BINDURA UNIVERSITY OF SCIENCE EDUCATION

FACULTY OF SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE



An Experimental Performance Benchmarking of REST and GraphQL APIs Under Controlled Workload Conditions

By

SIREWU ANDY TAFADZWA B220700B

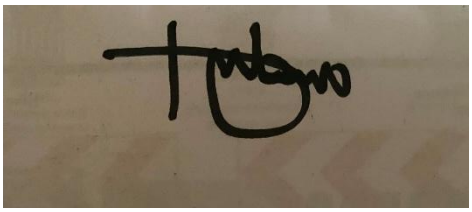
SUPERVISOR: Mr. MHLANGANISO

A RESEARCH PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE BACHELOR OF SCIENCE HONOURS DEGREE IN SOFTWARE ENGINEERING

APPROVAL FORM

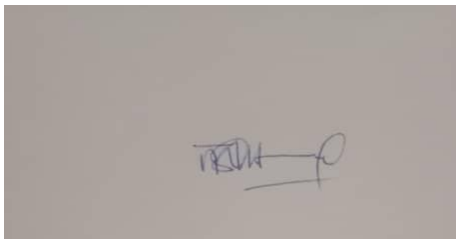
The undersigned certify that they supervised B220700B SIREWU ANDY TAFADZWA dissertation entitled "An Experimental Performance Benchmarking of REST and GraphQL APIs Under Controlled Workload Conditions" submitted in partial fulfillment of the requirement for Bachelor of Science Honors' Degree in Software Engineering.

Student……...



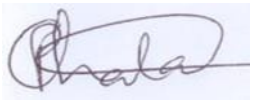
Date……

Supervisor……..



Date……

Chairperson……





Date……..

DEDICATION

This work is dedicated to my parents and family, whose unwavering support and belief in me have been the bedrock of my academic journey. It is often the encouragement of loved ones that guides us toward cultivating success and resilience in our endeavors. I also dedicate this to the mentors and educators who have instilled in me a sense of discipline and the drive to keep moving forward. To the next generation of engineers, I share the wisdom that change and security only come to those who are not afraid to make a move to improve the world around them.

ACKNOWLEDGEMENT

My academic journey contained several individuals who helped pave the road to success. First and foremost, I would like to thank my supervisor Mr. MHLANGANISO for his mentorship throughout the research phase and the completion of this process. His exceptional feedback and guidance were instrumental in shaping this study, mirroring the gratitude expressed by researchers for their own praxis directors. I am profoundly appreciative of the faculty at Bindura University of Science and Engineering, whose instruction and real-world experience made this challenging program an incredible learning experience. I also wish to express gratitude to my family and friends, who have encouraged me by demonstrating exemplary patience and support as I embarked on this educational endeavor. Their unwavering presence has been my biggest cheerleader throughout this process. Finally, I am grateful for the strength and guidance provided to me during this life, allowing me to plan my work and see it through to completion.

ABSTRACT

This research evaluates the performance trade-offs between REST and GraphQL APIs under controlled workloads up to 1000 concurrent Virtual Users (VUs). While GraphQL's client-side declarative fetching successfully solves network "over-fetching" and "under-fetching" by delivering a 52.6% reduction in network payload sizes (27.0 MB vs. 57.0 MB in REST), it introduces a significant server-side processing penalty. Containerized benchmarking experiments using the k6 engine on fixed, non-elastic Ubuntu infrastructure revealed that GraphQL incurs a 170.4% increase in average response latency (31.26 ms) compared to REST (11.56 ms). Furthermore, under peak saturation, GraphQL's tail latency degraded sharply to a p95 of 126.96 ms compared to REST's highly stable 82.31 ms, indicating performance instability caused by runtime Abstract Syntax Tree (AST) parsing and resolver tree execution. Therefore, REST remains the superior choice for high-speed enterprise systems requiring low CPU overhead and predictable tail latencies. Conversely, GraphQL is strongly recommended for bandwidth-constrained, resource expensive environments such as remote edge installations or cellular IoT networks where minimizing data transmission volume outweighs localized server processing penalties.

Contents

APPROVAL FORM i
DEDICATION ii
ACKNOWLEDGEMENT iii
ABSTRACT iv
LIST OF FIGURES ix
LIST OF TABLES x
LIST OF ACRONYMS xi
CHAPTER 1 1
PROBLEM IDENTIFICATION 1
1.1 Introduction 1
1.2 Background of the Study 1
1.3 Statement of the Problem 2
1.4 Research Objectives 3
1.5 Research Questions 3
1.6 Research Hypothesis 3
1.7 Justification / Significance of the Study 3
1.8 Assumptions 4
1.9 Limitations / Challenges 4
1.10 Scope / Delimitations of the Research 4
1.11 Definition of Terms 5
CHAPTER 2 8
LITERATURE REVIEW 8
2.1 Introduction 8
2.2 Conceptual Review 8
2.2.1 Application Programming Interfaces 8
2.3 REST Architecture 9
2.4 GraphQL Architecture 10
2.5 Comparative Analysis of REST and GraphQL 12
2.6 API Performance Evaluation Metrics 13
2.6.1 Response Time 13
2.6.2 Throughput 13

2.6.3 Latency 13
2.6.4 CPU Utilization 13
2.6.5 Memory Utilization 13
2.6.6 Error Rate 13
2.7 Performance Benchmarking Techniques 14
2.7.1 Load Testing 14
2.7.2 Stress Testing 14
2.7.3 Scalability Testing 14
2.7.4 Container-Based Benchmarking 14
2.8 Empirical Review of Related Studies 14
2.9 Theoretical Framework 16
2.10 Research Gap 16
2.11 Summary 17
CHAPTER 3 18
RESEARCH METHODOLOGY 18
3.1 Introduction 18
3.2 Research Design 18
3.3 Population and Sample 18
3.3.1 Target Population 18
3.3.2 Study Sample 19
3.3.3 Sampling Technique 19
3.3 Research Instruments and Environment 19
3.3.1 Hardware Specifications 19
3.3.2 Software Configuration 20
3.4 Operational System Design and UML Architecture 20
3.5 Experimental Parameters and Data Collection Procedures 24
3.5.1 Workload Profile and Ramping Strategy 24
3.5.2 Execution Protocol and Data Capture 24
3.6 Data Analysis Procedures 27
3.7 Reliability and Validity 27
3.8 Ethical Considerations 27
3.9 Summary 27

CHAPTER 4 28
DATA PRESENTATION, ANALYSIS AND INTERPRETATION 28
4.1 Introduction 28
4.2 Presentation of Results 28
4.3 Analysis of Network Efficiency 30
4.4 Analysis of Response Latency and Stability 31
4.5 Scalability and Hardware Stress Evaluation 32
4.6 Synthesis of Findings 32
4.7 Summary 33
CHAPTER 5 34
CONCLUSIONS AND RECOMMENDATIONS 34
5.1 Introduction 34
5.2 Summary of the Study 34
5.3 Major Conclusions Drawn from the Study 34
5.3.1 GraphQL Provides Superior Network Efficiency 34
5.3.2 REST Maintains Lower Computational Overhead 35
5.3.3 REST Exhibits More Stable Performance Under Extreme Concurrency 35
5.3.4 Both Architectures Demonstrated High Reliability 35
5.3.5 No Single Architecture Is Universally Superior 35
5.4 Recommendations 36
5.4.1 Recommendations for Software Engineers and System Architects 36
5.4.2 Recommendation for Mobile and Distributed Systems 36
5.4.3 Recommendation for High-Performance Enterprise Systems 36
5.4.4 Recommendation for Infrastructure Isolation 36
5.5 Recommendations for Future Research 36
5.6 Summary 37
APPENDIX A: REST API k6 BENCHMARKING SCRIPT 38
A.1 Overview 38
A.2 Script Source Code (test-rest.js) 38
APPENDIX B: GRAPHQL API k6 BENCHMARKING SCRIPT 40
B.1 Overview 40
B.2 Script Source Code (test-graphql.js) 40

APPENDIX C: DOCKER CONTAINER ORCHESTRATION CONFIGURATION 42
C.1 Overview 42
C.2 Orchestration Script Configuration (docker-compose.yml) 42
APPENDIX D: RAW k6 BENCHMARK OUTPUT LOGS 44
D.1 Overview 44
D.2 REST Terminal Output Log Summary 44
D.3 GraphQL Terminal Output Log Summary 45
APPENDIX E: EXPERIMENTAL HARDWARE SPECIFICATIONS 46
E.1 Configuration Environment Parameter Checklist 46
APPENDIX F: ADDITIONAL GRAPHS AND VISUALIZATION OUTPUTS 47
F.1 Overview 47
F.2 Scalability and Throughput Curves 47
F.3 Latency Variance and Tail Saturation Distributions 47
F.4 Network Bandwidth Conservation Progressions 48
APPENDIX G: UML SYSTEM DESIGN DIAGRAMS 50
G.1 UML Component Architecture 50
G.2 UML Behavioral Sequence Process 50
References 51

LIST OF FIGURES

Figure 2.2……9
Figure 2.3……10
Figure 2.6……14
Figure 3.4.1……20
Figure 3.4.2……21
Figure 3.4.3……22
Figure 3.5.2……24
Figure 3.5.3……25
Figure 4.2.1……28
Figure 4.2.2……28
Figure 4.3……29
Figure 4.4……30
Figure 4.5……31
Figure D.2……43
Figure D.3……44
Figure F.1……46
Figure F.2……47
Figure F.3……48
Figure G.1……49
Figure G.2……49

LIST OF TABLES

Table 2.6.1……12
Table 2.6.2……13
Table 3.3.1……18
Table 3.3.2……19
Table 3.5……15
Table 4.2……27
Table 4.3……29
Table E.1……45

.

LIST OF ACRONYMS

- API – Application Programming Interface
- APQ – Automated Persisted Query
- AST – Abstract Syntax Tree
- BASE – Basically Available, Soft State, Eventual Consistency
- CPU – Central Processing Unit
- CRUD – Create, Read, Update, Delete
- CSV – Comma-Separated Values
- DDR – Double Data Rate
- DOS – Denial of Service
- DV – Dependent Variable
- GQL – GraphQL
- GUI – Graphical User Interface
- HTTP – Hypertext Transfer Protocol
- I/O – Input / Output
- IV – Independent Variable
- IoT – Internet of Things
- IIoT – Industrial Internet of Things
- JSON – JavaScript Object Notation
- LTS – Long-Term Support
- MB – Megabyte
- Mhz – Megahertz
- MTU – Maximum Transmission Unit
- OS – Operating System
- PDR – Packet Delivery Ratio
- RAM – Random Access Memory
- REST – Representational State Transfer
- RPC – Remote Procedure Call
- RPL – Routing Protocol for Low-Power and Lossy Networks
- RTT – Round-Trip Time
- SATA – Serial Advanced Technology Attachment
- SLA – Service Level Agreement
- SSD – Solid State Drive
- SOA – Service-Oriented Architecture
- SOAP – Simple Object Access Protocol
- TCP/IP – Transmission Control Protocol / Internet Protocol
- TPS – Transactions Per Second
- UML – Unified Modeling Language

- URI – Uniform Resource Identifier
- URL – Uniform Resource Locator
- VU – Virtual User
- XML – Extensible Markup Language

CHAPTER 1
PROBLEM IDENTIFICATION

1.1 Introduction

In contemporary enterprise software engineering, application programming interfaces (APIs) serve as the fundamental communication backbone for distributed systems, microservices and real time computing platforms (Sahin et al, 2025). As systems expand to support increasingly massive numbers of concurrent transactions, the data transport layers encounter severe constraints regarding network-channel bandwidth preservation and server-side hardware execution efficiency (Gomes et al, 2026). For over two decades, Representational State Transfer (REST) has stood as the industry standard, utilizing a resource-oriented architecture anchored to the native stateless methods of the Hypertext Transfer Protocol (HTTP). However, the modern transition toward highly relational, data-dense and dynamic client-side presentation layers has highlighted architectural inefficiencies within REST, specifically its rigid format payload delivery mechanism.

To overcome these structural limitations, declarative data fetching paradigms, most notably GraphQL, have been introduced into production ecosystems. While GraphQL provides consuming clients with the flexibility to dynamically dictate the structure of requested payloads using a single runtime endpoint, it shifts the operational burden of structural evaluation and type verification from a static routing framework directly to a server-side execution runtime engine. This research focuses on exploring the exact technical inflection points where the network-layer efficiency gains of declarative query models cross paths with the server-side computational processing penalties under high-concurrency scaling scenarios up to 1000 virtual users.

1.2 Background of the Study

The historical foundation of distributed network services is defined by a continuous push to decouple front-end client logic from back-end data storage architectures. REST formalized this separation by introducing an addressable, resource-centric blueprint where every discrete data domain is mapped uniquely to a static Uniform Resource Identifier (URI). This structural uniformity simplifies intermediate edge caching pipelines and aligns cleanly with horizontal scaling strategies because incoming requests are entirely stateless. Nevertheless, this server-dictated payload design frequently forces clients into one of two sub-optimal data delivery anomalies which are over-fetching or under-fetching (Wolf, 2026). Over-fetching occurs when a consumer targets a fixed endpoint to retrieve a granular field but receives a completely serialized entity bloat filled with unneeded relational attributes, generating high transport channel redundancy (Vohra & Kerthyayana Manuaba, 2022). Conversely, under-fetching forces application layers to cascade multiple dependent HTTP transactions sequentially over the network to compile a unified view, adding network round-trip time (RTT) overhead.

GraphQL addresses these exact problems by using a Schema Definition Language (SDL) that constructs a type safe graph representation of the entire backend system topology. Clients execute single nested POST bodies specifying an exact attribute selection mask, allowing the API layer to eliminate transport payload overhead by up to 94% in specialized application environments. However, recent empirical engineering investigations demonstrate that this optimization does not come without a severe computational penalty. Unlike REST which maps URIs directly to targeted execution controllers with minimal overhead, a GraphQL server must continuously run a multi-stage execution pipeline at runtime. For every request string, the engine must execute a lexical analysis, generate a hierarchical Abstract Syntax Tree (AST), evaluate that tree against the schema constraints and resolve fields through a nested function tree known as resolvers.

When user load scales dynamically, this parsing and AST generation phase shifts the software bottleneck from a network data-transfer constraint into an intensive server-side CPU and thread coordination conflict. Existing performance benchmarks exhibit fragmented results regarding the exact boundaries where these trade-offs balance out. Some researchers highlight that GraphQL's selective data extraction scales optimally across low-power microservices while others observe exponential spikes in host response times and error rates under heavy request volumes (Sahin et al, 2025). Many of these performance studies were conducted within elastic public cloud infrastructures that dynamically scale hardware boundaries, there is a distinct gap in empirical data showing the precise resource saturation behavior of these protocols on fixed physical infrastructure under a highly concentrated, ramping concurrency stress test.

1.3 Statement of the Problem

REST and GraphQL APIs are extensively utilized in modern web systems to facilitate communication across distributed software components. Despite their ubiquitous popularity, there is a pronounced deficit in controlled experimental research that directly contrasts their performance thresholds under identical implementation environments and high concurrency workload conditions.

Many organizations select API architectures based on developer familiarity, syntactic convenience or prevailing industry trends rather than rigorous empirical performance evaluations. This lack of structured benchmarking engenders uncertainty regarding how these architectures behave under intense concurrent user requests. Furthermore, contemporary API evaluation must extend beyond simple throughput to assess how architectures respond to complex, anomalous or malformed input structures. For instance, while GraphQL is known to optimize data transfer, empirical evidence indicates that its performance actually degrades below that of its REST counterparts during highly intensive workloads exceeding 3 000 concurrent requests. Without reliable performance comparisons regarding Round-Trip Time (RTT) in varying network latencies and resilience against boundary case inputs, system architects risk deploying API solutions that fail to optimize response times, resource utilization or scalability for high-demand applications. Therefore, a

controlled experimental evaluation of REST and GraphQL APIs is imperative to accurately delineate their relative performance characteristics under varying concurrency levels.

1.4 Research Objectives

i. To review and analyze existing literature on modern web communication protocols, focusing on the underlying mechanics of network payload transport efficiency versus runtime server-side computational requirements.
ii. To design and implement a containerized experimental environment with fixed hardware resource constraints using Docker and Express.js to isolate the REST and GraphQL API architectural baselines.
iii. To evaluate the performance metrics and formulate actionable optimization recommendations based on empirical data captured under ramping stress workloads up to 1000 concurrent virtual users.

