OPTIMIZING WSO2 API MANAGER FOR IMPROVED GRAPHQL QUERY PERFORMANCE IN A DISTRIBUTED SETUP

STANLEY MAKORI ONDERO

MASTER OF SCIENCE

(Computer Systems)

JOMO KENYATTA UNIVERSITY

OF

AGRICULTURE AND TECHNOLOGY

2025

Optimizing Wso2 Api Manager for Improved Graphql Query Performance in a Distributed Setup

Stanley Makori Ondero

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Systems of the Jomo Kenyatta University of Agriculture and Technology

2025

DECLARATION

This thesis is my original work and has not been presented for a degree in any other University

Signature..Date……………………………

Stanley Makori Ondero

This thesis has been submitted for examination with my approval as the University Supervisors

Signature..Date..

Dr. Mwangi Karanja, PhD

JKUAT, Kenya

Signature...Date...

Dr. Dennis Kaburu, PhD

JKUAT, Kenya

DEDICATION

This thesis is dedicated to my beloved family, whose unwavering support and encouragement have been the cornerstone of my academic journey. To my parents, who instilled in me the values of hard work, perseverance, and the pursuit of knowledge, and who made countless sacrifices to ensure I could reach this milestone. To my spouse and children, who patiently endured the long hours of research and writing, offering understanding when studies took precedence over family time, and whose love sustained me through the challenges of this academic pursuit.

To my mentors and lecturers at Jomo Kenyatta University of Agriculture and Technology, who shared their wisdom and expertise, challenged my thinking, and guided me toward excellence in the field of computer systems.

To my fellow graduate students and colleagues, who provided intellectual stimulation, collaborative support, and friendship throughout this journey, making the rigorous demands of graduate study more bearable and rewarding.

To my colleagues at Safaricom PLC, who consistently encouraged me to persevere and provided valuable insights from their professional experience, demonstrating that continuous learning and academic excellence can coexist with professional responsibilities.

And finally, to all those who believe in the transformative power of technology and its potential to solve real-world problems and improve lives. May this work contribute, however modestly, to the advancement of knowledge in computer science and inspire others to pursue excellence in this vital field.

"The best way to predict the future is to create it."

ACKNOWELDGEMENT

I wish to express my sincere gratitude to all those who contributed to the successful completion of this thesis.

First and foremost, I extend my deepest appreciation to my supervisors, Dr. Mwangi Karanja and Dr. Dennis Kaburu, for their exceptional guidance, constructive feedback, and unwavering support throughout the research process. Their expertise in computer systems and commitment to academic excellence have been instrumental in shaping this work. I am particularly grateful for their patience in reviewing multiple drafts and their insightful suggestions that significantly improved the quality of this thesis.

I would like to thank Professor Cheruiyot for his valuable contributions and scholarly insights that enriched this research. His expertise helped strengthen both the theoretical foundation and practical applications of this work.

My sincere gratitude goes to the faculty and staff of the Department of Computer Science at Jomo Kenyatta University of Agriculture and Technology for providing an intellectually stimulating environment and necessary resources. I extend special appreciation to Mercy Kagunya, Senior Administrator in the Computer Science Department, whose diligent guidance on administrative processes and meticulous attention to ensuring all documentation was properly filed was invaluable. Special thanks also to the technical staff who assisted with laboratory facilities and equipment.

I acknowledge the support of my Safaricom PLC colleagues, who encouraged my academic pursuits and provided practical insights from industry experience that enhanced the real-world relevance of this research.

Finally, I am deeply grateful to my family for their patience, understanding, and continuous encouragement throughout this demanding academic endeavour. Their sacrifices and emotional support made this achievement possible.

