of acceptable scaling to avoid stalling or crashing servers due to too many concurrent users. When compared to for example a qualitative approach, I´m not sure that it would have been that easy to redo the interviews if needed. To me it highlights the importance of pre-analysis and preparations such as knowing your domain and restrictions before running tests. At the same time, a strength of this type of research is that I could rerun the tests over and over again as I tweaked the settings of the tests.

It was a challenge to identify and exclude the independent variables as they seemed to appear in new shapes as tests were being executed. As an example I didn't realize that the traditional REST server was hosted in the cloud until i´d already run tests, this meant that among other things, hardware running the software was an independent variable. To solve this issue, instead, I hosted the REST server on the same laptop that the GraphQL server was on, so that all server implementations were being run on the exact same type of hardware.

A huge take away from this study has been the weight of doing a proper analysis prior to execution and data collection. I could´ve probably saved a lot of time if i didn't have to rerun my tests so many times, so isolating and acting against the independent variables as early as possible in the process, this could have opened up possibilities for even more relevant results, when the obvious issues are removed early, the more complex and perhaps more finetune-oriented parameters could´ve been in focus during the test execution. It definitely taught me a lot about how important it is to try to reduce the independent variables to only one, because the outcome of the test varied quite a bit as did new tests after eliminating newly discovered independent variables. These types of issues will be easier to see and act upon when my personal experience grows within the field of research.

8.1.1 Threats to validity

The tests described in this paper have been run locally on a ThinkPad T14s. It was necessary to scale down request volumes to be able to run stable tests on just a laptop. So there is possibly an issue when scaling up to several thousands of concurrent users.

The test environment was fetching data from non-cached data. For example, a service like cloudflare can protect a site by acting as an exterior barrier for requests of all sorts aiming for a server, and these services can also cache data. Thus, if User B fetches the same data as User A did earlier, that data would be cached and made more readily available for User B who wouldn’t even need to reach the organizations servers at all since they were serviced by the exterior cache system. The tests in this paper have all been run via non-cached sources, and since cache is the norm, it might have an effect on the results once cache is implemented.

8.1.2 Reliability

It has been tedious to eliminate as many independent variables as possible to get reliable results, and the results do show some volatile behavior with some unexplainable spikes, but those spikes also repeated themselves throughout testing, so they were not based on single one-run tests, but rather the tests were run many many times and then averages were calculated from the results.

The results of this paper should perhaps be seen as an indication to the fact that there are potentially big differences in performance regarding both server setup and network type. Organizations should consider these results as indicators that it is important to

look at their requirements, examine their customer segments, determine what devices their customers are on and what network types they typically use. These are important considerations that should be weighed in the decision-making process in the contexts of efficiency, scaling capacities and stability of the type of server needed.

8.2 Scientific Aspects

8.2.1 Previous studies

Earlier studies in the field have shown results that are very similar between GraphQL and REST. They have not been transparent regarding things such as network and data sources, it has been difficult to claim a clear winner between the two technologies and even if it wasn't, since the transparency regarding independent variables are so unsure, it would´ve been hard to definitely claim that one technology is superior to another.

I´ve tried to take a more open approach and have identified different data sources, and network capacities and I believe that this will make the results more relatable and applicable for decision makers and researchers across the board. In the results I´ve shown that it matters greatly what type of data source and setting you use for your server. Depending on the requirements of your organization these insights can have a great impact on what you decide to spend resources on.

8.2.2 Future studies

Future studies carried out in this area could look into how processing power on the data source side is affected by the different types of server setups. Is efficiency directly connected to CPU usage and, therefore, high energy consumption? What is the best option in terms of efficiency, reliability and energy? This would be interesting to learn more about as these aspects are deeply connected to relevant societal issues such as green energy consumption and sustainability.

Furthermore, to enhance the results of this paper, future studies should aim to take place in a hardware and software environment that utilizes the type of power that a typical data-driven organization does. This means to make a market analysis and gain understanding of what is the quintessential information system architecture, and how different types of data fetching technologies and designs can affect these systems. There are certainly different optimal setups for different systems architectures, therefore the results would probably be of great importance for organizations aiming to build anew or reconstruct their approach to data and digital interconnectivity.

Future studies are encouraged to look at these results and weigh in the sizes and complexities of the process for the server to receive a request from a client, or tools such as jmeter, and see how heavy these requests are to compute. Is it possible to measure the processing power of computing a single typical request through the whole chain of

affected services and then put it into relation to the efficiency? If organizations had knowledge about an, as exact number as possible in terms of cost and effect, the right decisions could be made early and resources be used in the most effective way. It will not always be the fastest system that is the obvious choice, requirements of the users has to be in focus here, meaning that if you do not have the need for superfast blink of an eye response times, then choosing a system that perhaps demand less resources and still fulfills your needs can be made easier. If future studies can create a clear map over these issues, data driven organizations can reduce their environmental footprint globally.