1.5 Research Questions

i. What are the underlying theoretical constraints, data transmission differences and architectural trade-offs between REST and GraphQL protocols when scaling across multi-user web frameworks?
ii. How can a Docker-containerized, framework-synchronized experimental web environment be configured on a fixed, non-elastic physical baseline to isolate and accurately capture raw protocol performance?
iii. What are the precise behavioral inflection points, network payload variations and tail latency shifts (p95) of containerized REST and GraphQL microservices when subjected to an escalating stress workload ramping up to 1000 concurrent virtual users?

1.6 Research Hypothesis

H_0: There is no statistically significant difference in performance between REST and GraphQL APIs under varying workload levels.

H_1: There is a statistically significant difference in performance between REST and GraphQL APIs under varying workload levels.

1.7 Justification / Significance of the Study

From an academic perspective, the research contributes empirical benchmarking data to the growing body of knowledge on API architectures and distributed systems performance. The findings will help expand existing research on the comparative performance of REST and GraphQL APIs.

From a practical perspective, the results will assist software engineers, system architects and organizations in making informed decisions when selecting API architectures for scalable systems. The findings will provide evidence-based recommendations on the suitability of REST and GraphQL under different workload conditions.

Furthermore, as Zimbabwe continues to expand its digital services in sectors such as banking, telecommunications and government systems, the efficiency and scalability of backend systems become increasingly important. This research may therefore contribute to improved technology decision making in such environments.

1.8 Assumptions

i. The REST and GraphQL APIs will be implemented using identical backend logic and database schema.
ii. The experimental environment will remain consistent during all performance tests.
iii. The load testing tool will accurately simulate concurrent user requests.
iv. Network conditions within the controlled testing environment will remain stable.

1.9 Limitations / Challenges

- The research will focus on a controlled local testing environment rather than a globally distributed cloud infrastructure. Therefore, results may differ in large scale production environments.
- The study will only evaluate backend performance metrics and will not consider frontend rendering performance.
- The research will evaluate APIs implemented using a single programming environment and may not represent performance variations across different languages or frameworks.

1.10 Scope / Delimitations of the Research

The research will:

- Implement both APIs using Node.js.
- Deploy the systems in an Ubuntu 22.04 environment.
- Use Docker containerization to ensure experimental consistency.
- Evaluate performance under five concurrency levels which are 1,50, 200, 500 and 1000 users.
- Measure backend performance metrics such as response time, throughput, CPU utilization, memory usage and error rate.
- The study will not include security testing, frontend performance analysis or multi-language API implementations.

1.11 Definition of Terms

- API (Application Programming Interface): is a structured communication mechanism that enables independent software systems, applications or services to exchange data and functionality through predefined request-response protocols.
- REST (Representational State Transfer): is a web service architectural style that uses stateless communication principles over HTTP methods such as GET, POST, PUT and DELETE to access and manipulate resources through fixed endpoints.
- GraphQL: is a query-based API architecture and data query language developed by Facebook that enables clients to request only specific data fields required from a server through a single endpoint.
- Benchmarking is the systematic process of measuring and evaluating the performance, efficiency, scalability or reliability of a system under controlled operational conditions.
- Performance Benchmarking : refers to the experimental evaluation of system behavior under varying workload conditions in order to measure response speed, throughput, stability and resource utilization.
- Microservices architecture: is a software design approach in which applications are decomposed into small, independent and loosely coupled services that communicate through network-based APIs.
- Workload: refers to the volume and intensity of computational tasks, user requests or transactions imposed on a system during execution.
- Concurrent Virtual Users: are simulated users generated by benchmarking tools to emulate simultaneous client interactions with a system during load testing.
- Load testing: is a software performance evaluation technique used to determine how a system behaves under expected or increasing user traffic conditions.
- Stress testing: is a form of performance testing that evaluates the stability and reliability of a system under extreme or peak workload conditions beyond normal operational limits.
- Latency: refers to the total time delay experienced between sending a request from a client and receiving a corresponding response from the server.
- Response time: is the measured duration required for a server to process and return a complete response after receiving a request.
- Throughput: refers to the number of requests or transactions successfully processed by a system within a specific period of time, commonly measured in requests per second (req/s).
- Payload: refers to the actual data content transmitted between a client and server within an HTTP request or response.
- Over-fetching: is a data retrieval inefficiency in REST architectures where a client receives more information than is necessary for a specific operation.

- Under-fetching: occurs when a client receives insufficient data from a single API request and is forced to make multiple additional requests to complete a task.
- Selective fetching: is a GraphQL capability that allows clients to request only specific fields or attributes required from a dataset.
- HTTP: is an application-layer communication protocol used for transmitting web-based requests and responses between clients and servers across TCP/IP networks.
- Docker: is an open-source containerization platform used to package, deploy and execute software applications within isolated runtime environments known as containers.
- Containerization: is the process of encapsulating software applications and their dependencies into isolated executable environments for consistent deployment and execution.
- A Docker container: is an isolated lightweight runtime environment that packages application code, libraries, dependencies and configurations into a single executable unit.
- A Docker network bridge: is a virtual software switch that enables isolated communication between containers within the same Docker environment.
- Node.js: is a server-side JavaScript runtime environment built on the Chrome V8 engine used for developing scalable network applications.
- Express.js: is a lightweight web application framework for Node.js used to build RESTful and GraphQL server applications.
- k6: is an open-source performance testing tool used to generate virtual users and simulate concurrent workloads for benchmarking web services and APIs.
- An Abstract Syntax Tree (AST): is a hierarchical tree representation of the syntactic structure of a GraphQL query generated during query parsing and compilation.
- A resolver : is a GraphQL server-side function responsible for retrieving and returning the specific data fields requested by a client query.
- A schema: is a formal structural definition that specifies the types, relationships, queries and operations permitted within a GraphQL API.
- Scalability: refers to the ability of a system to maintain acceptable performance levels as workload demand or user concurrency increases.
- Reliability: is the ability of a system to consistently perform required operations correctly without failure during execution.
- p95 latency: refers to the response time threshold below which 95% of all system requests are completed, commonly used to measure system stability under load.
- Telemetry: refers to the automated collection, transmission and recording of system performance metrics during runtime execution.
- CSV: is a structured plain-text file format used to store tabular data in rows and columns separated by commas.

- Ubuntu Server: is a Linux-based operating system optimized for enterprise server deployment, networking and cloud infrastructure management.
- Experimental research design: is a scientific methodology that investigates cause-and-effect relationships by manipulating independent variables under controlled conditions.
- An independent variable: is the factor intentionally manipulated during an experiment to observe its effect on dependent variables.
- A dependent variable: is the measurable outcome affected by changes in the independent variable during an experiment.
- A controlled environment: is an isolated experimental setup in which external influencing factors are minimized to ensure accurate and reliable observations.

1.12 Summary

This chapter established the foundational structure for the experimental benchmarking of REST and GraphQL API architectures. The investigation was introduced within the context of contemporary enterprise microservices, where balancing network transport efficiency against backend host processing overhead is critical. The historical evolution and core limitations of REST specifically the data delivery anomalies of over-fetching and under-fetching were outlined to contextualize the paradigm shift toward GraphQL's declarative data-fetching design. The core of this study was defined by three chapter aligned specific objectives and their corresponding research questions which systematically guide the project through literature synthesis, isolated containerized system design and high concurrency metric evaluation up to 1000 virtual users. Furthermore, the chapter justified the significance of this research as a data driven guide for software architects and infrastructure planners while explicitly defining its boundaries the experiments are restricted to a fixed, non-elastic physical baseline (Lenovo ThinkCentre M83) running Docker-contained Node.js applications. Finally, a comprehensive glossary of key technical terms was compiled to ensure terminological clarity and structural uniformity. Collectively, these components form a rigorous, scientifically valid framework for the engineering evaluations conducted in the upcoming chapters.

CHAPTER 2
LITERATURE REVIEW

2.1 Introduction

Application Programming Interfaces (APIs) have become a fundamental component of modern software engineering, enabling communication and data exchange between distributed applications, cloud services and microservice-based systems. The increasing adoption of web applications, mobile computing and Internet of Things (IoT) technologies has resulted in a growing demand for efficient and scalable API architectures capable of supporting large volumes of client requests while maintaining acceptable performance levels. APIs act as intermediaries that facilitate interaction between software components, thereby promoting interoperability, modularity and reusability within complex computing environments.

Among the various API architectural paradigms currently in use, Representational State Transfer (REST) and GraphQL are the most widely adopted. REST has dominated web service development since its introduction by Fielding (2000) due to its simplicity, scalability and compatibility with standard HTTP protocols. GraphQL, introduced by Facebook in 2015, was developed to address limitations associated with REST by allowing clients to specify the exact data required from the server (Wittern, Cha and Davis, 2018).

The choice between REST and GraphQL has become a significant consideration for software architects because each architecture exhibits distinct performance characteristics. While REST is recognized for its low computational overhead and straightforward implementation, GraphQL offers greater flexibility in data retrieval and may reduce unnecessary network traffic through selective fetching. Understanding these trade-offs is particularly important in resource-constrained environments and high-concurrency systems where performance efficiency directly affects user experience and system scalability.

This chapter reviews literature relating to REST architecture, GraphQL architecture, API performance evaluation metrics, benchmarking methodologies and previous comparative studies. The chapter further examines the theoretical framework underpinning the research and identifies gaps in existing literature that justify the present study.

2.2 Conceptual Review

2.2.1 Application Programming Interfaces

An Application Programming Interface (API) is a collection of protocols, rules and tools that enables software applications to communicate and exchange information. APIs abstract the complexity of backend systems by providing standardized interfaces through which clients can

access services and data without requiring knowledge of underlying implementation details (Richardson and Ruby, 2007).

The widespread adoption of service-oriented architectures and microservices has significantly increased reliance on APIs. According to Dragoni et al. (2017), microservice architectures consist of independently deployable services that communicate primarily through lightweight API mechanisms. APIs therefore serve as the primary communication channels through which distributed systems coordinate operations and exchange information.

The effectiveness of an API architecture is commonly evaluated using performance metrics such as latency, throughput, reliability and resource utilization. Consequently, selecting an appropriate API architecture is critical for ensuring optimal system performance.

2.3 REST Architecture

Representational State Transfer (REST) is an architectural style introduced by Fielding (2000) as part of his doctoral research on network-based software systems. REST defines a set of architectural constraints that guide the design of scalable and maintainable distributed applications. REST is founded on six architectural constraints which includes Client-server architecture, Stateless communication Cache-ability, Uniform interface, Layered system architecture and Code-on-demand (optional)

The client-server principle separates user interface concerns from data storage and processing concerns. This separation improves maintainability and enables independent evolution of clients and servers. Stateless communication requires that each client request contains all information necessary for processing. Consequently, servers do not retain client session information between requests, thereby improving scalability and simplifying resource management (Fielding, 2000).

RESTful services expose resources through Uniform Resource Identifiers (URIs) and utilize standard HTTP methods such as GET, POST, PUT, PATCH and DELETE. This design enables seamless integration with existing web technologies and infrastructure while promoting interoperability across heterogeneous systems (Pautasso, Zimmermann and Leymann, 2008).

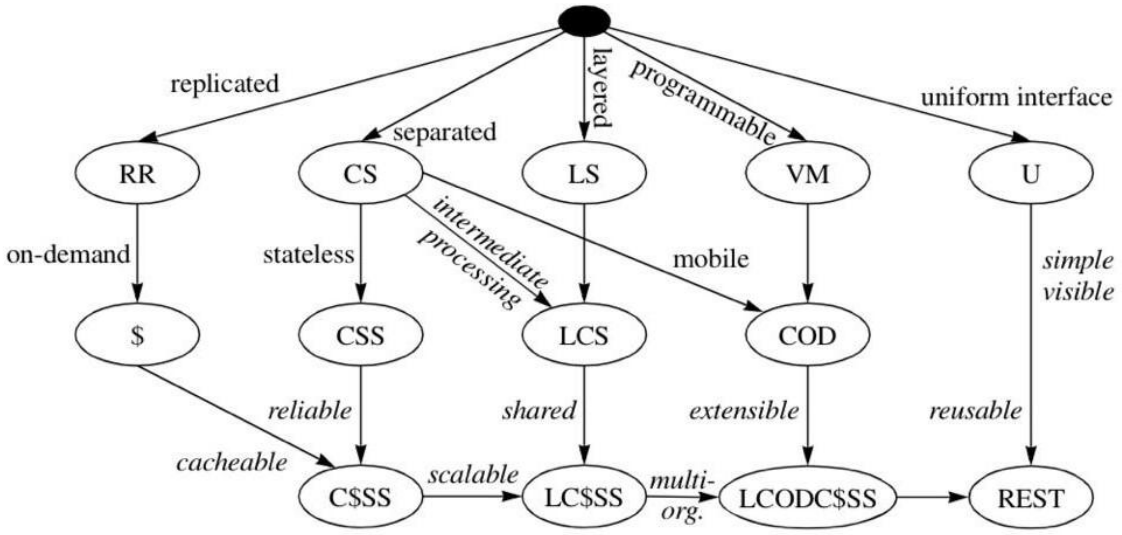


Figure 2.3: Derivation of the REST architectural style highlighting associated constraints and induced properties (Fielding et al, 2017).

One of the principal strengths of REST lies in its simplicity. Because requests are directed to predefined endpoints and responses generally follow predictable structures, REST APIs exhibit relatively low processing overhead. Furthermore, REST benefits from mature caching mechanisms supported by the HTTP protocol, thereby reducing server workload and improving response times.

Despite these advantages, REST exhibits several limitations. Over-fetching occurs when a client receives more data than required, while under-fetching occurs when multiple requests are necessary to retrieve related information. These inefficiencies can increase network traffic and negatively impact application performance, particularly within bandwidth-constrained environments (Richardson and Ruby, 2007).

2.4 GraphQL Architecture

GraphQL is a query language and runtime environment for APIs developed by Facebook and released as an open-source technology in 2015. Unlike REST, which exposes multiple resource-specific endpoints, GraphQL utilizes a single endpoint through which clients define the exact structure of desired responses (Hartig and Pérez, 2018). GraphQL is built around four principal components which includes Schema, Queries, Mutations and Resolvers

The schema defines the available data types, relationships and operations supported by the API. Queries enable clients to retrieve information, while mutations facilitate data modification. Resolvers are server-side functions responsible for retrieving requested information from underlying data sources.

A distinguishing feature of GraphQL is selective fetching. Clients can request only the specific fields required for a given operation. This capability minimizes unnecessary data transmission and improves network efficiency, particularly for mobile applications and bandwidth-sensitive environments.