TABLE OF CONTENTS

DECLARATION…………………………………………………………………..….ii

DEDICATION……………………………………………………………………...iii

ACKNOWELDGEMENT……………………………...………………………….iv

TABLE OF CONTENTS…...……………………………………………………… v

LIST OF TABLES…………………………………………………………….....xi

LIST OF FIGURES…………………………………………………………….xii

ABBREVIATIONS AND SYMBOLS………...………………………………xiii

ABSTRACT…………………………………………………………………..xiv

CHAPTER ONE……..………………………………………………………..1

INTRODUCTION…………...…………………………………………………...1

1.1 Background 1

1.2 VMware.. 3

1.3 Cloud Services 3

1.4 Security and Networking 3

1.5 Global Impact.. 4

1.6 Top Enterprise API Management Systems 4

1.7 Statement of the Problem 5

1.8 Objectives 5

1.8.1 Broad Objective 5

1.8.2 Specific Objectives 6

1.8.3 Research Questions 6

1.9 Significance of the Research 6

1.10 Scope of Study 7

CHAPTER TWO 8

LITERATURE REVIEW 8

2.1 Introduction 8

2.2 Application Programming Interfaces 8

2.3 API Management 8

2.4 Importance of Optimizing GraphQL and WSO2 API Manager 9

2.5 Available API Optimization Techniques for API Managers and GraphQL 10

2.5.1 Caching 10

2.5.2 Bundle-splitting 11

2.5.3 Query Merging 11

2.5.4 Query Splitting 12

2.5.5 Query Pruning 13

2.5.6 Content Compression 13

2.5.7 Query Batching 13
2.6 WSO2 API Manager Performance 14
2.6.1 Operating System Optimization 14
2.6.2 Database Connection Pooling 15
2.6.3 Backend Choice Rationale 15
2.6.4 Justification for Heap and JVM Settings 16
2.7 Synthesis of Related Work 17
2.8 Research Gaps Identified 19
2.9 Justification 20
2.10 APIM-OM Conceptual Model 20
2.10.1 Scope of APIM-OM Model 21
2.10.2 Evaluation of APIM-OM Model 21
CHAPTER THREE..22
RESEARCH METHODOLOGY..22
3.1 Introduction 22
3.2 Research Design 22
3.3 Research Environment 22
3.3.1 Hardware Specifications 23
3.4 Test Case Selection 23

3.4.1 Real-World API Scenario 24

3.4.2 Query Payload Design 24

3.5 Performance Metrics and Measurement 24

3.5.1 Key Performance Indicators 25

3.5.2 Load Testing Parameters 25

3.6 Implementation of Optimization Techniques 25

3.6.1 Transport Layer Optimization (Server Tuning) 25

3.6.2 Application Layer Optimization (Query Splitting) 26

3.6.3 Query Splitting Pseudocode 29

3.6.4 Data Layer Optimization (Redis Caching) 30

3.7 Data Collection Methodology 31

3.8 Data Quality Assurance 31

3.9 Ethical Considerations 32

3.9.1 Comparative Analysis Framework 32

3.9.2 Statistical Validation 32

3.9.3: Visualization and Interpretation 33

3.9.4 Internal Validity 33

3.9.5 External Validity 33

3.9.6 Reliability 34

CHAPTER FOUR……………………………………………………………..35

RESULTS, ANALYSIS AND DISCUSSIONS……………………………...35

4.1 The Experiment 35

4.2 Performance before Any Optimizations 35

4.3 Performance after JVM and Heap Optimization 38

4.3.1 Query Splitting Results 39

4.3.2 Performance Behaviour after Caching 41

4.3.3 Discussions of the Data Findings 43

CHAPTER FIVE……………………………...……………………...45

CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK…….45

5.1 Introduction 45

5.2 Summary 45

5.3 Conclusions 46

5.3.1 Objective I: Investigate the Performance Challenges of GraphQL Queries in WSO2 API Manager 46

5.3.2 Objective II: Develop an Optimization Model to Enhance GraphQL Query Performance 46

5.3.3 Objective III: Evaluate the Effectiveness of the Proposed Optimization Model 47

5.4 Contributions and Implications of this Research 47

5.5 Recommendations for Future Work 49

REFERENCES……………………………………………………………………..50

LIST OF TABLES

Table 2.1: Operating System Level Optimization.. 14

Table 3.1: Sever Metrics 23

Table 3.*2*: Performance Measurement Metrics 25

Table 4.1: Performance Data before JVM and Heap Tuning................................... 36