If you are a decision maker and are presented with the possibility of choosing which way to approach data fetching, you would have to analyze the requirements of the system at hand. In case the needs include fast response times and seamless fetching of data with reliable response times and capacities, an Elasticsearch implementation in combination with GraphQL holds great promise. Yet, it might take time to implement properly if the right technical and human resources are not available. On the other hand, if there are no demands for exceptionally fast response times and reduced payload size, then a regular, traditional REST implementation will do just fine, especially considering the vast adoption of REST within the developing community. With these things in mind, future studies could look into the time and resources needed to implement the different types of servers by developers of different levels of knowledge, this could help organizations to assess even further what technology to choose based on their inhouse human resources available and capacity to implement new software.

8.3 Societal Aspects

The majority of society is in some way dependent on data traffic and we use the Internet almost, if not every day. There are countless units consuming data being exposed either publicly and/or internally. Either way it is important that there is appropriate knowledge available for decision makers to make those calls based on data applicable to their organizational needs and goals. If the requirements do not include super fast data delivery speed, then the GraphQL-Elastic server might not be necessary.

On the other hand, if there are requirements of lightning speed data delivery, then it could be beneficial for decision makers to know that not only is GraphQL-Elastic fast, but it also appears to consume fewer resources. In the long run, this could mean reduced needs for hardware and less cost on the capacity investment department. This paper can at least shed some light on the fact that there is more to the choice of server implementation than perhaps once thought, so that organizations can build more efficient and less resource-heavy solutions in the future.

Society as a whole will benefit from organizations broadly adapting technology that is both reliable and efficient. As it is at the core of many daily functions in our lives, it is very important to be able to make well waged decisions regarding these matters. This study shows that there are potentially big differences in the efficiency between the various server types, does this also manifest in CPU usage for both server and

datasource computation? If so, then it is evident that the decisions made regarding systems architecture can and most likely do have a big impact on global issues such as the climate question which is very relevant and important for many in society today.

When thinking about it some more, it could even turn out to be a win-win situation where organizations potentially can reduce their energy consumption and cost and at the same time, their environmental footprint and therefore benefiting society by reducing their needs for energy. So by empowering people with more knowledge regarding what systems they use that consume the most energy, perhaps people can put pressure on these organizations to do everything in their might to reduce that dependency. Society as a whole benefits from sustainable solutions being broadly implemented and all of us in the sector of software development ought to keep this aspect at the core of our approach and design for the future of digitalized society.

I think the results in this study shows, in contrast to previous studies, that architecture can affect the outcome of an experimental data driven comparative test, and that future studies must get even better at being transparent regarding the exact test environment utilized, only then the results can be properly applicable and generalized.

8.4 Ethical Aspects

8.4.1 GDPR

Today, whenever the discussions end up in data and ethical aspects, GDPR is looming in the corners. Personal data and its integrity is a central part in development and business today. The old days of zero ethics in regard to data collection and exploitation is seeing its final sunset. Developers, system architects, business and many many more industry players have to consider usage and storage of data connected to an individual that is protected under GDPR.

In terms of this paper, it makes sense to use GraphQL from a protection point of view because one can explicitly define which data is needed. In this way, the client never sees the unneeded and potentially legally-protected personal data that was over-fetched. From an ethical perspective, GraphQL could potentially be a powerful tool in filtering out data not needed to be exposed, which not only observes the privacy rights of citizens, but also has the added benefit of reducing regulatory burdens and liability concerns for the organization. GDPR is something that affects developers and organizations on a daily basis, so any means that can make these issues easier to navigate has to be seen as a benefit to society as it can create better ways to stand up for proper ethical use of data.

Moreover, the enforcement of GDPR highlights the importance of data minimization, transparency, and user consent. GraphQL's ability to limit data fetching to only what is necessary aligns perfectly with these principles. This selective data fetching can potentially help mitigate risks associated with data breaches and misuse of personal information. Organizations adopting GraphQL can perhaps more easily comply with

GDPR rules, thereby enhancing their trustworthiness in the eyes of users and regulations. Efficiency and expedited response times do not come for free. It takes resources to generate power, be it to send a rocket into orbit or to make super-fast computations. It all takes resources of one nature or another. In this case, there is a lot of computation happening all over the globe today, and it consumes immense volumes of energy.