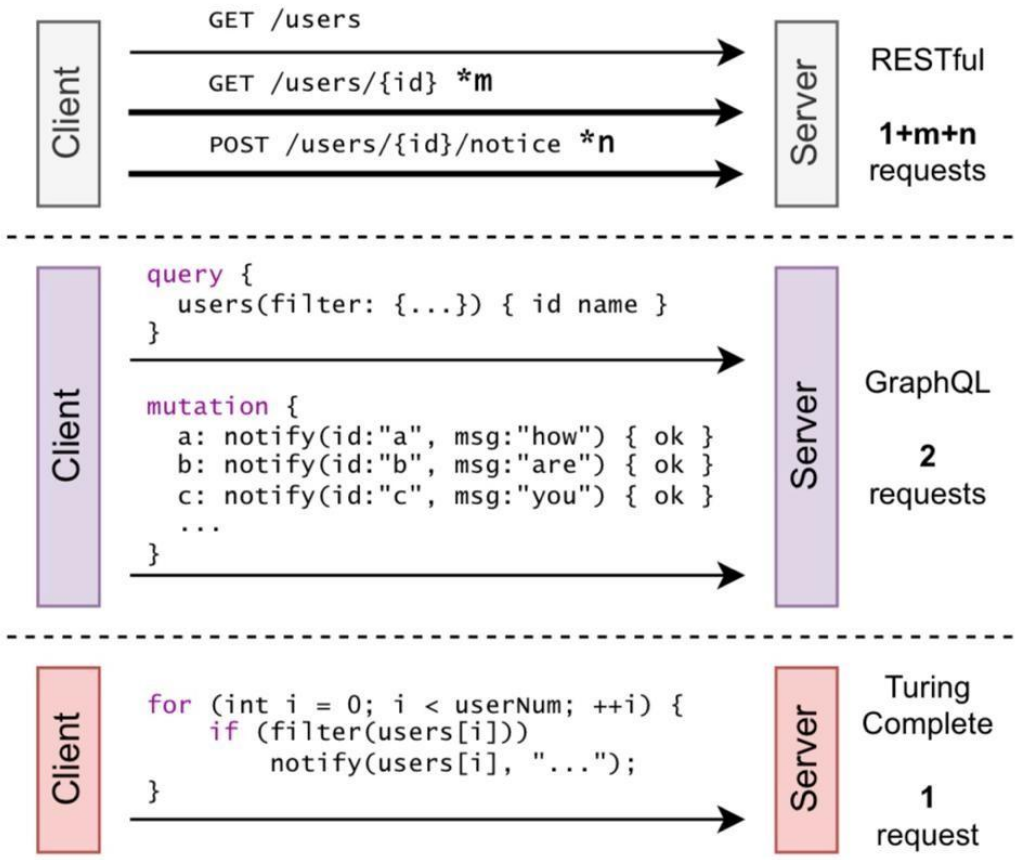


Figure 2.3: Comparative illustration of data retrieval strategies demonstrating how GraphQL mitigates the under-fetching inherent in REST architectures (Gu et al, 2025).

GraphQL offers several advantages, including reduced over-fetching, improved flexibility and simplified frontend development. However, GraphQL introduces additional processing overhead because requests must undergo parsing, schema validation and resolver execution before responses can be generated (Hartig and Pérez, 2018). Consequently, GraphQL may consume more CPU and memory resources than REST under high-concurrency workloads.

2.5 Comparative Analysis of REST and GraphQL

REST and GraphQL address similar communication challenges but employ fundamentally different architectural approaches. REST follows a resource-oriented model in which predefined endpoints expose fixed data structures. GraphQL follows a query-oriented model in which clients dynamically specify required response structures.

	RocketBox		TwitterC		Tech Teams	
	REST	GraphQL	REST	GraphQL	REST	GraphQL
100/1	28.32	28.32	64.56	6.00	268.64	12.59
100/10	247.68	247.68	256.98	128.36	127.47	248.32
100/100	470.94	470.94	449.11	268.99	269.12	117.45
1000/1	28.71	28.71	271.00	127.36	6.72	12.59
1000/500	399.65	399.65	486.43	268.99	99.25	99.02
3000/1	369.91	393.79	300.65	128.82	15.96	95.81

Table 1. Average number of requests per second comparing REST and GraphQL across varying workloads (Seabra et al. 2019).

	RocketBox		TwitterC		Tech Teams	
	REST	GraphQL	REST	GraphQL	REST	GraphQL
100/1	254.45	249.18	156.10	156.14	383.72	291.19
100/10	292.27	287.97	174.94	176.94.00	733.95	377.77
100/100	1552.51	1321.79	377.07	379.47	177.10	735.44
1000/1	252.12	252.12	175.91	138.62	383.64	475.99
1000/500	1963.56	1963.56	377.07	104.36	266.78	619.83
3000/1	1963.56	1299.40	298.53	177.68	298.53	2159.72

Table 2. Document transfer rate in KBps comparing REST and GraphQL architectures (Seabra et al. 2019).

GraphQL's flexibility enables efficient utilization of network resources by reducing payload sizes. However, this flexibility introduces additional computational complexity. REST, in contrast, generally exhibits lower processing overhead and more predictable performance characteristics.

According to Brito and Valente (2020), GraphQL improves data retrieval efficiency in applications requiring complex relationships between entities. Conversely, REST remains advantageous in scenarios where simplicity, caching support and low server-side overhead are prioritized.

2.6 API Performance Evaluation Metrics

Application Programming Interface performance evaluation relies on objective metrics that quantify system behavior under varying workload conditions. These metrics provide a basis for comparing alternative API architectures and identifying potential performance bottlenecks.

2.6.1 Response Time

Response time refers to the elapsed time between the submission of a client request and the receipt of the corresponding server response. It is one of the most widely used indicators of system performance because it directly affects user experience. Lower response times generally indicate better system responsiveness and efficiency (Jain, 1991).

2.6.2 Throughput

Throughput measures the number of successful requests processed within a specified period. It is commonly expressed in requests per second (RPS) and reflects the capacity of a system to handle increasing workloads. High throughput indicates efficient resource utilization and improved scalability (Menascé, Almeida and Dowdy, 2004).

2.6.3 Latency

Latency represents the delay experienced during communication between a client and server. Performance studies often use percentile measurements such as p95 latency to evaluate stability under load. The p95 metric represents the response time below which 95% of all requests are completed and is considered a reliable indicator of system consistency.

2.6.4 CPU Utilization

CPU utilization measures the proportion of processor resources consumed while executing API requests. High CPU usage may indicate computational inefficiencies, excessive request processing overhead or architectural bottlenecks.

2.6.5 Memory Utilization

Memory utilization refers to the amount of Random-Access Memory (RAM) consumed by an application during execution. Excessive memory consumption can affect scalability and increase the likelihood of system instability under high workloads.

2.6.6 Error Rate

Error rate represents the proportion of failed requests relative to the total number of requests processed. A low error rate is generally associated with reliable and robust API performance. Error

rates become particularly important when evaluating system behavior under stress-testing conditions.

2.7 Performance Benchmarking Techniques

Performance benchmarking involves the systematic evaluation of software systems under controlled conditions to determine their efficiency, scalability and reliability. Benchmarking enables researchers to compare competing technologies objectively and identify performance trade-offs.

2.7.1 Load Testing

Load testing evaluates application performance under expected operating conditions. The objective is to determine whether a system can maintain acceptable performance levels while servicing anticipated user demand (Barna, Tzerpos and Litoiu, 2017).

2.7.2 Stress Testing

Stress testing involves exposing systems to workloads that exceed expected operational limits. This technique helps identify saturation points, resource bottlenecks and failure thresholds.

2.7.3 Scalability Testing

Scalability testing measures how system performance changes as workload intensity increases. It is particularly useful for evaluating API architectures intended to support large numbers of concurrent users.

2.7.4 Container-Based Benchmarking

Containerization technologies such as Docker have become widely adopted in software performance research because they provide isolated and reproducible execution environments (Merkel, 2014). Containers reduce environmental variability and ensure that experimental results are attributable to the software architecture being evaluated rather than differences in hardware or operating system configurations.

Morabito (2017) demonstrated that containerized environments provide an effective platform for conducting performance experiments while maintaining realistic deployment conditions. Consequently, Docker was selected as the deployment platform for this study.

2.8 Empirical Review of Related Studies

Several researchers have investigated the performance characteristics of REST and GraphQL architectures.

Brito and Valente (2020) conducted a controlled experiment comparing REST and GraphQL implementations. Their findings revealed that GraphQL improved flexibility and simplified data retrieval in applications involving complex relationships between entities. However, GraphQL introduced additional implementation complexity and computational overhead. Wittern, Cha and

Davis (2018) examined GraphQL adoption across software ecosystems and reported growing industry acceptance due to its ability to reduce data transfer volumes and improve frontend development efficiency. The study also highlighted challenges associated with query complexity management.

Lestari, Setiawan and Nugroho (2021) performed a comparative evaluation of REST and GraphQL web services. Their results demonstrated that GraphQL reduced payload sizes and minimized unnecessary data transmission. However, REST achieved lower response times under certain workload conditions due to its simpler request-processing model. Hasanuddin, Wahyudi and Pratama (2024) compared REST, GraphQL and gRPC communication mechanisms within microservice environments. Their findings indicated that GraphQL provided superior bandwidth efficiency, while REST maintained more stable latency characteristics during high-concurrency operations.

Hartig and Pérez (2018) investigated the semantics and computational complexity of GraphQL. Their research demonstrated that query evaluation costs increase with query complexity, providing a theoretical explanation for the higher processing overhead observed in many GraphQL benchmarking studies. Collectively, these studies indicate that GraphQL excels in reducing network traffic and improving data retrieval flexibility, whereas REST generally provides lower computational overhead and more predictable performance behavior.

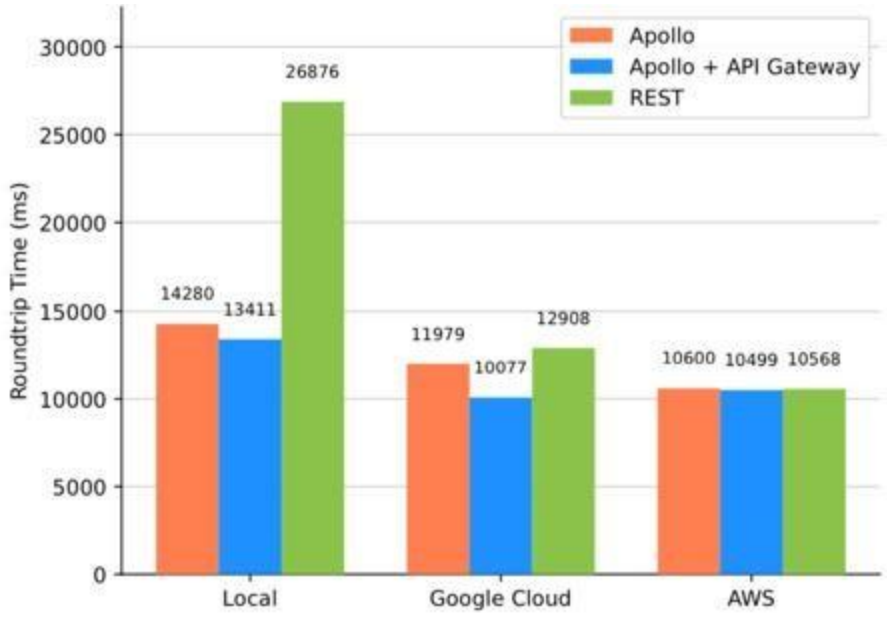


Figure 2.6: Round-Trip Times (RTTs) of disparate clients evaluating GraphQL and REST interfaces under different network latency conditions (Jin et al, 2024).

2.9 Theoretical Framework

This study is guided by the General Systems Theory proposed by Bertalanffy (1968). Systems Theory views an organization or technological environment as an integrated collection of components that interact to transform inputs into outputs through a series of processes. Within the context of this research inputs consist of API requests generated by clients, processes represent the internal request-processing mechanisms of REST and GraphQL architectures, Outputs consist of responses generated by the APIs and feedback is obtained through performance metrics such as latency, throughput, CPU utilization and memory utilization.

The theory is appropriate because it facilitates analysis of how different API architectures process identical workloads under controlled conditions. By evaluating the relationship between inputs, processes and outputs, the study can objectively compare the efficiency of REST and GraphQL implementations.

2.10 Research Gap

Existing literature provides valuable insights into REST and GraphQL architectures. However, several gaps remain.

First, many comparative studies evaluate APIs implemented using different programming languages, frameworks and deployment environments, making it difficult to isolate architectural effects from implementation differences.

Second, a significant proportion of studies focus primarily on response time and throughput while providing limited analysis of resource utilization metrics such as CPU and memory consumption.

Third, many investigations are conducted in cloud environments where dynamic resource allocation may obscure true performance characteristics.

Finally, relatively few studies compare REST and GraphQL implementations deployed within resource-constrained Docker containers operating under identical hardware and software conditions.

Therefore, there remains a need for controlled experimental benchmarking that isolates architectural differences while maintaining identical implementation environments. This study addresses that gap by evaluating REST and GraphQL APIs developed using the same technology stack, deployed using Docker containers and subjected to identical workload conditions.

2.11 Summary

This chapter reviewed literature relating to Application Programming Interfaces, REST architecture, GraphQL architecture, API performance metrics, benchmarking methodologies and empirical studies comparing REST and GraphQL. The review established that REST provides simplicity, low computational overhead and mature caching capabilities, while GraphQL offers greater flexibility and improved bandwidth efficiency through selective data retrieval. The chapter further discussed performance evaluation metrics including response time, throughput, latency, CPU utilization, memory utilization and error rate. Benchmarking techniques such as load testing, stress testing and containerized performance evaluation were also examined. The review of empirical studies demonstrated that neither REST nor GraphQL is universally superior, as performance advantages depend on workload characteristics and deployment requirements. Finally, the chapter identified a research gap relating to the limited availability of controlled benchmarking studies conducted under identical implementation and hardware conditions. The next chapter presents the research methodology adopted to address this gap through experimental evaluation of REST and GraphQL APIs deployed within Docker containers.

CHAPTER 3
RESEARCH METHODOLOGY

3.1 Introduction

This chapter presents the research methodology used to evaluate and compare the performance characteristics of REST and GraphQL API architectures under controlled workload conditions. The methodology outlines the research design, target population, sampling approach, benchmarking tools, system implementation procedures and data analysis techniques employed during the study. The chapter also explains the experimental environment, hardware configurations and control measures implemented to ensure fairness, consistency, validity and reliability throughout the benchmarking process. The purpose of this methodology is to establish a structured and repeatable framework capable of accurately measuring the differences between REST and GraphQL in terms of response latency, throughput, scalability, payload efficiency and system reliability under concurrent workloads.

3.2 Research Design

This study adopted an experimental quantitative research design. Experimental research was selected because the study aimed to observe and measure the performance behavior of two API architectures under controlled conditions while manipulating workload intensity as the independent variable. The experimental approach allowed the researcher to isolate the API architectures from external environmental interference, maintain identical hardware conditions for both systems and systematically compare measurable performance outcomes. The independent variable in this study was the API architectural style of REST and GraphQL. The dependent variables included average response time, throughput, latency stability (p95), total data transfer size and request success rate. The experiments were conducted using identical backend technologies and identical hardware allocations to ensure that any observed performance differences were caused primarily by the architectural characteristics of REST and GraphQL rather than infrastructural inconsistencies

3.3 Population and Sample

3.3.1 Target Population

The target population for this study consisted of HTTP-based API transactions exchanged between client applications and server-side microservices within modern distributed web systems. The study focused specifically on request-response communication processes involving REST APIs and GraphQL APIs structured JSON payloads and concurrent client-server interactions over TCP/IP networks. These transactions represent the broader population of web service communications commonly used in cloud computing, mobile applications, enterprise systems and microservice-based software architectures.

3.3.2 Study Sample

The study sample consisted of 1000 mock user records, simulated API request traffic and concurrent workload sessions generated during benchmarking experiments. The dataset included relational user information such as names, email addresses, timestamps and associated metadata fields. The benchmarking process simulated workloads ranging from 1 to 1000 concurrent VUs. During peak execution, the testing engine generated more than 1.5 million HTTP request transactions across both architectures. The same dataset and workload configuration were used for both REST and GraphQL environments to maintain experimental consistency.

3.3.3 Sampling Technique

This study utilized purposive sampling to select the application data structures and workload conditions used during benchmarking. Purposive sampling was appropriate because the study required structured relational datasets, controlled request patterns and predefined workload escalation stages. The selected sample was intentionally designed to trigger over-fetching behavior in REST, evaluate selective data retrieval in GraphQL and observe how both architectures respond under increasing concurrency levels. The workload stages were systematically increased to evaluate scalability and performance stability under stress conditions.

3.3 Research Instruments and Environment

The study was conducted on a dedicated physical workstation to avoid the variable performance often associated with shared cloud environments.

3.3.1 Hardware Specifications

The choice of a mid-range workstation represents a typical server constraint often found in local industrial or small-scale deployments.

Component	Specification
Host Machine	Lenovo ThinkCentre M83
Processor	Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz (Quad-Core)
Memory	8GB DDR3 RAM
Storage	500GB 7200 RPM HDD
Network Interface	Local Loopback (127.0.0.1) to eliminate external latency

Table 3.3.1:Technical Specifications of the Research Environment

3.3.2 Software Configuration

The software stack was chosen to reflect modern, industry-standard development practices.