Table 4.2: G1GC Algorithm and Heap Optimization 39

Table 4.3: Results after Query Splitting Algorithm 40

Table 4.4: Results after Caching 42

LIST OF FIGURES

Figure 2.1: Concept Model 20

Figure 3.1: Transport Layer Optimization 26

Figure 3.2: Payload before Splitting 27

Figure 3.3: Payload after Splitting 28

Figure 3.4: Sender Details after Splitting.. 28

Figure 3.5: Sample Receiver Details after Splitting 28

Figure 3.7: Data Layer Optimization 30

Figure 4.1: Error Rate versus Concurrent Users 37

Figure 4.2: Concurrent Users versus Error Rate 38

Figure 4.3: Throughput versus Concurrent Users 41

Figure 4.4: Throughput versus Response Time 43

ABBREVIATIONS AND SYMBOLS

AST -	Abstract Syntax Tree
CBK -	Central Bank of Kenya
G1GC -	Garbage First Garbage Collector Tuning
GC -	Garbage Collection
JSON -	JavaScript Object Notation
JWT -	Json Web Tokens
JVM -	Java Virtual Machine
OAUTH2 -	Open Authorization
PAAS -	Platform as a service
REST -	Representational State Transfer
SAAS -	Software as a Service
SDK -	Software Development Kit
SMEs -	Small and medium enterprises
SOA -	Service Oriented Architecture
SOAP -	Simple Object Access Protocol
SSO -	Single sign on
TCP -	Transmission control protocol
UDP -	User datagram protocol
WSO2 -	An Open-source API Management system
XML -	Extensible Markup Language
API –	Application Programming Interface

ABSTRACT

In the rapidly evolving landscape of API management, GraphQL has emerged as a powerful query language that enables efficient data retrieval. However, optimizing GraphQL query performance within the context of distributed WSO2 API Manager remains a significant challenge. This research aimed to address this challenge by investigating the performance bottlenecks of GraphQL queries in WSO2 API Manager and proposing a comprehensive optimization model. The study begins by conducting a thorough literature review to identify existing performance challenges and potential optimization techniques. Through empirical testing and analysis, a baseline performance profile is established, highlighting key areas for improvement. The proposed optimization model encompasses three main components: JVM optimization, query splitting algorithm, and caching mechanisms. JVM optimization focuses on tuning parameters such as heap size and garbage collection settings to enhance resource utilization. The query splitting algorithm which exploits parallel execution to enhance performance was applied. Caching mechanisms, implemented using Redis, enable efficient storage and retrieval of frequently accessed data, minimizing redundant database queries. The effectiveness of the optimization model was evaluated using the M-PESA Payment API as a real-world test case. Experiments were conducted under various concurrency levels, measuring key performance metrics such as response time, throughput, and error rate. The optimized performance was compared against the baseline, revealing significant improvements. The model achieves an average response time reduction of 4.62% and throughput growth of 5.89%, demonstrating its efficacy in enhancing GraphQL query performance. This study evaluates three complementary techniques in a distributed WSO2 API Manager: (i) server-level optimization (OS TCP tuning and JVM G1GC with fixed heap), (ii) query splitting for parallel sub-resolution of GraphQL fields, and (iii) Redis-based distributed caching for tokens and hot query results.

CHAPTER ONE

INTRODUCTION

1.1 Background

There is an accelerated growth in digital services for various industries. These are powered by a suite of web services and APIs (Odun-Ayo et al., 2018). APIs and web services have been around since early 90s (Shyam, 2020). To fully realize the potential value of APIs, organizations should understand their usage, potential business models and monetization strategies should be present (Mantziou et al., 2023). Leading organizations in the API economy such as WSO2, Oracle with their Fusion Middleware API Manager, Google with Apigee, Amazon Web Service with Amazon API Gateway, Red hat with their API Connect and many others, have created API management platforms to expose their assets as well as sell to their partners (Meng et al., 2020).

API management platform has multiple components (Borissova & Mustakerov, 2016). This includes an API Gateway where runtime policies such us throttling and payload size are created and enforced, a Security component for access and API key management, a Developer Portal which provides a centralized location for developers to discover, subscribe to and test APIs and generate their access keys (consumer key and secret key), an API Publisher designing and publishing APIs, an API Analytics intelligence on API usage, security threats, monitoring and streaming of traffic etc (Goel et al., 2017).

WSO2 API Manager is a complete, enterprise-ready solution for managing APIs across the complete API lifecycle. It provides a wide range of features for optimizing API performance, security, and governance (Morra et al., 2019). GraphQL is a query language for APIs that allows clients to request exactly the data they need, and nothing more. This can lead to significant performance improvements, especially for complex queries (Fauvel et al., 2022).