8.4.2 Energy Consumption

We are fully dependent on the internet and the fast communication it brings us. Therefore, if we can't live without it, we must ensure we build sustainable options for the market to adopt. Data fetching, writing, and reading data are core functionalities of digital interconnectivity. If we can build efficient solutions based on actual needs and not "nice to haves," and then utilize proper software and hardware to power those needs, we can succeed in creating a more efficient digital world that consumes fewer resources and empowers the continued green revolution for other sectors as well. This is done by making the right decisions in terms of productivity, efficiency, and resource consumption. Addressing the ethical concerns of energy consumption, it's crucial to recognize the environmental impact of our technological choices. Data centers, which power the internet and digital services, consume huge amounts of electricity. By optimizing data fetching techniques, such as utilizing GraphQL to minimize unnecessary data transfer, we can potentially reduce the energy footprint of our digital operations. This can not only lower operational costs but also contribute to global efforts in reducing the environmental footprint.

Furthermore, the push for greener technologies in data handling and server operations is becoming increasingly relevant. Sustainable software development practices, such as optimizing code for energy efficiency, using renewable energy sources for data centers, and implementing energy-efficient hardware, are critical steps toward a more sustainable digital future. Organizations that prioritize these practices will not only comply with emerging environmental regulations but also put themselves in a good position in terms of competing for the ever increasing amount of environmentally enlightened users and consumers.

In conclusion, the integration of ethical considerations such as GDPR compliance and energy efficiency into data fetching paradigms like GraphQL and REST is essential for developing sustainable and responsible digital solutions. These aspects ensure that the technology we rely on daily is both respectful of user privacy and mindful of its environmental impact. By addressing these ethical perspectives, we can create a digital future that is both fast, reliable and sustainable.

References

Autili, M., Filippone, G. and Tivoli, M. (2023). Migrating from Monoliths to Microservices: Enforcing Correct Coordination. Proceedings of the 2023 ASEW Conference. doi: https://doi.org/10.1109/asew60602.2023.00020.

Barbaglia, G., Murzilli, S. and Cudini, S. (2016). Definition of REST web services with JSON schema. Software: Practice and Experience, 47(6), pp. 907–920. doi: https://doi.org/10.1002/spe.2466.

Brito, G., Mombach, T. and Valente, M.T. (2019). Migrating to GraphQL: A Practical Assessment. Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER). Hangzhou, China: IEEE, pp. 140-149.

Brito, G. and Valente, M.T. (2020). REST vs GraphQL: A Controlled Experiment. IEEE Xplore. doi: https://doi.org/10.1109/ICSA47634.2020.00016.

Goda, K. and Kitsuregawa, M. (2012). The History of Storage Systems. Proceedings of the IEEE, 100(Special Centennial Issue), pp. 1433–1440. doi: https://doi.org/10.1109/jproc.2012.2189787.

Hartina, D.A., Lawi, A. and Panggabean, B.L.E. (2018). Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University. In: The 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT). IEEE, pp. 237.

Marchuk, Y., Dyyak, I. and Makar, I. (2023). Performance Analysis of Database Access: Comparison of Direct Connection, ORM, REST API and GraphQL Approaches. doi: https://doi.org/10.1109/elit61488.2023.10310748.

Mosleh, M., Dalili, K. and Heydari, B. (2018). Distributed or Monolithic? A Computational Architecture Decision Framework. IEEE Systems Journal, 12(1), pp. 125–136. doi: https://doi.org/10.1109/jsyst.2016.2594290.

Nawaz, M.S., Khan, S.U.R., Hussain, S. and Iqbal, J. (2022). A study on application programming interface recommendation: state-of-the-art techniques, challenges and future directions. Library Hi Tech. doi: https://doi.org/10.1108/lht-02-2022-0103.

Oates, B.J., Griffiths, M. and McLean, R. (2022). Researching Information Systems and Computing. SAGE.

Rodriguez, A. (2008). RESTful Web services: The basics. IBM. [online] Available at: https://cs.calvin.edu/courses/cs/262/kvlinden/references/rodriguez-restfulWS.pdf [Accessed 12 February 2024].

Roksela, P., Konieczny, M. and Zielinski, S. (2020). Evaluating execution strategies of GraphQL queries. Proceedings of the 2020 IEEE 13th International Conference on Electronics and Information Technologies (ELIT), pp. 640-644. Available at: https://ieeexplore.ieee.org/document/9188361.

Rui André Oliveira, M., Martínez Raga, M., Laranjeiro, N. and Vieira, M. (2020). An approach for benchmarking the security of web service frameworks. Future Generation Computer Systems, 110, pp. 833–848. doi: https://doi.org/10.1016/j.future.2019.10.027.