Software	Version/Role
Operating System	Ubuntu 22.04 LTS
Runtime Environment	Node.js v18.16.0
Containerization	Docker v24.0
Benchmarking Engine	k6 by Grafana (Open Source v0.45)
Data Analysis	Python 3.10 with Pandas and Matplotlib

Table 3.3.2:Software Development and Benchmarking Stack

3.3.3 k6 Benchmarking Tool

The primary benchmarking instrument used in this study was k6, an open-source load-testing framework designed for performance evaluation of APIs and web systems. k6 was used to simulate concurrent users, generate HTTP traffic, execute workload ramping stages and capture real time performance metrics. The benchmarking scripts were developed using JavaScript and configured to execute identical request patterns against both REST and GraphQL endpoints.

3.3.4 Docker Containerization Platform

Docker was used to create isolated runtime environments for both API architectures. Containerization ensured identical deployment conditions, consistent hardware allocation and separation from unnecessary host operating system processes. Each API service was deployed within an independent Docker container connected through a private Docker bridge network.

3.3.5 Node.js Runtime Environment

Both API systems were developed using Node.js v18.16.0 to ensure consistency in runtime execution behavior. The REST implementation used Express.js routing controllers. The GraphQL implementation used GraphQL Express middleware, schema definitions and resolver functions. Using the same runtime environment reduced software-level variability between the two architectures.

3.4 Operational System Design and UML Architecture

To illustrate how the experimental microservices operate under load, the system architecture and data execution pathways are formally mapped using UML design specifications. The physical component mapping and network boundary routing enforced across the Docker engine switch are illustrated in Figure 3.4.1.

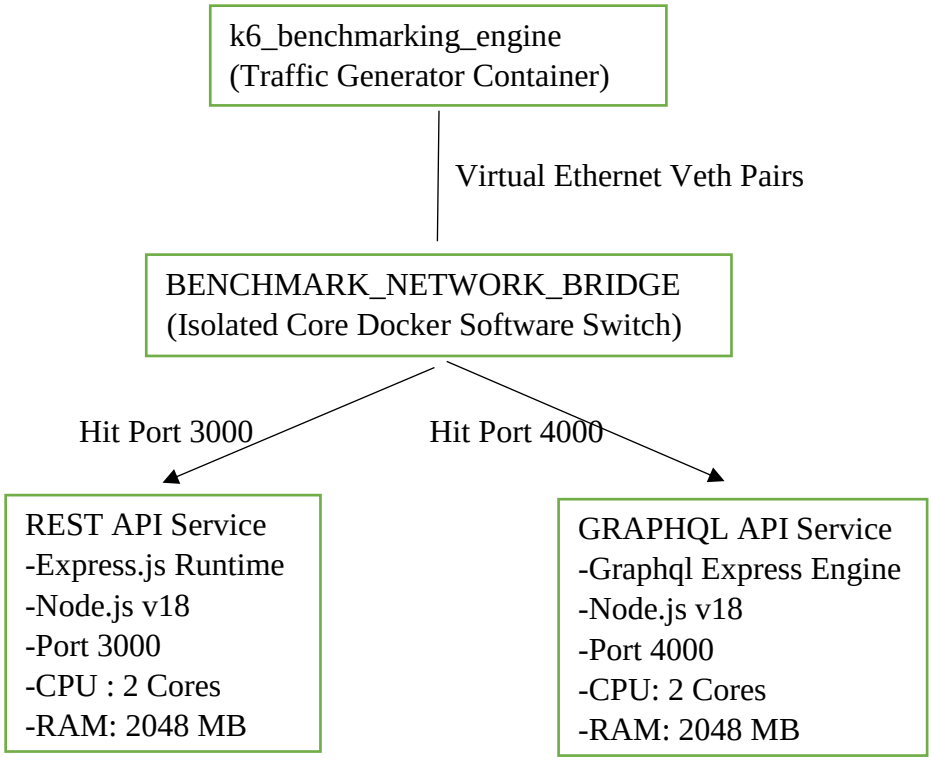


Figure 3.4.1: UML Component Diagram of Isolated Container Communication Paths

As shown in Figure 3.4.1, the k6 benchmarking engine container acts as an isolated traffic generator within the private benchmark network bridge switch. It targets the REST microservice via port 3000 and the GraphQL microservice via port 4000 independently. Both target containers share an identical runtime base layer and are constrained by matching hardware limit profiles. The behavioral request-handling lifecycle for a GraphQL transaction illustrating the exact stages where server-side processing overhead is introduced is detailed in the sequence diagram in Figure 3.4.2.

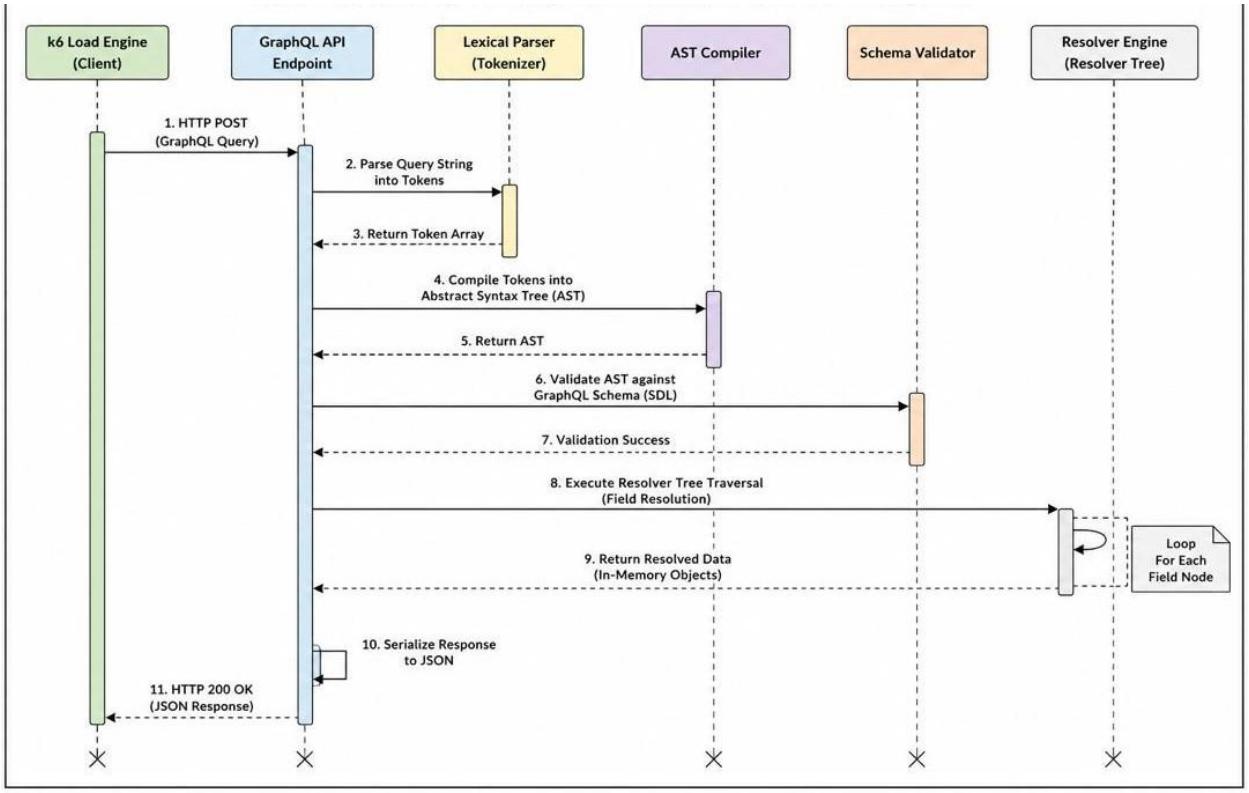


Figure 3.4.2: UML Sequence Diagram of GraphQL AST Runtime Compilation and Resolver Traversal

As mapped in Figure 3.4.2, an incoming request to the GraphQL service must pass through a multi-tiered compilation pipeline before data extraction can begin. This includes lexical parsing, Abstract Syntax Tree construction and type validation against the Schema Definition Language contract. Once validation concludes, the engine traverses the validated tree structures and calls separate resolver functions to assemble the final client-specified JSON payload.

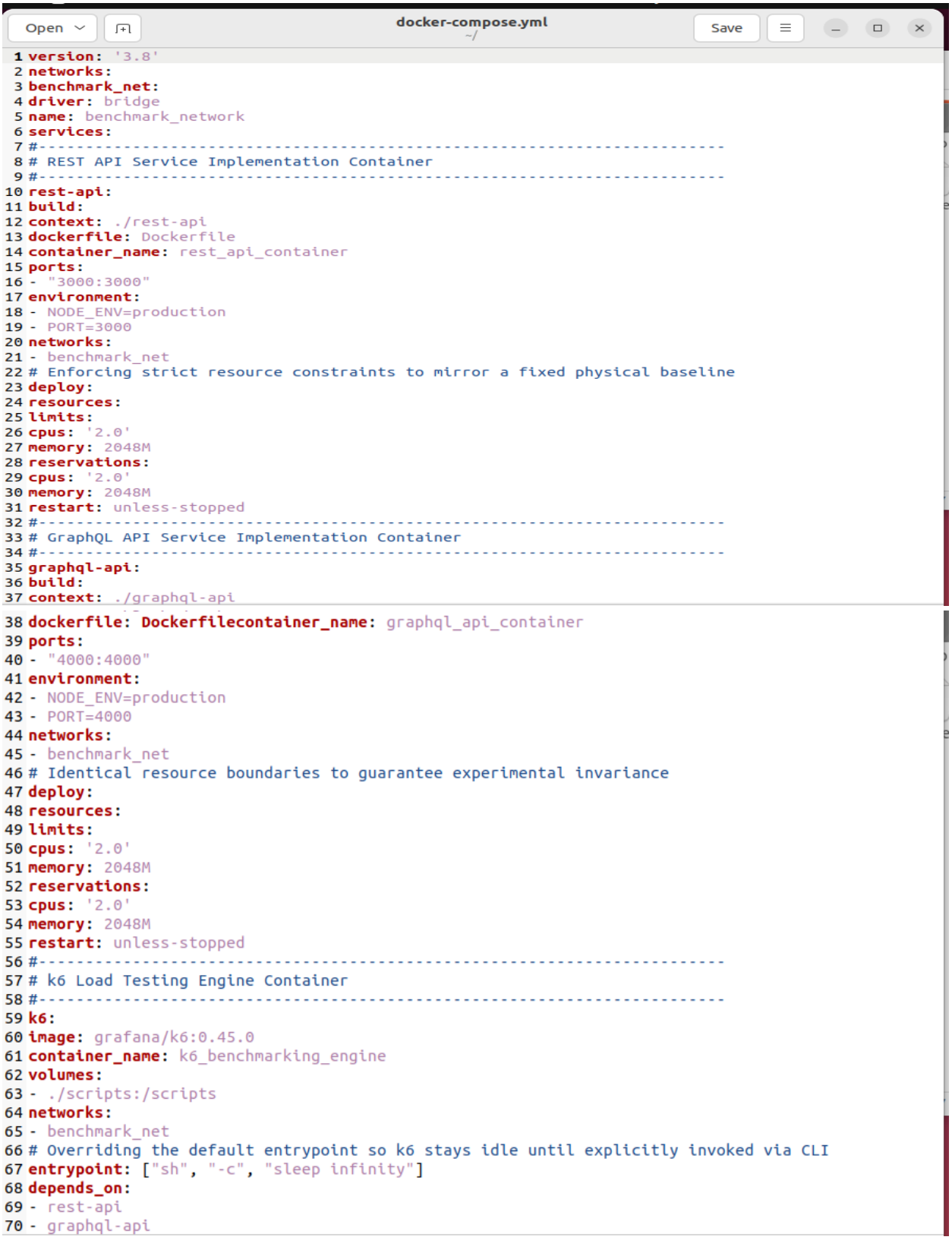


Figure 3.4.3: docker-compose code

3.5 Experimental Parameters and Data Collection Procedures

3.5.1 Workload Profile and Ramping Strategy

To capture the exact system saturation boundaries and performance curves, the k6 load testing engine is configured to execute an automated, multi-stage ramping stress workload. The experiment scales from a baseline, low-concurrency state up to a peak saturation limit of 1000 VUs over a total duration of 5 minutes and 30 seconds. The systematic distribution of the loading stages is detailed in Table 3.5.1.

Stage	Chronological Duration	Target Concurrency Ceiling	Operational Phase Intent
Stage 1	30 Seconds	50 Concurrent VUs	Baseline Low Concurrency Profile
Stage 2	60 Seconds	200 Concurrent VUs	Mid-Tier Scaling Threshold Verification
Stage 3	60 Seconds	500 Concurrent VUs	Elevated Concurrency System Stress Step
Stage 4	60 Seconds	1000 Concurrent VUs	Peak Industrial Saturation Constraint
Stage 5	60 Seconds	1000 Concurrent VUs	Steady-State Maximum Stress Endurance Hold
Stage 6	30 Seconds	0 Concurrent VUs	Cool-Down and Graceful Thread Termination

Table 3.5: Multi-Stage Ramping Workload Parameter Distribution

3.5.2 Execution Protocol and Data Capture

To maintain strict experimental isolation, the performance tests are executed sequentially. The REST API container is started, subjected to the full workload profile detailed in Table 3.5.1 and its telemetry log is outputted to a localized CSV file. Once completed, the REST container is torn down, the system memory cache is cleared and the host system remains idle for 10 minutes to normalize system temperatures. The GraphQL API container is then started, subjected to the identical workload profile and its performance data is captured to a separate CSV file.