WSO2 API Manager provides several ways to optimize GraphQL APIs, including:

1. Rate limiting: WSO2 API Manager can be used to limit the number of requests that a client can make to a GraphQL API in each period. This can help to prevent the API from being overloaded.
2. Query depth and complexity limits: WSO2 API Manager can also be used to limit the depth and complexity of GraphQL queries. This can help to prevent the API from being used to perform malicious or computationally expensive operations.
3. Caching: WSO2 API Manager can be used to cache the results of GraphQL queries. This can improve performance by reducing the number of times that the backend API needs to be called.
4. Load balancing: WSO2 API Manager can be used to load balance traffic across multiple instances of a GraphQL API. This can improve performance and scalability.
5. In addition to these features, WSO2 API Manager also provides a few other features that can be used to optimize GraphQL APIs, such as:
    a. API authentication and authorization: WSO2 API Manager can be used to authenticate and authorize users before they are allowed to access GraphQL APIs. This can help to improve security and prevent unauthorized access.
    b. API analytics: WSO2 API Manager can be used to collect and analyze API usage data. This data can be used to identify performance bottlenecks and areas for improvement.
    c. API monitoring: WSO2 API Manager can be used to monitor GraphQL APIs for performance and availability problems. This information can be used to proactively resolve issues before they impact users.
    d. Overall, WSO2 API Manager provides a comprehensive set of features for optimizing GraphQL APIs. By using these features, organizations can improve the performance, security, and scalability of their GraphQL APIs.

1.2 VMware

VMware is a prominent global leader in virtualization and cloud computing solutions, offering a diverse range of products and services to enhance IT infrastructure and operations. Founded in 1998, VMware has played a pivotal role in revolutionizing the way organizations manage and deploy their computing resources (Farheen & Raghuwanshi, 2021).

At the core of VMware's offerings is its virtualization technology, which enables the creation and management of virtual machines (VMs). Virtualization allows multiple operating systems and applications to run on a single physical server, optimizing resource utilization, improving scalability, and streamlining IT operations (Nian et al., 2021).

1.3 Cloud Services

VMware has made significant strides in cloud computing, offering solutions for both private and hybrid cloud deployments. VMware Cloud Foundation provides a comprehensive cloud infrastructure platform, and VMware Cloud on AWS extends VMware's capabilities to the Amazon Web Services (AWS) cloud (Azeroual et al., 2018).

Addressing the needs of the mobile workforce, VMware's Horizon Suite provides solutions for virtual desktop infrastructure (VDI) and enterprise mobility management (EMM). This enables organizations to securely deliver applications and data to end-users across various devices (Anggoro & Aziz, 2021).

1.4 Security and Networking

VMware NSX is a key component of the company's networking and security offerings. It provides software-defined networking, allowing organizations to create and manage virtualized networks with enhanced flexibility and security (Ghanavati et al., 2020).

VMware continues to innovate and adapt to evolving technological trends. The company collaborates with various technology partners and is actively involved in

open-source initiatives to contribute to the advancement of the IT industry (Cg & Cn, 2019).

1.5 Global Impact

With a global presence and a customer base that spans enterprises of all sizes, VMware has had a profound impact on the efficiency, agility, and scalability of IT infrastructures worldwide(Dhingra, 2025).

1.6 Top Enterprise API Management Systems

According to (Tejaswi Adusumilli, 2025), the following ARE the top API management platforms:

i) *Amazon API Gateway* –used purely for Amazon's webservices consumption, the gateway is described to be easy to use and easy to manage.

ii) *Google Apigee* – A commercial API management system. It can be provided both on premises and on cloud. Provides both web and backend services. Manufactured by google.

iii) *MuleSoft* – Commercial API gateway which provided both API publisher and enterprise service bus. Manufactured by MuleSoft LLC.

iv) *Microsoft Azure API management system* – A commercial API management system that is said to be fast in push and pull requests, from Microsoft Corporation.

v) *IBM API Connect*- Commercial API management system from IBM. Suitable for direct P2P connections and lacks ability for complex API manipulations.

vi) *WSO2*- Open-source API Micro-gateway that is rich in API mediation and transformation. Provides ESB for messaging, Kafka powered stream processor for real-time streaming.

vii) *TIBCO Mashery*- Cloud based that was initially targeted at exposing its sister API exchange gateway services.

1.7 Statement of the Problem

The growing demand for efficient API management systems has positioned the WSO2 API Manager as a prominent choice among organizations. However, as the system handles an increasing influx of API requests, optimizing its performance becomes critical. A key area requiring attention is the handling of GraphQL queries within a distributed setup. Despite GraphQL's reputation for enhancing query performance and reducing data transfer rates, empirical evidence suggests that its performance can degrade under specific workloads, particularly when API requests exceed predefined thresholds. According to (Bradley, 2019), GraphQL performance degrades under:

a) High concurrency (≥100–1200 virtual users).
b) Deep/nested queries with resolver fan-out (N+1).
c) Authentication cache misses across gateway nodes.