Vadlamani, S.L., Emdon, B., Arts, J. and Baysal, O. (2021). Can GraphQL Replace REST? A Study of Their Efficiency and Viability. 2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP). doi: https://doi.org/10.1109/ser-ip52554.2021.00009.

Vohra, N. and Kerthyayana Manuaba, I.B. (2022). Implementation of REST API vs GraphQL in Microservice Architecture. 2022 International Conference on Information Management and Technology (ICIMTech). doi: https://doi.org/10.1109/icimtech55957.2022.9915098.

Wennerström, T., Jespersen, J. and Lundquist, M. (2002). Simple Object Access Protocol: A basic overview Distributed Systems Fall 2002. [online] Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5ff665a3254400cbb6330d0ef93253d41d7bdf80.

Xu, A. (2023). The Evolving Landscape of API Protocols in 2023. Postman Blog. [online] Available at: https://blog.postman.com/api-protocols-in-2023/ [Accessed 1 May 2024].

Appendix

The Queries have been washed and fields have been replaced with car mock names, to protect the data of the organization involved in the test cases. These queries fetch data from different parts of the data preparation stage, this is why they look slightly different in their structure, but they do deliver the aligned payload sizes at about 13600 bytes. These queries are included for others to replicate the results and see on what grounds these tests have been run.

GraphQL - Cosmos DB Query

```
query CarConfigDetails($carVariantId: ID!) {
 CarConfigDetails(carVariantId: $carVariantId) {
        vehicleId
        configurationCode
        modelName
        variantId: $carVariantId
        articleId
        engineCode
        productCode
        vehicleInformation {
        isBundle
        approvals
        attributes {
        variantDescription
        colorOptions
        originCountry
        wheelDesign
        materialDetails
        bodyType
        seriesCollection
        upholsteryType
```

vehicleShape

manufacturer

finishMaterial

removableCover

durabilityRating

}

}

configOptions {

description

features

availableFrom

optionId

}

resourceLinks {

files {

description

fileType

title

suggestions {

input}}}

variantId

category

dealerships {

address

dealershipName

id

nearbyDealerships

city

```
        country}
    }
}
```

GraphQL-REST Query

```
query CarConfigSwagger($carVariantId: ID!) {
 CarConfigSwagger(carVariantId: $carVariantId) {
vehicleId
articleId
groupId
modelName
series
configCode
articleId
salesType
assemblyCostGroup
description {
brief
}
webpage
classification {
typeCode
typeName
categoryLeafId
categoryLeafName
googleCategory
showQuantityDropdown
categoryHierarchy
}
details {
variantDescription
```

colorOptions

originCountry

wheelDesign

fadeResistance

materialDetails

materialType

seriesCollection

upholsteryType

shape

vehicleShape

manufacturer

finishMaterial

hasRecliner

removableCover

leftChaiseLounge

leftDivan

chaiseLoungeDivanCombo

durabilityRating

alternativeDepth

depth

height

adjustableHeight

adjustableSeatingHeight

armrestHeight

armrestWidth

vehicleWidth

legDiameter

legHeight

```
seatingWidth
seatingHeight
seatingDepth
weight
width
seatCount
}
pricing {
currentPrice
originalPrice
}
imageUrls
inventory {
status
storeId
}
dealerships {
city
country
showStockOnline
dealershipName
closed
warehouse
deliveringWarehouse
driveThru
email
keywords
}
```

```
        colorScale
        navigationPath {
        name
        path
        }
  }
}
```

GraphQL-Elastic Query

```
query CarConfigElastic($carVariantId: ID!) {
 CarConfigElastic(carVariantId: $carVariantId) {
vehicleId
configCode
modelName
variantId: $carVariantId
colorOptions
engineCode
configCode
details {
variantDescription
colorOptions
originCountry
wheelDesign
materialDetails
materialType
seriesCollection
upholsteryType
vehicleShape
manufacturer
finishMaterial
removableCover
durabilityRating
depth
adjustableHeight
adjustableSeatingHeight
armrestHeight
```

```
        armrestWidth
        vehicleWidth
        legHeight
        seatingWidth
        }
        variantId
        description
        accessories {
        ids
        types}
        classification {
        categoryLeafId
        categoryLeafName
        googleCategory
        showQuantityDropdown
        categoryHierarchy
        }
        completeImageCollection
        inventory {
        stockId
        status
        deliveryDays
        deliveryWorkingDays
        stockStatus
        medianDeliveryDays
        }
 }
}
```