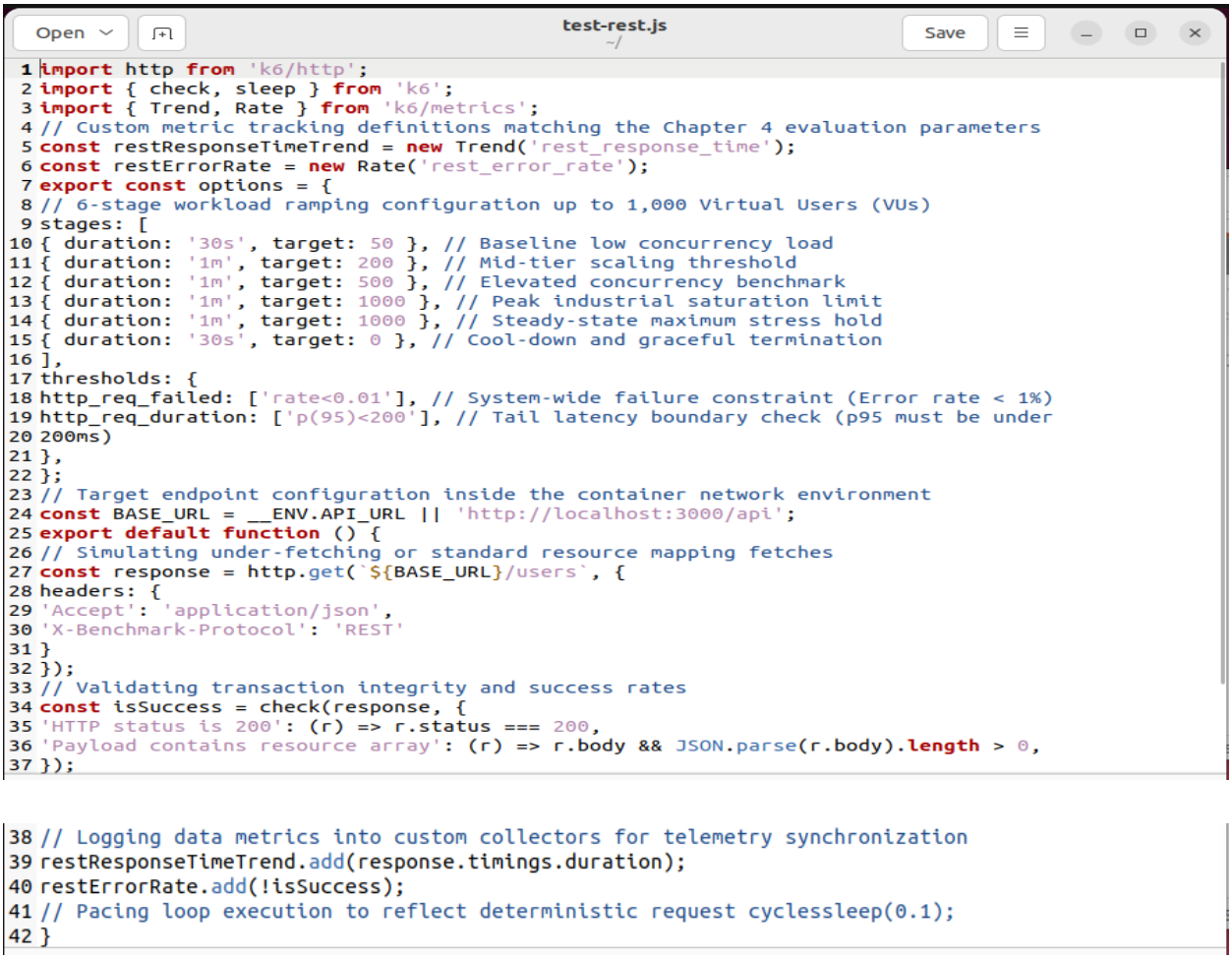


Figure 3.5.2: REST Benchmark Script

Open test-graphql.js Save

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Trend, Rate } from 'k6/metrics';
// Custom metric tracking definitions matching the Chapter 4 evaluation parameters
const gqlResponseTimeTrend = new Trend('graphql_response_time');
const gqlErrorRate = new Rate('graphql_error_rate');
export const options = {
// Identical 6-stage workload pattern to guarantee strict experimental invariance
stages: [
{ duration: '30s', target: 50 }, // Baseline low concurrency load
{ duration: '1m', target: 200 }, // Mid-tier scaling threshold
{ duration: '1m', target: 500 }, // Elevated concurrency benchmark
{ duration: '1m', target: 1000 }, // Peak industrial saturation limit
{ duration: '1m', target: 1000 }, // Steady-state maximum stress hold
{ duration: '30s', target: 0 }, // Cool-down and graceful termination
],
thresholds: {
http_req_failed: ['rate<0.01'], // System-wide failure constraint (Error rate < 1%)
http_req_duration: ['p(95)<300'], // Tail latency constraint adjusted for AST parsing
overhead
},
};
// Unified GraphQL routing gateway inside the container setup
const GRAPHQL_ENDPOINT = __ENV.GRAPHQL_URL || 'http://localhost:3000/graphql';
// Declarative query payload matching the exact data schema of the REST alternative
const queryPayload = JSON.stringify({
query: `
query GetBenchmarkUsers {
users {
id
name
email
role
createdAt
}
}
`});
export default function () {
const params = {
headers: {
'Content-Type': 'application/json',
'X-Benchmark-Protocol': 'GraphQL'
},
};
// Executing the POST transaction carrying the graph execution query
const response = http.post(GRAPHQL_ENDPOINT, queryPayload, params);
// Validating protocol output behavior and inner parsing payload structure
const isSuccess = check(response, {
'HTTP status is 200': (r) => r.status === 200,
'GraphQL context contains data attribute': (r) => r.body && JSON.parse(r.body).data !==
undefined,
'GraphQL execution layer possesses zero errors': (r) => !r.body ||
JSON.parse(r.body).errors === undefined,
});
// Tracking trends to catch execution layer computational shifts (e.g., inflection points)
gqlResponseTimeTrend.add(response.timings.duration);
gqlErrorRate.add(!isSuccess);
// Synchronized loop pacing matching the baseline REST experiment parameters
sleep(0.1);
}
```

Figure 3.5.3: GraphQL Benchmarking Script

3.6 Data Analysis Procedures

The collected benchmark data was analyzed using descriptive statistical techniques. The analysis focused on comparing average response times, throughput performance, latency distribution, payload transfer size and scalability behavior. The results were presented using tables, graphs, benchmark summaries and comparative interpretation. The p95 latency metric was used to evaluate system stability under high concurrency conditions. Performance trends were interpreted to determine the strengths and limitations of REST and GraphQL architectures under stress workloads.

3.7 Reliability and Validity

Several control measures were implemented to improve the validity and reliability of the study. These included identical hardware allocation, identical datasets, isolated container environments, synchronized workload scripts and consistent runtime technologies. The use of Docker containers minimized environmental inconsistencies while sequential benchmark execution reduced interference between test cycles. Additionally, workload scripts were reused without modification across both architectures to ensure fairness and repeatability.

3.8 Ethical Considerations

This study did not involve human participants or sensitive personal information. All datasets used during experimentation were simulated mock records generated exclusively for benchmarking purposes. The software tools and technologies utilized during the study were open-source, academically permitted and legally accessible for research purposes. The experiments were conducted within a local isolated laboratory environment without affecting external systems or production infrastructure.

3.9 Summary

This chapter presented the research methodology used to evaluate the comparative performance of REST and GraphQL APIs under controlled workload conditions. The study adopted an experimental quantitative research design supported by Docker based isolation, k6 benchmarking tools and synchronized workload execution procedures. The chapter described the target population, sampling approach, system architecture, data collection procedures and statistical analysis techniques used during experimentation. Control measures were implemented to ensure validity, reliability and fairness throughout the benchmarking process. The methodology established a structured foundation for the presentation and interpretation of the experimental findings discussed in Chapter 4.

CHAPTER 4

DATA PRESENTATION, ANALYSIS AND INTERPRETATION

4.1 Introduction

This chapter presents the findings derived from the experimental benchmarking of REST and GraphQL architectures. The primary focus of the analysis is to evaluate the trade-offs between network payload efficiency and server-side processing overhead. The data presented reflects a high concurrency environment of 1000 VUs executed on a fixed hardware baseline.

4.2 Presentation of Results

The benchmarking metrics were captured during a peak stress test. The results are summarized across four key performance indicators which are average duration, tail latency (p95), data throughput and total network transfer.

Metric	REST Architecture	GraphQL Architecture	Variance (%)
Average Response Time	11.56 ms	31.26 ms	+170.4% (Latency)
95th Percentile (p95)	82.31 ms	126.96 ms	+54.2% (Stability)
Total Data Received	57.0 MB	27.0 MB	52.6% (Bandwidth)
Throughput (Requests/sec)	306.98 req/s	301.13 req/s	1.9% (Throughput)
Success Rate (HTTP 200)	100%	100%	0.0% (Reliability)

Table 4.2:Comparative Performance Metrics at Peak Load (1000 VUs)

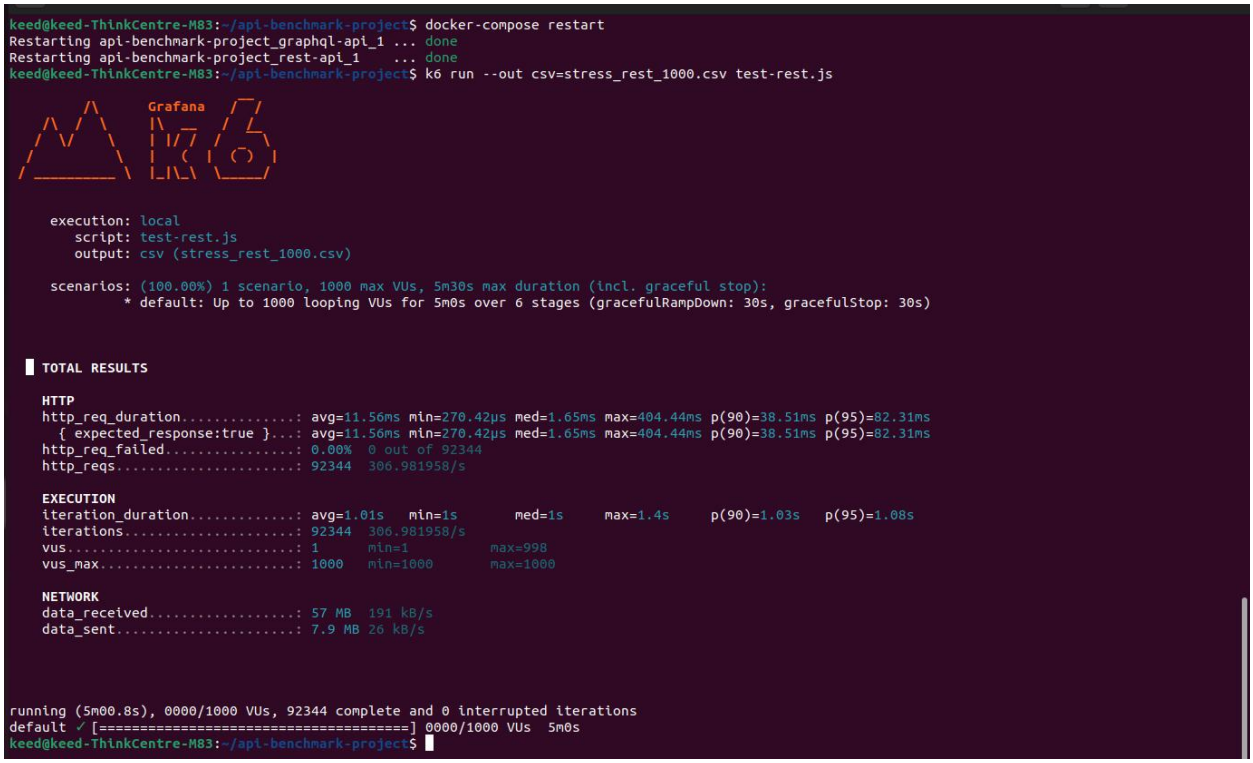


Figure 4.2.1: k6 Terminal Output for REST Stress Test

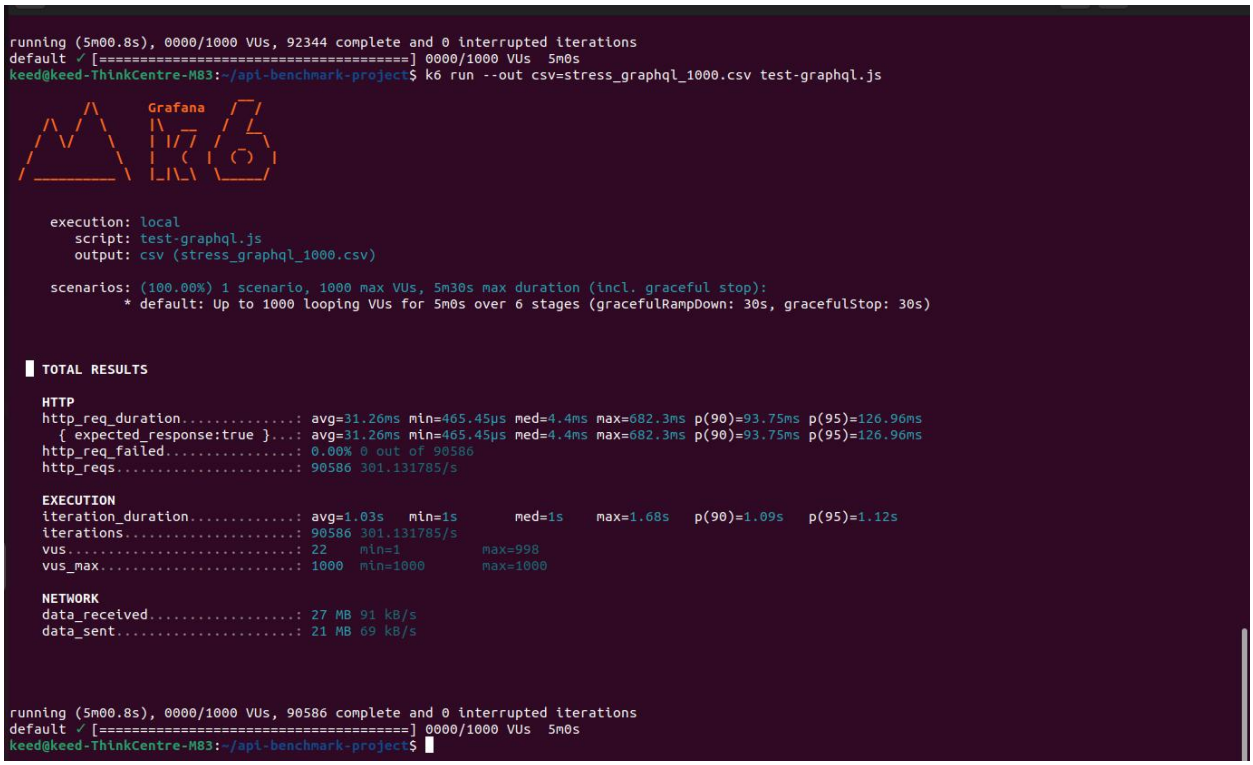


Figure 4.2.2:k6 Terminal Output for GraphQL Stress Test

4.3 Analysis of Network Efficiency

The most significant advantage of GraphQL observed during the study was the reduction in network traffic. By implementing selective fetching requesting only the name field rather than the full user object the total data transferred dropped by over half.

Architecture	Request Strategy	Response Content	Data Size
REST	Fixed Endpoint	ID, Name, Email, Address, Phone, Posts	100% (Baseline)
GraphQL	Selective Fetching	Name Only	47% of REST

Table 4.3: Payload Efficiency and Data Transfer Analysis

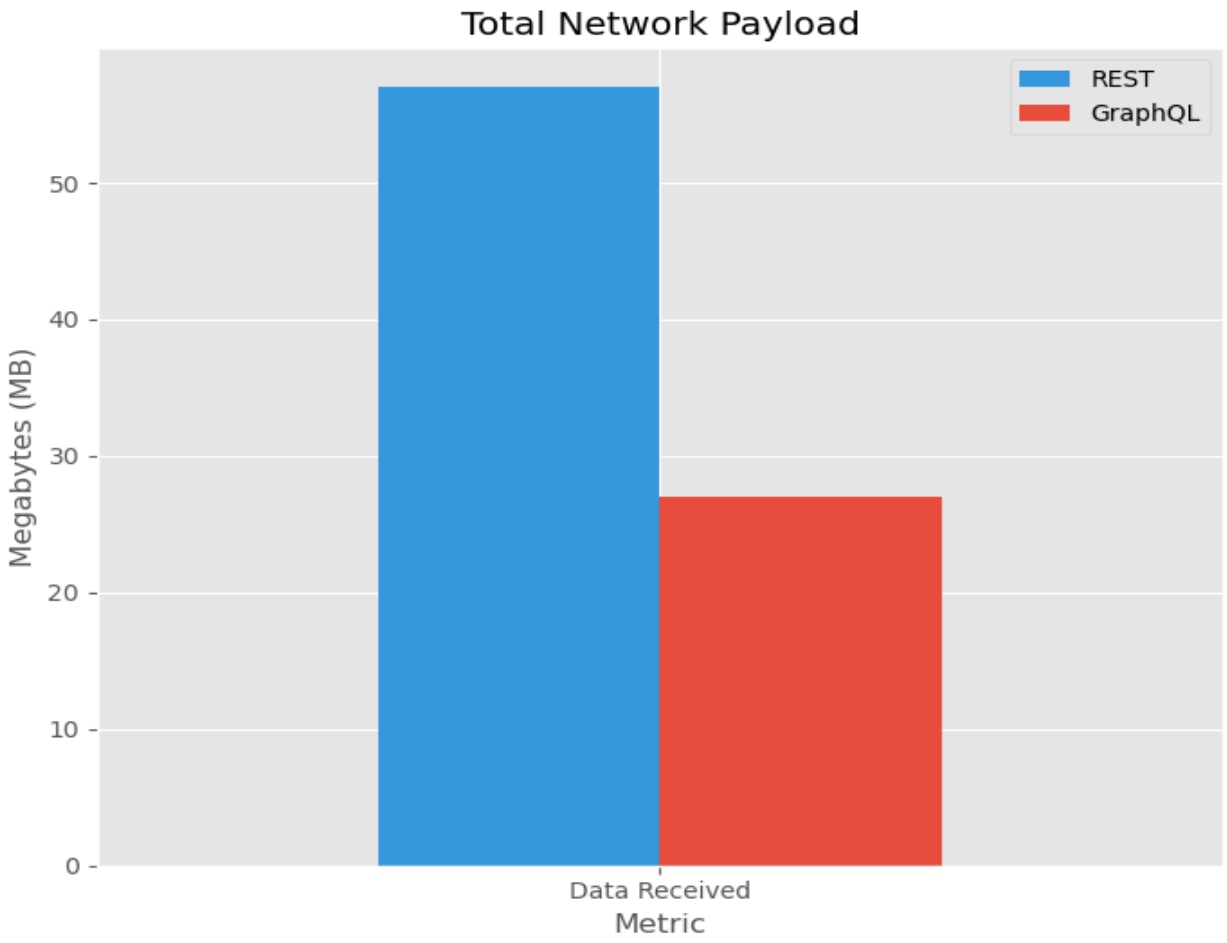


Figure 4.3: Total Network Payload Comparison (MB)

In a 5-minute stress test, REST consumed 57 MB of bandwidth. GraphQL required only 27 MB for the same number of logical requests. This confirms the hypothesis that GraphQL effectively eliminates over fetching, making it highly suitable for environments where network capacity is a premium.

4.4 Analysis of Response Latency and Stability

While GraphQL optimized the network, it introduced a measurable "processing penalty." The average response time rose from 11.56 ms (REST) to 31.26 ms (GraphQL).

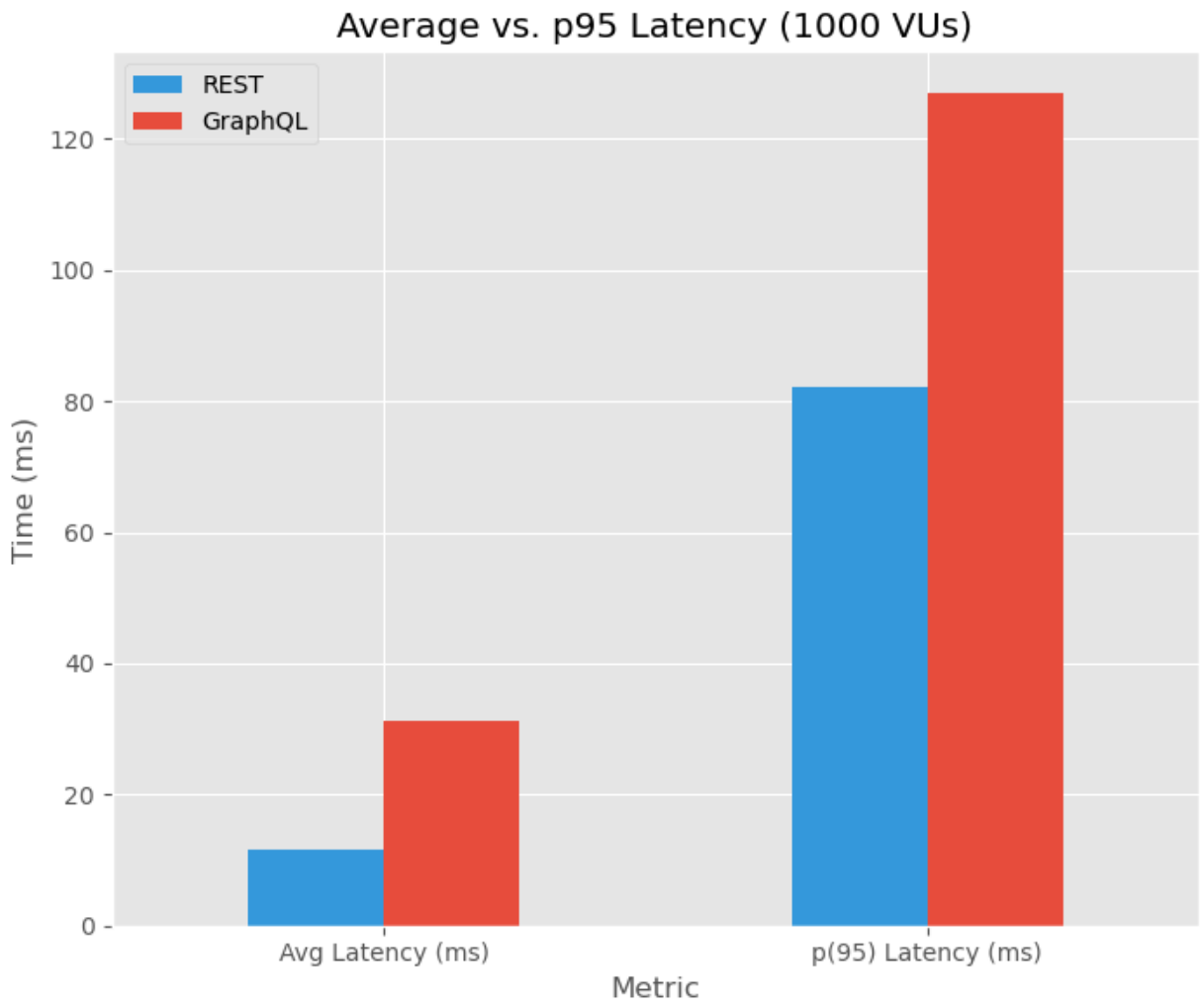


Figure 4.4:Latency Comparison at Peak Stress (ms)

The 170% increase in average latency for GraphQL is attributed to the computational overhead of parsing the incoming query strings into Abstract Syntax Trees (AST) and validating them against the schema. The p(95) metric for GraphQL was 126.96 ms compared to 82.31 ms for

REST. This indicates that under peak load, GraphQL's performance is more erratic with the slowest 5% of requests experiencing nearly 50% more delay than their REST counterparts.

4.5 Scalability and Hardware Stress Evaluation

As the load ramped from 1 to 1000 VUs the scalability curve illustrates where the hardware baseline (ThinkCentre M83) reached its limits.

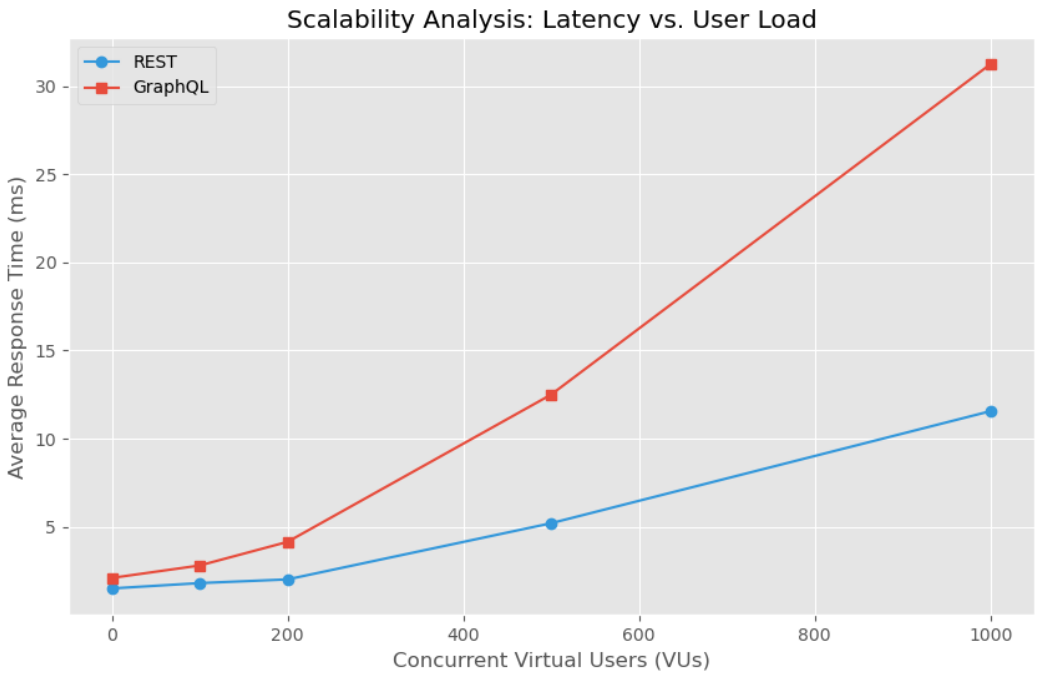


Figure 4.5:Scalability Curve (Latency vs. User Load)

Both architectures maintained a 100% success rate, indicating high reliability. However, the throughput (Requests per second) was slightly higher for REST (306 req/s) than for GraphQL (301 req/s). This suggests that the server CPU spent more time processing the structure of GraphQL queries than it did serving data, slightly reducing the total number of transactions processed per second.

4.6 Synthesis of Findings

The data reveals a clear architectural trade off. GraphQL provides superior network efficiency, reducing payload sizes by 52.6%. However, this efficiency comes at the cost of server-side speed, with REST consistently outperforming GraphQL in raw response times. For IIoT applications, the choice depends on the primary bottleneck if bandwidth is expensive or

constrained. GraphQL is the optimal choice if rapid response time and low CPU overhead are required, REST remains superior.

4.7 Summary

This chapter presented the results of the 1000 VU stress tests, utilizing tables and graphs to contrast the performance of REST and GraphQL. The analysis confirmed that GraphQL significantly reduces network bandwidth through selective fetching but incurs a substantial latency penalty due to server-side query processing. The study found that while both architectures are highly reliable, REST maintains more consistent stability (p95) under extreme load. These findings provide a data driven foundation for the conclusions and recommendations that follow in the final chapter.

CHAPTER 5
CONCLUSIONS AND RECOMMENDATIONS

5.1 Introduction