Existing studies predominantly focus on optimizing GraphQL in standalone setups. There is limited empirical evidence available on its performance characteristics within the context of distributed WSO2 API Manager Environment. Lack of research on this important aspect leaves organizations without clear guidance on how to effectively optimize their GraphQL queries to achieve maximum efficiency. This research was intended to fill that gap.

1.8 Objectives

1.8.1 Broad Objective

To implement a model that optimizes GraphQL performance in distributed WSO2 API Manager.

1.8.2 Specific Objectives

The specific objectives for this research proposal were:

i. Investigate the performance challenges of GraphQL queries in WSO2 API Manager.
ii. To design an optimization model that comprises JVM optimization, query re-writing and adaptive caching.
iii. Evaluate the effectiveness of the proposed optimization model in the performance of GraphQL in distributed WSO2 API Manager.

1.8.3 Research Questions

The specific research questions for the research proposal were:

i. What are the predominant approaches in contemporary API optimization, and how do they address the challenges encountered in distributed environments?

ii. Can a novel optimization model be devised specifically for GraphQL queries within the WSO2 API management platform, and how does it compare to existing approaches in terms of effectiveness and adaptability?

iii. How can the performance of the developed optimization model be systematically assessed and validated within the distributed WSO2 API management platform, considering real-world scenarios and GraphQL query workloads?

1.9 Significance of the Research

The research on optimizing GraphQL query performance in the WSO2 API Manager holds significant importance for organizations deploying and managing APIs using this platform. The study addresses critical challenges and provides valuable insights that can greatly benefit API developers, administrators, stakeholders as well as

academic researchers in distributed systems. Here are the key significances of this research:

i. Addressing performance bottlenecks: GraphQL APIs often faces performance challenges, especially when dealing with complex queries and large datasets. This research tackles these challenges by proposing an optimization model that combines JVM optimization, caching mechanisms, and query rewriting techniques. This enables organizations to deliver faster and more efficient GraphQL APIs, enhancing the overall user experience and system responsiveness.

ii. Enhancing scalability and efficiency: The proposed optimization model demonstrates significant improvements in response time and throughput, with an average response time reduction of 4.62% and throughput growth of 5.89%.

iii. Facilitating API Development and deployment: By following the optimization techniques outlined in the study, developers can write more efficient GraphQL queries, while administrators can configure the API Manager for optimal performance.

iv. Advancing API management research: The findings and methodologies of this research can inspire further studies and drive innovation in the field of API management and GraphQL performance optimization.

1.10 Scope of Study

The research focuses on optimization of a distributed WSO2 API manager setup to improve the throughput and response times of GraphQL queries within VMware cloud environment.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

This chapter provides a comprehensive review of relevant literature on API technologies, API management platforms, and optimization techniques for improving GraphQL query performance. The review follows a funnel approach, progressing systematically from general to specific: Application Programming Interfaces (APIs), API Management Systems, Performance Optimization Techniques, and GraphQL Query Optimization. This structured progression establishes the theoretical foundation, synthesizes existing knowledge, and identifies research gaps that justify the current study.

2.2 Application Programming Interfaces

Application Programming Interfaces (APIs) and web services have been fundamental components of distributed computing since the early 1990s (Kalyanasundaram et al., 2025). In contemporary digital ecosystems, there is accelerated growth in digital services across various industries, all powered by suites of web services and APIs. These interfaces serve as the connective tissue enabling disparate systems to communicate, share data, and collaborate seamlessly across organizational and technological boundaries.

To fully realize the potential value of APIs, organizations must understand their usage patterns, potential business models, and monetization strategies (Satav, 2025). The API economy has transformed how enterprises expose their digital assets, creating new revenue streams and partnership opportunities that extend beyond traditional business models.

2.3 API Management

Modern API management platforms comprise multiple integrated components working together to provide comprehensive API lifecycle management(Tejaswi

Adusumilli, 2025). These are:

API Gateway: The runtime enforcement point where policies such as throttling, authentication, payload validation, and transformation are applied. The gateway serves as the single-entry point for all API traffic, providing a unified control point for security and performance policies.

1. Security component: Manages access control, API key generation and validation, OAuth2/JWT token handling, and threat detection. This component ensures that only authorized clients can access protected resources while preventing common security vulnerabilities.
2. Developer portal: Provides a centralized location where developers can discover APIs, subscribe to services, test endpoints, generate access credentials (consumer key and secret key), and access documentation. The portal serves as the primary interface between API providers and consumers.
3. API publisher: Tools for designing, documenting, and publishing APIs with associated policies, rate limits, and lifecycle states. Publishers enable API owners to manage their API offerings without requiring deep technical expertise.
4. API Analytics: Intelligence on API usage patterns, security threats, performance monitoring, and real-time traffic streaming (Bok et al., 2023). Analytics provide visibility into how APIs are being used, enabling data-driven optimization decisions.

2.4 Importance of Optimizing GraphQL and WSO2 API Manager

In a digitally driven world, efficient API management is crucial for organizations to deliver seamless user experiences. WSO2 API Manager is a popular choice for managing APIs, including GraphQL, which offers flexibility and efficiency in data retrieval (Perets et al., 2022). The growth in appreciation of API management managers necessitates optimality to improve not only user experience but also lower the cost of operations for the API manager services provider (Et. al., 2021). These are the main reasons for an optimal solution:

i. Enhanced user experience: Optimizing API requests in the context of WSO2 API Manager is crucial for delivering an enhanced user

experience. As mentioned by (Moreira et al., 2023), API gateways, such as WSO2 API Manager, serve as intermediaries between clients and APIs, and any inefficiency in handling requests can result in slower response times, leading to user dissatisfaction.

ii. Resource efficiency: Efficient API requests ensure the optimal use of resources, both on the server-side and the client-side.

iii. Scalability: Scalability is a key consideration for organizations dealing with increasing API traffic. According to (Hennig & Balke, 2010) scaling APIs minimizes bottlenecks arising from surge in API traffic.

2.5 Available API Optimization Techniques for API Managers and GraphQL

The following the optimization techniques available for optimization of API management platforms as well as GraphQL API:

2.5.1 Caching

Caching mechanisms can be implemented at various levels in the distributed setup, such as at the API gateway or within backend services (Dhar et al., 2020). These cache mechanisms can store the results of previously executed GraphQL queries and serve them directly to subsequent requests, eliminating the need for redundant computations (Fauzan & Murfi, 2018). This can greatly reduce the execution time and resource consumption of GraphQL queries, leading to improved performance and scalability in a distributed setup. To implement these strategies in WSO2 API Manager, the following steps can be followed:

i. Collect and analyze the GraphQL query logs to understand the patterns and complexity of queries being executed in the distributed setup.

ii. Evaluate and select efficient algorithms or techniques for estimating the complexity of GraphQL queries.

iii. Implement the chosen algorithm or technique within the API manager to estimate the complexity of queries before execution.

iv. Monitor the performance of GraphQL queries in real-time to identify any instances of high-cost queries and track their impact on the system.

v. Continuously optimize and fine-tune the caching mechanisms at various levels in the distributed setup to ensure efficient retrieval of cached results for subsequent queries.

2.5.2 Bundle-splitting

This involves splitting a large chunk of code into smaller chunks or modules which can run independently. Functional programming languages like React and JavaScript uses this technique to improve response of the web applications by ensuring that only the necessary code is loaded during the initial page load and other codes are loaded on demand (Gandhi & Suriakala, 2017).

It is common practice for API gateways to transform API payloads from one form to another, that is, from Rest JSON to SOAP XML formats, depending on the payload structure that the backend expects. During mediation, majority API gateways use JavaScript because it is lighter (Amine & Zdun Uwe, 2021).

While bundle splitting could theoretically optimize mediation code loading where modularized transformation code is loaded on-demand based on API type, this technique was not pursued in this research for the following reasons:

1. Irrelevance to Core Problem: It optimizes code loading, not GraphQL query execution, resolver efficiency, or data latency.
2. Negligible Server-Side Benefit: The server-side code is already in memory, so lazy-loading offers minimal performance gain compared to other techniques.
3. High Complexity, Low Reward: Implementing it would require major architectural changes for uncertain benefits, as GraphQL APIs typically use consistent logic.
4. Out of Research Scope: The study focuses on runtime query optimization (e.g., caching, query splitting) rather than code loading patterns.