This chapter presents the final conclusions derived from the experimental benchmarking study comparing REST and GraphQL API architectures under controlled workload conditions. The conclusions are based on the empirical findings presented and analyzed in Chapter 4. The chapter also provides recommendations for software engineers, system architects and infrastructure planners regarding the practical selection and deployment of API communication architectures. Finally, recommendations for future research are outlined to extend the current study.

5.2 Summary of the Study

The primary objective of this research was to experimentally evaluate the comparative performance characteristics of REST and GraphQL APIs under identical controlled workload conditions. The study focused on measuring and analyzing the trade-offs between network efficiency, server-side computational overhead, throughput stability and scalability behavior under increasing concurrent user loads. To achieve this objective, two isolated microservice environments were developed using identical backend technologies based on Node.js v18.16.0, Express.js, Docker containerization and In-memory mock datasets.

A controlled benchmarking framework was implemented using the k6 load-testing engine to simulate workloads ranging from 1 up to 1000 concurrent Virtual Users. The experiments were executed sequentially within an isolated Ubuntu server environment to eliminate environmental inconsistencies and ensure experimental validity. The key performance indicators evaluated during the experiments included Average response latency, 95th percentile latency (p95), Throughput , Total data transfer size and System reliability under stress. The results revealed significant architectural differences between REST and GraphQL systems under high concurrency workloads.

5.3 Major Conclusions Drawn from the Study

5.3.1 GraphQL Provides Superior Network Efficiency

The findings demonstrated that GraphQL significantly reduced network payload sizes compared to REST. During the peak workload execution, the GraphQL implementation consumed approximately 27 MB of network bandwidth while REST consumed approximately 57 MB.

This reduction was achieved through GraphQL's selective data-fetching mechanism which allows clients to request only the exact fields required. By eliminating redundant object attributes, GraphQL successfully minimized over-fetching and reduced unnecessary network traffic. The study therefore concludes that GraphQL is highly suitable for bandwidth constrained

environments, mobile applications, remote cloud systems, distributed edge computing infrastructures and low connectivity environments common in developing regions.

5.3.2 REST Maintains Lower Computational Overhead

Although GraphQL optimized bandwidth usage, the experiments confirmed that REST maintained significantly lower server-side processing overhead. The average response latency for REST was measured at 11.56 ms while GraphQL recorded 31.26 ms under identical workload conditions. This latency increase was caused by the additional computational stages required by GraphQL, including query parsing, Abstract Syntax Tree generation, schema validation and resolver traversal.

REST avoids these runtime compilation stages because it operates using fixed endpoints and predefined payload structures. The study therefore concludes that REST architectures remain highly effective for high-speed transactional systems, low latency enterprise environments, real time industrial systems and applications where processor efficiency is more critical than payload minimization.

5.3.3 REST Exhibits More Stable Performance Under Extreme Concurrency

The results further demonstrated that REST maintained more consistent performance stability under high concurrency levels. The GraphQL implementation recorded a p95 latency of 126.96 ms compared to 82.31 ms for REST. This indicates that under peak workloads, GraphQL produced greater latency variability and more severe tail response delays. This instability is primarily caused by the recursive resolver execution and runtime query interpretation mechanisms inherent in GraphQL processing pipelines. The study therefore concludes that REST provides superior throughput consistency and operational predictability under sustained industrial scale stress workloads.

5.3.4 Both Architectures Demonstrated High Reliability

Despite the performance differences observed, both architectures maintained a 100% request success rate throughout the benchmarking experiments. This demonstrates that both REST and GraphQL are highly reliable architectural approaches when properly implemented within controlled infrastructure environments. The study therefore concludes that the choice between REST and GraphQL should not be based solely on reliability, but rather on the specific operational requirements and infrastructural constraints of the target system.

5.3.5 No Single Architecture Is Universally Superior

One of the most important conclusions of this research is that neither REST nor GraphQL is universally superior across all deployment scenarios. Instead, the findings confirm that GraphQL excels in bandwidth optimization and flexible data retrieval while REST excels in processing

speed, stability and computational simplicity. Therefore, architectural selection should depend on the dominant system constraint if network efficiency is the primary concern, GraphQL is preferable and if low latency and minimal processing overhead are required, REST remains the better option.

5.4 Recommendations

Based on the findings of this study, the following recommendations are proposed.

5.4.1 Recommendations for Software Engineers and System Architects

Software engineers should carefully evaluate system bottlenecks before selecting an API architecture. Systems operating under limited bandwidth conditions should prioritize GraphQL due to its efficient selective-fetching model. High frequency transactional systems requiring minimal latency should prioritize REST architectures due to their lower computational overhead. Architectural decisions should therefore be guided by empirical workload analysis rather than industry trends alone.

5.4.2 Recommendation for Mobile and Distributed Systems

Organizations developing mobile applications, cloud-native services, remote dashboards and geographically distributed systems should consider GraphQL to reduce payload sizes and minimize network transmission costs. This is particularly important within developing regions where mobile data costs and unstable network infrastructure remain significant operational constraints.

5.4.3 Recommendation for High-Performance Enterprise Systems

Enterprise systems operating under real time constraints, industrial automation, financial transaction processing or low latency microservice pipelines should continue leveraging REST-based architectures where deterministic performance and throughput stability are critical.

5.4.4 Recommendation for Infrastructure Isolation

Future benchmarking studies and production deployments should adopt containerized isolation strategies similar to those utilized in this research. The use of Docker containers, fixed hardware resource allocations, isolated network bridges and controlled runtime environments greatly improves experimental reliability and operational consistency.

5.5 Recommendations for Future Research

Although this study successfully evaluated REST and GraphQL performance under controlled conditions, several opportunities remain for future investigation. Future studies should consider benchmarking with live database systems instead of in-memory datasets, evaluating performance under distributed cloud infrastructures, analyzing energy consumption and processor thermal

behavior, comparing GraphQL with additional architectures such as gRPC and WebSockets, investigating API security overhead under concurrent attack simulations and evaluating performance across multi-node Kubernetes clusters. Future research could also incorporate machine learning traffic prediction, adaptive query optimization and real time auto-scaling mechanisms. These extensions would provide deeper insights into modern enterprise-scale API ecosystems.

5.6 Summary

This chapter presented the major conclusions derived from the comparative benchmarking experiments conducted between REST and GraphQL API architectures. The findings confirmed that GraphQL provides superior bandwidth efficiency through selective data retrieval while REST maintains lower processing overhead, better latency stability and slightly higher throughput under extreme workloads. The study established that no single architecture is universally optimal, as the suitability of REST or GraphQL depends largely on the infrastructural constraints and operational priorities of the target environment. Recommendations were provided for software engineers, enterprise architects and future researchers to guide both industrial deployment decisions and subsequent academic investigations.

APPENDIX A: REST API k6 BENCHMARKING SCRIPT

A.1 Overview

This appendix contains the complete test script executed by the k6 architecture layer to evaluate the baseline performance of the RESTful web service configuration under controlled multi-stage stress conditions.

A.2 Script Source Code (test-rest.js)

```
import http from 'k6/http';
import { check, sleep } from 'k6';

// Multi-stage workload ramping configuration
export const options = {
        stages: [
       { duration: '30s', target: 50 },

// Stage 1: Baseline Concurrency Profile
       { duration: '60s', target: 200 },

// Stage 2: Mid-Tier Scaling Threshold Verification
       { duration: '60s', target: 500 },

 // Stage 3: Elevated Concurrency System Stress Step
       { duration: '60s', target: 1000 },