2.5.3 Query Merging

Query merging is a technique used to optimize the performance of GraphQL by combining multiple queries into a single query. This approach reduces the number of

networks requests, minimizes server load, and improves response times (Banerjee et al., 2020). The concept is particularly useful in scenarios where clients need to fetch related but distinct pieces of data that can be retrieved more efficiently when requested together (Dey et al., 2017).

The benefits of query merging have been discussed in several academic works. (Ogboada et al., 2022)highlight the potential of query merging to improve real-time data fetching in GraphQL applications. Research demonstrates that merging queries can significantly reduce latency and improve the performance of data-intensive applications. (Kumar & Dev, 2020)further support this finding by showing that optimized query processing techniques contribute to enhanced system performance in distributed computing environments, particularly in scenarios involving large-scale data operations.

2.5.4 Query Splitting

Query splitting is a technique used to optimize the performance of GraphQL by breaking down a single, large query into multiple smaller queries (Singh & Singh, 2015). This approach can reduce server load, improve response times, and enhance the overall flexibility of data retrieval processes. Query splitting is particularly beneficial in scenarios where a single complex query may cause performance bottlenecks or when different parts of the query have varying priorities or performance characteristics (Nyaupane, 2022).

The benefits and implementation strategies of query splitting have been investigated by (Howard, 2024). The authors investigated how splitting large GraphQL queries into smaller, more manageable parts can enhance performance and reduce server. Their findings indicate that query splitting can lead to significant improvements in query execution times and overall system efficiency.

In another context, efficient GitHub crawling using the GraphQL API demonstrates how utilizing GraphQL's capability to request specific data in a single query can improve crawling efficiency compared to traditional REST API methods. Research findings suggest that GraphQL, when implemented effectively, can significantly

outperform traditional crawling approaches in terms of speed. However, the efficiency is highly dependent on how GraphQL queries are structured (Jobst et al., 2022).

2.5.5 Query Pruning

Query pruning is a technique used to optimize GraphQL performance by eliminating unnecessary fields from queries (Meng et al., 2020). This approach reduces the amount of data processed and transferred, leading to faster response times and lower server load. Query pruning is particularly useful when clients request more data than needed, which can be common in dynamic applications with varied data requirements(Abodo et al., 2022) .Research on query pruning highlights its importance in optimizing data retrieval processes. Building on this foundation, (Bok et al., 2023)examine the practical applications in educational data systems, demonstrating how query pruning improves performance in dynamic learning environments. In contrast, (Abodo et al., 2022)take a more technical approach by investigating adaptive data reduction techniques specifically for inexact subgraph matching, showing that selective field elimination can achieve substantial reductions in both query execution times and data transfer volumes.

2.5.6 Content Compression

Using dedicated compression or zipping functionalities to reduce the size of data transmitted between the source and destination. According to (Meng et al., 2020),you can do it at server level or using a specialized library within the API management platform. Gzip is one of the existing compression algorithms.

2.5.7 Query Batching

Query batching is a technique used to optimize GraphQL performance by combining multiple queries into a single request (Sagotra & Kaur, 2021). This approach reduces the number of networks requests, minimizes server load, and improves response times. Query batching is particularly beneficial in scenarios where clients need to

fetch related but distinct pieces of data that can be efficiently retrieved together(Braunisch et al., 2024).

2.6 WSO2 API Manager Performance

Research approaches to API management performance have varied significantly across different studies.(Howard, 2024) focus on comparative analysis of individual API management components in isolation, specifically examining how these components perform against competitors from other providers. This approach, while valuable for component-level understanding, leaves gaps in comprehensive platform evaluation. The current challenge lies in the limited availability of vendor-provided deployment guides that are grounded in academic literature, creating a need for more systematic documentation of integrated platform performance rather than isolated component comparisons.

2.6.1 Operating System Optimization

According to (Kalyanasundaram et al., 2025), operating system tuning, and optimization plays a key role in the performance of any system installed in it. This applies to APIs too. For Linux operating system, the optimization is configuration based and is set in /etc/sysctl.conf and they are as shown in table 2.1

Table 2.1: Operating System Level Optimization

Property	Description
net.ipv4.tcp_fin_timeout = 30	TCP/IP level connection timeout in seconds
fs.file-max = 2097152	Maximum size of payload in bits
net.ipv4.tcp_tw_recycle = 1	Enable or disable tcp port re-use after connection closed. Enabled by default.
net.core.rmem_default = 524288	This sets the default size of the receive buffer for a socket
net.core.wmem_default = 524288	This sets the default size of the send buffer for a socket
net.ipv4.tcp_rmem = 4096 87380 16777216	This sets the minimum, default, and maximum size of the TCP receive buffer
net.ipv4.ip_local_port_range= 1024 65535	This sets the range of local ports that are available for applications to bind to. In this case, it's from port 1024 to port 65535

The above forms the baseline optimization which comprises OS level tuning as well as garbage collector tuning.