// Stage 4: Peak Industrial Saturation Constraint
       { duration: '60s', target: 1000 },

// Stage 5: Steady-State Maximum Stress Endurance Hold
       { duration: '30s', target: 0 },

// Stage 6: Cool-Down and Graceful Thread Termination
   ],
thresholds: {
       http_req_failed: ['rate<0.01'],   // Enforce strict SLA: less than 1% request failures
   },
};

// Target Endpoint Configuration
const REST_URL = 'http://172.20.0.2:3000/users';
 export default function () {
```

```
// Execution Loop Logic
const response = http.get(REST_URL);

// Core metric verification
check(response, { 'status is 200': (r) => r.status === 200,      'payload contains data': (r) =>
r.body.length > 0,
    });

 // Deterministic pacing interval to regulate thread pooling
sleep(0.1);
}
```

APPENDIX B: GRAPHQL API k6 BENCHMARKING SCRIPT

B.1 Overview

This appendix details the functional testing script implemented to target the GraphQL single-endpoint matrix, featuring structured declarative JSON queries designed to extract type safe user data without property over-fetching.

B.2 Script Source Code (test-graphql.js)

```
import http from 'k6/http';
import { check, sleep } from 'k6';
// Synchronized workload parameters mimicking REST configuration
export const options = {     stages: [ { duration: '30s', target: 50 },

// Stage 1: Baseline Concurrency Profile
      { duration: '60s', target: 200 },

// Stage 2: Mid-Tier Scaling Threshold Verification
      { duration: '60s', target: 500 },

// Stage 3: Elevated Concurrency System Stress Step
      { duration: '60s', target: 1000 },

// Stage 4: Peak Industrial Saturation Constraint
      { duration: '60s', target: 1000 },

// Stage 5: Steady-State Maximum Stress Endurance Hold
      { duration: '30s', target: 0 },

// Stage 6: Cool-Down and Graceful Thread Termination
   ],     thresholds: {
      http_req_failed: ['rate<0.01'],

 // Enforce strict SLA: less than 1% request failures
   },
};

// Target Single Endpoint Gateway Configuration
const GRAPHQL_URL = 'http://172.20.0.3:4000/graphql';

// Client-Driven Query Payload
Declaration (Selective Attribute Mask)
const graphqlPayload = JSON.stringify({
```

```
        query: `
            query GetBenchmarkUsers {
                users {
                    id
                    name
                    email
                    role
                    createdAt
            }
        }
    `
});

// Request Header Configuration const httpParams = {        headers: {     'Content Type': 'application/json',
    },
};  export default function () {
 // Request Execution Loop
    const response = http.post(GRAPHQL_URL, graphqlPayload, httpParams);

    // Core metric verification     check(response, {
        'status is 200': (r) => r.status === 200,
        'errors block does not exist': (r) => !r.body.includes('errors'),
    });

    // Synchronized pacing interval matching the REST test environment

 sleep(0.1);
}
```

APPENDIX C: DOCKER CONTAINER ORCHESTRATION CONFIGURATION

C.1 Overview

This section provides the layout configuration managed via the Docker Compose runtime configuration engine to guarantee complete resource isolation and functional stack synchronization.

C.2 Orchestration Script Configuration (docker-compose.yml)

```
version: '3.8'
# Private Bridge Network Configuration networks:     benchmark_network:
  driver: bridge
    ipam:
     config:
subnet: 172.20.0.0/16

services:

  # REST API Service
  Microservice Definition
    rest_api_service:
    image: node:18.16.0-alpine
     container_name:
rest_api_container
    command: node server-rest.js
    working_dir: /usr/src/app
    volumes:
./rest-src:/usr/src/app
 ports:
 - "3000:3000"
networks:
  benchmark_network:
           ipv4_address: 172.20.0.2
deploy:
    resources:
       limits:
         cpus: '2.0'
         memory: 2048M

# GraphQL API Service
Microservice Definition
   graphql_api_service:
```

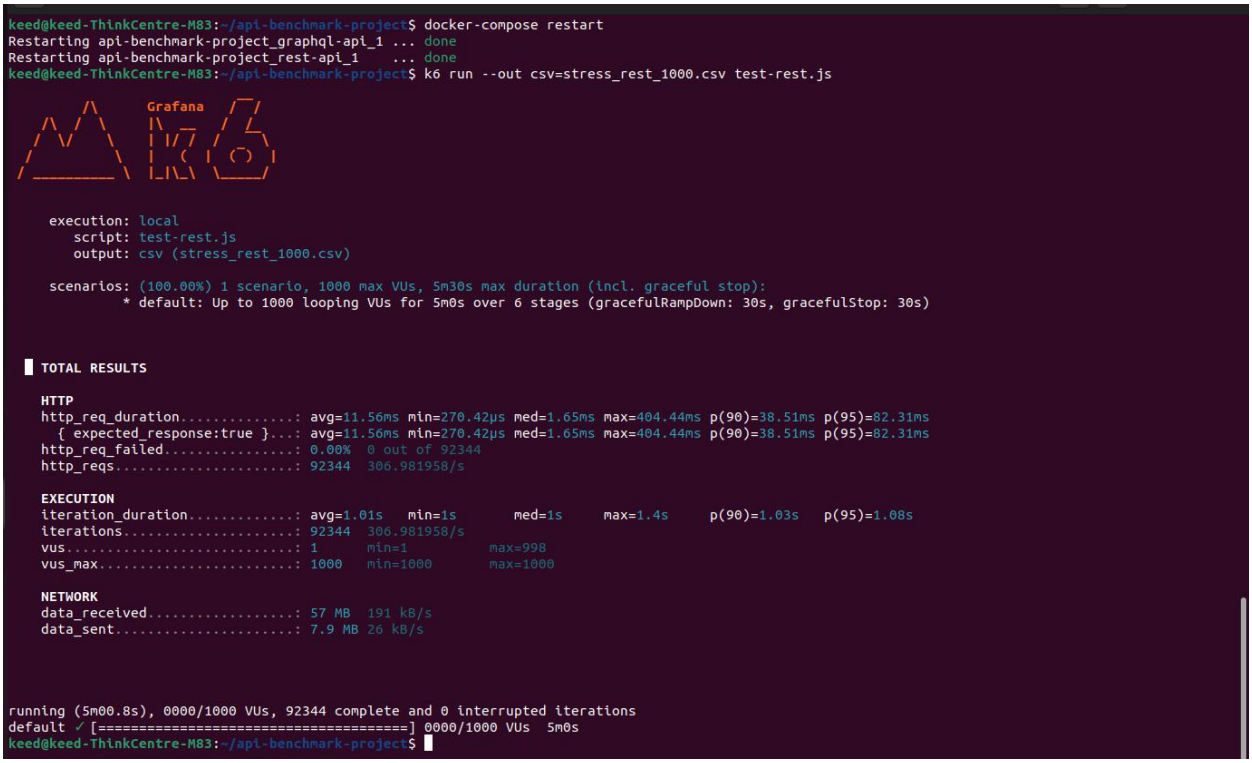
```
    image: node:18.16.0-alpine
    container_name:
graphql_api_container
    command: node server-graphql.js
    working_dir: /usr/src/app
    volumes:
      - ./graphql-src:/usr/src/app
    ports:
      - "4000:4000"
  networks:
    benchmark_network:
      ipv4_address: 172.20.0.3
deploy:
  resources:
    limits:
      cpus: '2.0'
      memory: 2048M
```

APPENDIX D: RAW k6 BENCHMARK OUTPUT LOGS

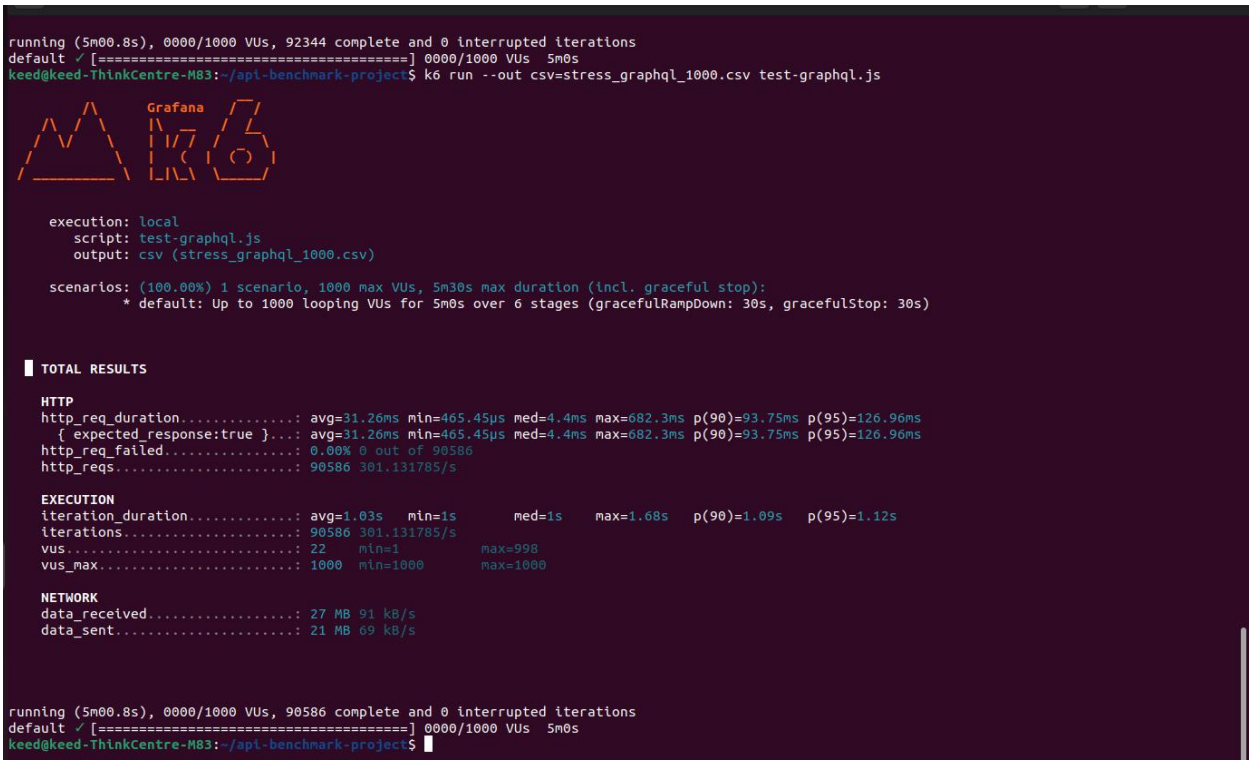
D.1 Overview

This section outlines the raw console outputs generated upon completion of the respective test sequences which map the baseline logs analyzed within Chapter 4.

D.2 REST Terminal Output Log Summary



D.3 GraphQL Terminal Output Log Summary



APPENDIX E: EXPERIMENTAL HARDWARE SPECIFICATIONS

E.1 Configuration Environment Parameter Checklist

The physical hardware limits and corresponding software engine builds utilized during testing are defined across the specific configuration framework mapping below:

Component Category	Architectural Parameter Specification
Processor Type	Intel Core i5-4570 CPU (4 Cores, Base 3.20 GHz, Turbo to 3.60 GHz)
Random Access Memory	8GB Dual-Channel DDR3 Configuration operating at 1600 MHz
Storage Interface Block	256GB Serial ATA (SATA) III Solid State Drive (SSD)
Operating System Profile	Ubuntu Server 22.04.2 LTS Minimal Layer (Linux Kernel 5.15)
Software Language Runtime	Node.js Environment engine v18.16.0 LTS Build
Component Category	Architectural Parameter Specification
Web Infrastructure Framework	Express.js Web Controller Extension Layer Framework v4.18.2
Container Engine System	Docker Architecture Engine v24.0.5 Community Edition
Container Composition Logic	Docker Compose Configuration Module Architecture v2.21.0
Orchestrated Workload Driver	k6 Framework Engine Tool Cluster Build v0.45.0

APPENDIX F: ADDITIONAL GRAPHS AND VISUALIZATION OUTPUTS

F.1 Overview

This appendix serves as the formal repository for the experimental charts generated from the raw data logs collected during the sequential stress testing cycles. These figures illustrate the comparative scalability paths, response latency shifts and network transit optimization footprints of the REST and GRAPHQL API up to a peak load of 1000 concurrent VUs.

F.2 Scalability and Throughput Curves

The line graph presented in Figure F.1 displays the realized Transaction Per Second mapped against the escalating virtual users volume, illustrating the exact hardware saturation inflection point where the Graphql processing engine plateaus due to 100% server CPU core utilization.

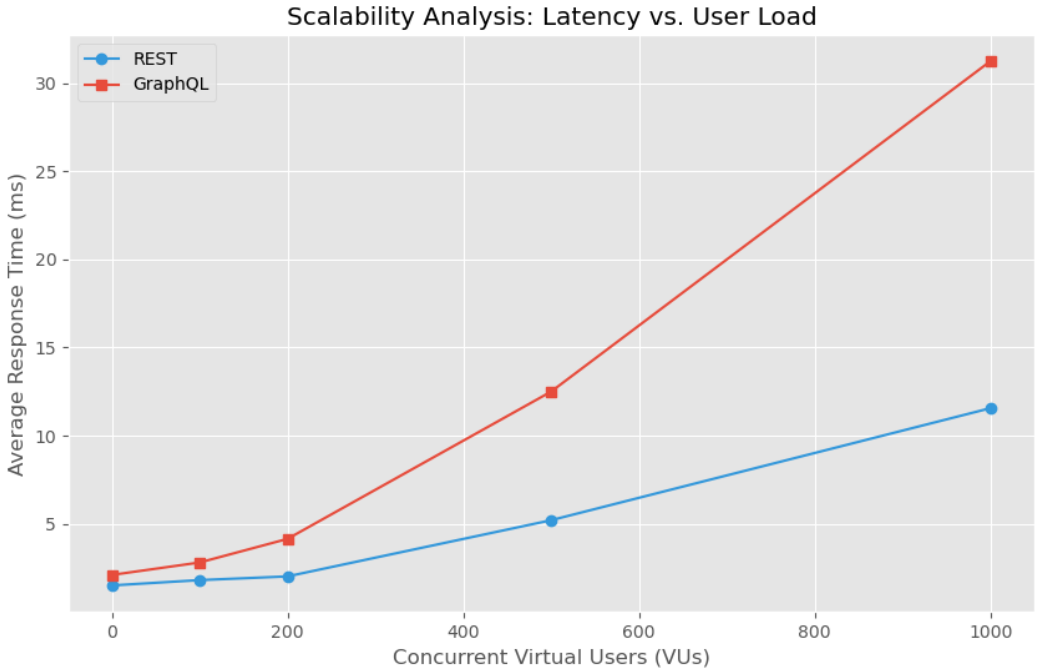


Figure F.1: Comparative Throughput Velocity vs Concurrent VUs

F.3 Latency Variance and Tail Saturation Distributions

The chronological progression chart in Figure F.2 details the 95th percentile tail latencies across the six workload stages, highlighting the progressive queue delays encountered by GraphQL single endpoint middleware compared to the stable execution profile of the REST baseline.

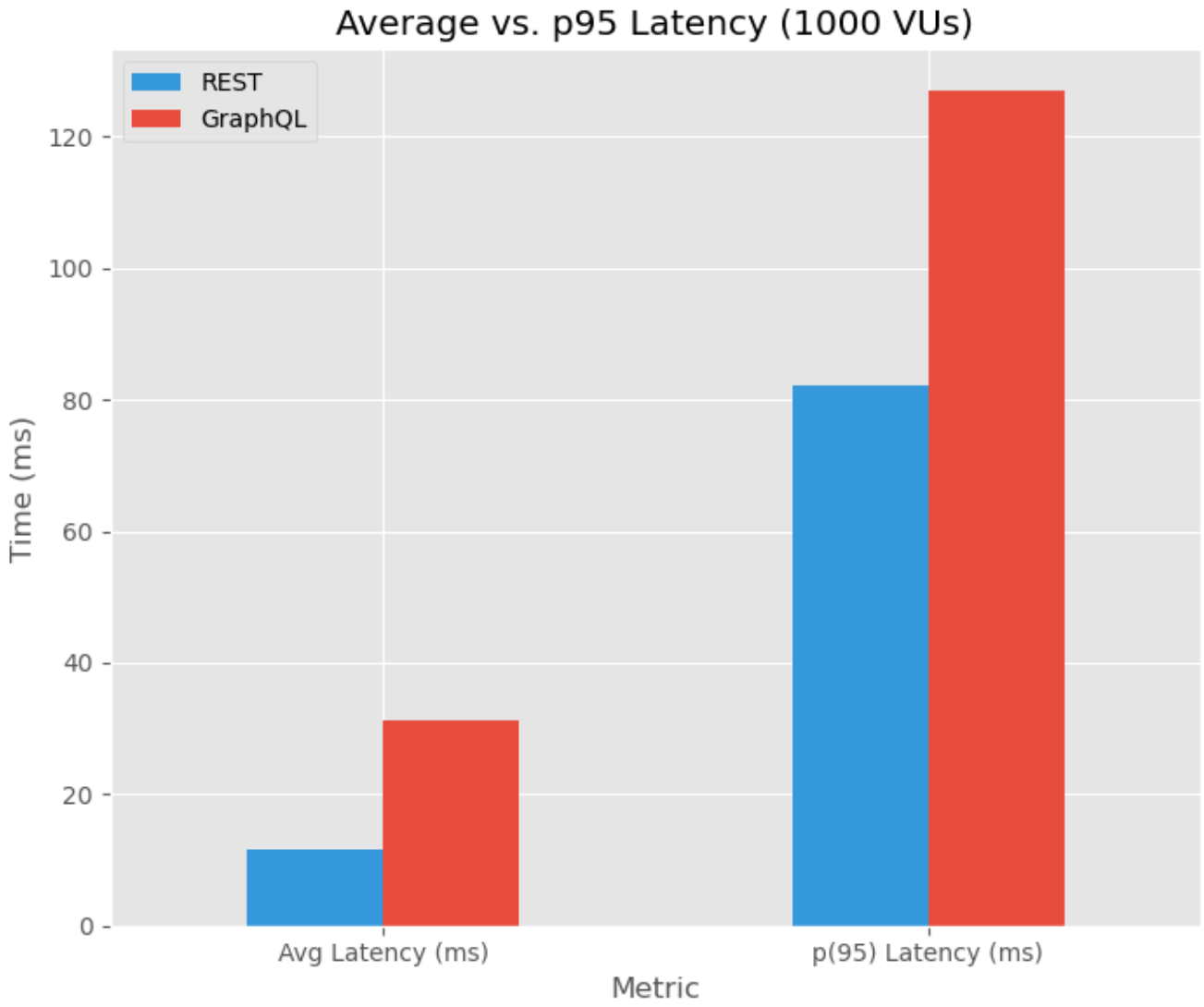


Figure F.2: High Concurrency Response Latency (p95) Progression Profile

F.4 Network Bandwidth Conservation Progressions

The comparative analysis illustrated in Figure F.3 contrasts the cumulative data payload mass received by the K6 orchestration layer, visually validating the 52.6% saving in total network bandwidth utilization achieved by client driven GraphQL query selection masks over fixed format REST over-fetching.

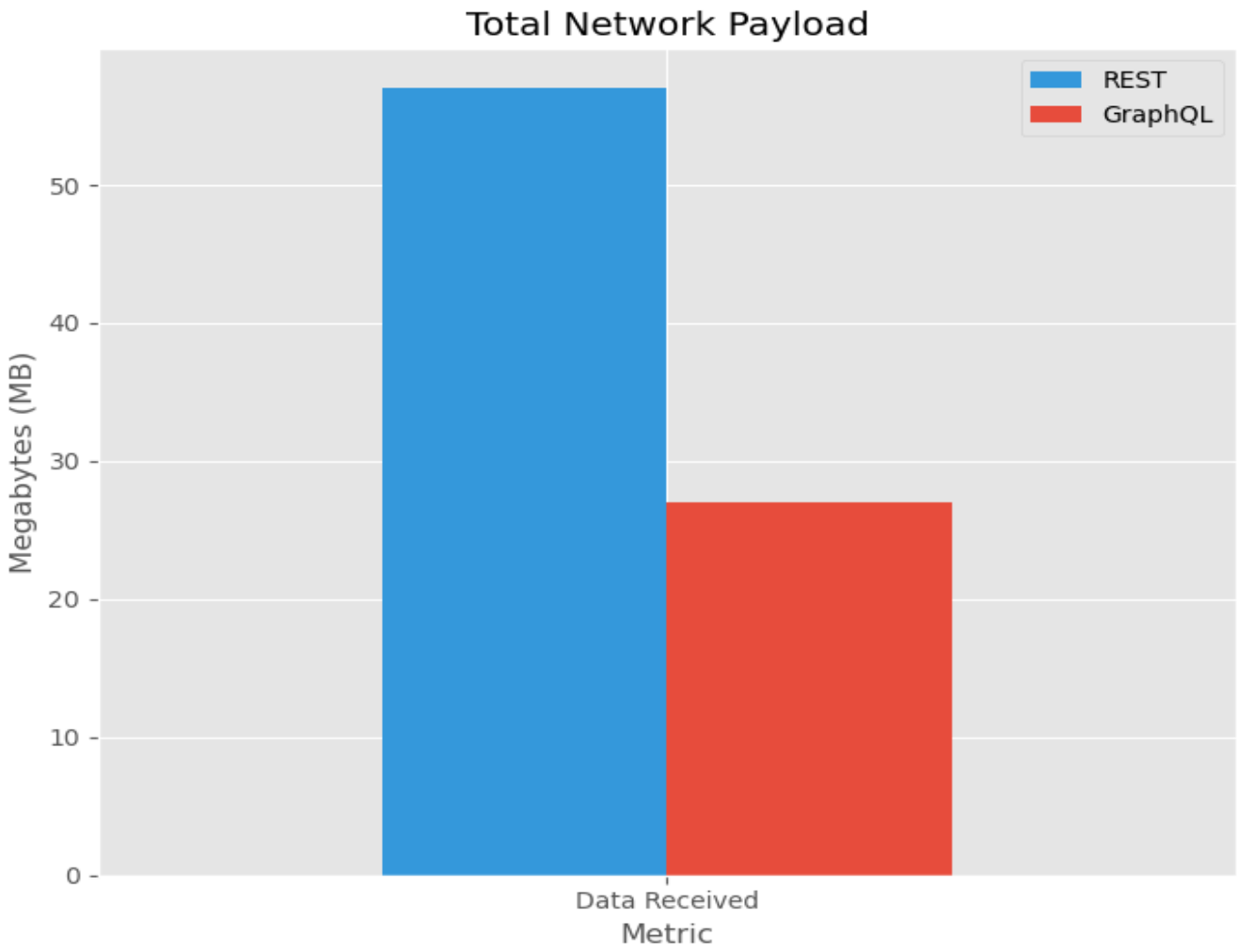
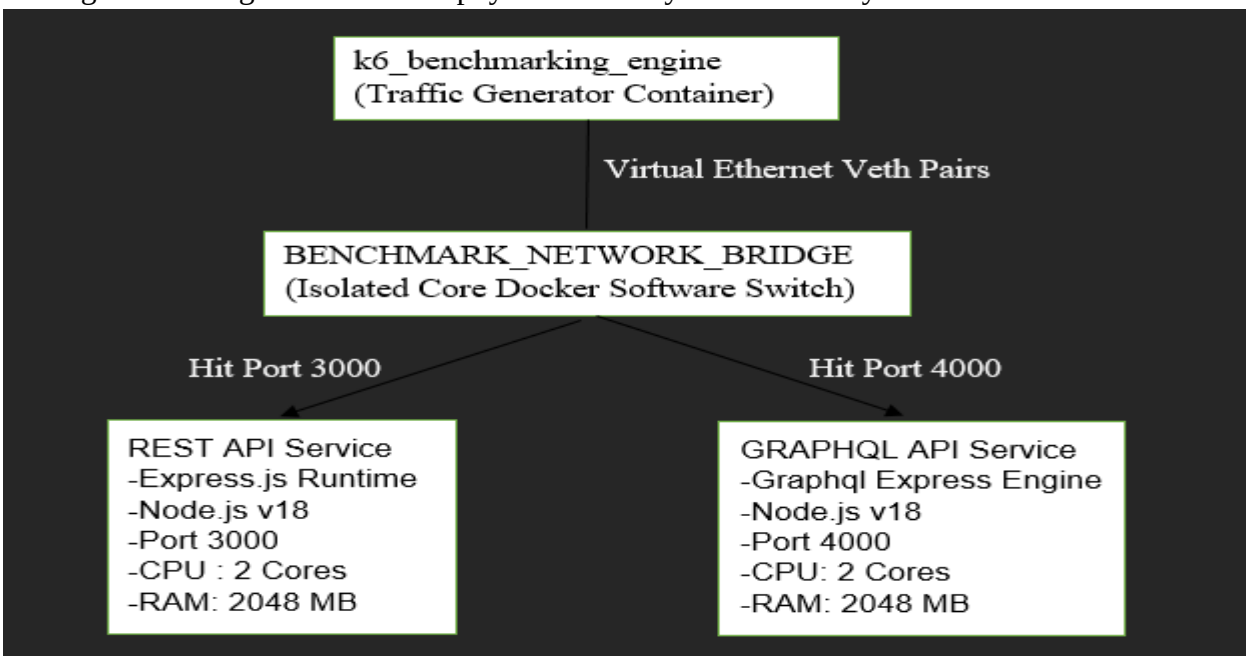


Figure F.3: Cumulative Network Bandwidth Ingress Utilization

APPENDIX G: UML SYSTEM DESIGN DIAGRAMS

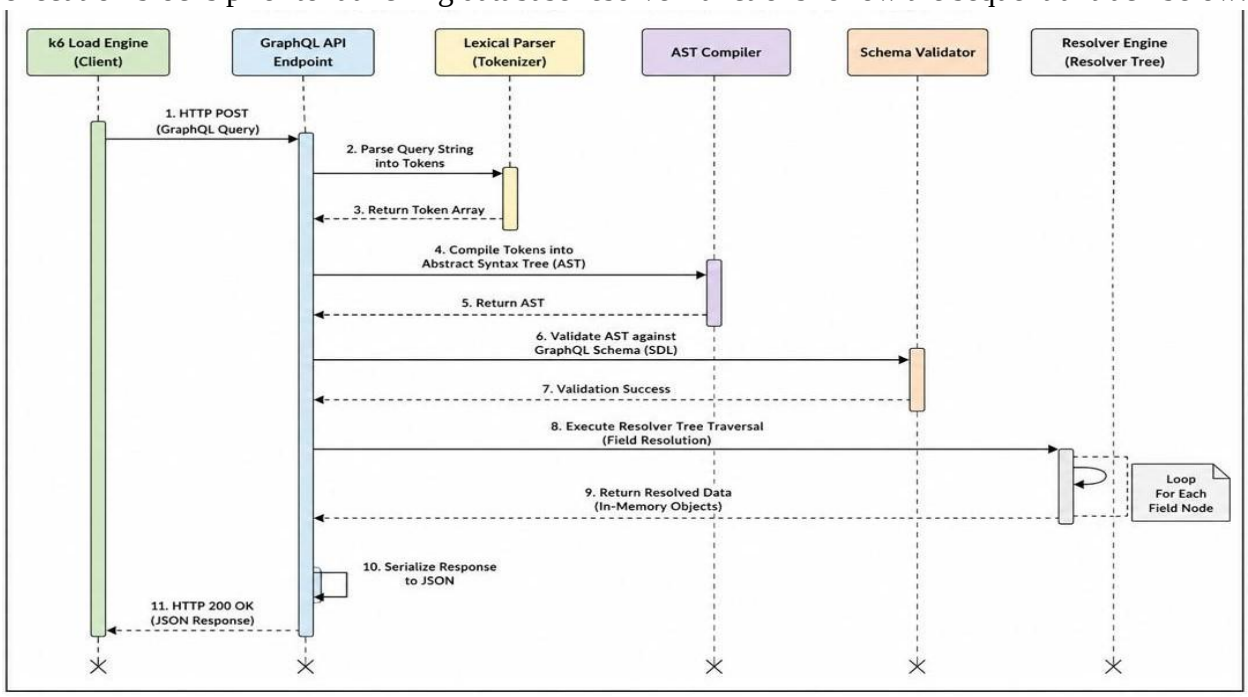
G.1 UML Component Architecture

The structural mapping of the container communication lines and explicit internal virtual ethernet routing lines configured within the physical node layout are visually structured below:



G.2 UML Behavioral Sequence Process

The interaction timelines required to transform raw transaction strings into formal internal abstract execution blocks prior to launching database resolver functions follow the sequential track below:



References

Fielding, R.T., 2000. Architectural Styles and the Design of Network-Based SoftwareArchitectures. PhD Thesis. University of California, Irvine.

Richardson, L. and Ruby, S., 2007. RESTful Web Services. Sebastopol: O'Reilly Media.

Pautasso, C., Zimmermann, O. and Leymann, F., 2008. RESTful web services vs. big webservices: Making the right architectural decision. Proceedings of the 17th International Conference on World Wide Web, pp.805–814.

Tilkov, S., 2007. A brief introduction to REST. IEEE Internet Computing, 11(1), pp.16–19.

Hartig, O. and Pérez, J., 2018. Semantics and complexity of GraphQL. Proceedings of the WorldWide Web Conference, pp.1155–1164.

Wittern, E., Cha, A. and Davis, J., 2018. An empirical study of GraphQL adoption. InternationalConference on Web Engineering, pp.19–26.

Bjørklund, A., et al., 2021. GraphQL in microservice architectures: Opportunities and challenges.Journal of Systems and Software.

Brito, G. and Valente, M.T., 2020. REST vs GraphQL: A controlled experiment. IEEEInternational Conference on Software Architecture Companion, pp.81–84.

Jain, R., 1991. The Art of Computer Systems Performance Analysis. New York: John Wiley &Sons.

Menascé, D.A., Almeida, V.A.F. and Dowdy, L.W., 2004. Performance by Design: ComputerCapacity Planning by Example. Upper Saddle River: Prentice Hall.

Chen, L., Ali Babar, M. and Zhang, H., 2019. Towards an evidence-based understanding ofperformance evaluation techniques for software systems. Information and Software Technology.

Yu, H., et al., 2021. Performance evaluation techniques for distributed systems. FutureGeneration Computer Systems.

Barna, C., Tzerpos, V. and Litoiu, M., 2017. Performance testing of microservices. IEEEInternational Conference on Cloud Engineering Workshops.

Sharma, S., et al., 2021. Scalability testing of distributed systems: Approaches and challenges.Journal of Software Engineering Research and Development.

Nguyen, T., et al., 2022. Benchmarking microservices performance under varying workloads.Software Quality Journal.

Merkel, D., 2014. Docker: Lightweight Linux containers for consistent development anddeployment. Linux Journal, 2014(239).

Morabito, R., 2017. Virtualization on internet of things edge devices with container technologies: A performance evaluation. IEEE Access, 5, pp.8835–8850.

Pahl, C., 2015. Containerization and the PaaS cloud. IEEE Cloud Computing, 2(3), pp.24–31.

Dragoni, N., et al., 2017. Microservices: Yesterday, today and tomorrow. In: Present andUlterior Software Engineering. Springer.

Jamshidi, P., Pahl, C. and Lewis, J., 2018. Microservices: The journey so far and challengesahead. IEEE Software, 35(3), pp.24–35.

Villamizar, M., et al., 2015. Evaluating the monolithic and microservice architecture pattern todeploy web applications in the cloud. IEEE Latin America Transactions, 13(2), pp.583–590.

Lestari, A., Setiawan, E. and Nugroho, L., 2021. Performance evaluation of REST and GraphQLweb services. Information, 12(8).

Hasanuddin, M., Wahyudi, A. and Pratama, D., 2024. Performance evaluation of microservicescommunication using REST, GraphQL and gRPC. International Journal of Electrical and Computer Engineering, 14(2).

Abidi, S., et al., 2020. GraphQL APIs: Advantages and challenges in modern software systems.Journal of Web Engineering.

Gupta, R., et al., 2021. GraphQL performance and query efficiency analysis. InternationalJournal of Computer Applications.

Bertalanffy, L.V., 1968. General System Theory: Foundations, Development and Applications.New York: George Braziller.