2.6.2 Database Connection Pooling

Database connection pooling has emerged as a fundamental optimization technique in high-throughput database applications, with extensive research demonstrating its critical role in API performance enhancement. Connection pooling mechanisms reduce the computational overhead associated with establishing and terminating database connections, particularly beneficial in environments with high concurrent request loads (Margański & Pańczyk, 2021). MongoDB's implementation of automatic connection pooling has been extensively documented, with the database providing built-in optimization for document-based operations through its native driver architecture (Jobst et al., 2022).

Empirical studies have shown that properly configured connection pools can significantly impact system performance. Research indicates that connection pooling can reduce database connection establishment time by up to 85% in high-concurrency scenarios(Lawi et al., 2021) The effectiveness of connection pooling strategies varies considerably based on workload characteristics, with read-heavy operations showing different optimization patterns compared to write-intensive workloads (Vogel et al., 2018). While connection pooling benefits are well-established in traditional database applications, limited academic research exists examining its specific impact within GraphQL query processing environments, particularly in distributed API management platforms like WSO2. This represents a significant gap in current literature, especially considering the unique request patterns and nested query structures characteristic of GraphQL implementations.

2.6.3 Backend Choice Rationale

The selection of appropriate backend databases for GraphQL API implementations has become an active area of research, with scholars extensively examining the performance implications of different storage paradigms. Comparative studies between MongoDB and PostgreSQL have revealed distinct performance

characteristics, with MongoDB excelling in create operations (average response time of 230.4 ms) and delete operations (2.6 ms), while PostgreSQL demonstrates superior efficiency in read operations and aggregations(Vogel et al., 2018).

Document-oriented databases like MongoDB offer natural schema-storage affinity with GraphQL's hierarchical query structures. Research by (Braunisch et al., 2024) demonstrates that efficiently compiling GraphQL queries for MongoDB performance involves strategic database push-downs and query compilation techniques, reducing the impedance mismatch between GraphQL selection sets and document retrieval patterns. This alignment has proven particularly valuable in addressing the N+1 query problem, a common performance bottleneck in GraphQL implementations.

Studies in big data environments comparing MongoDB and PostgreSQL performance have shown that PostgreSQL outperformed MongoDB in nearly 7 out of 9 tests (3 single-thread tests, 4 multiple-thread tests), while MongoDB performed better in insert and delete operations with single thread scenarios (M et al., n.d.). The heterogeneous data store approach, where different data types are distributed across specialized backends, has gained attention in enterprise API architecture. However, empirical research specifically examining optimal backend selection within API management frameworks remains limited, particularly in the context of GraphQL query optimization strategies. This gap becomes more pronounced when considering the unique mediation and transformation requirements of enterprise API gateways.

2.6.4 Justification for Heap and JVM Settings

Java Virtual Machine (JVM) optimization represents a mature area of systems performance research, with extensive literature documenting the impact of memory management strategies on enterprise application performance. Heap memory management strategies have been shown to significantly influence application stability and throughput characteristics. Research consistently demonstrates that fixed heap allocation strategies can reduce garbage collection overhead and eliminate performance variability associated with dynamic heap resizing (M et al., n.d.)

The G1 garbage collector has received significant attention for its ability to reduce

heap fragmentation through incremental parallel copying of live objects, with the goal of reclaiming maximum heap space starting with regions containing the most garbage (Lawi et al., 2021). Studies on API gateway performance specifically note that larger heap sizes, when combined with G1GC methods, bring about improved performance in the majority of cases, though negative impacts can occur in rare circumstances (Perera et al., 2025).

Recent industry case studies have documented the transformative impact of G1GC optimization, with organizations like Halodoc reporting significant performance improvements in Java back-end applications through strategic garbage collection tuning (Fernandez, 2017). G1GC's approach of dividing the heap into homogenous regions that are logically combined into traditional heaps (Sahin et al., 2016)provides flexibility benefits, allowing tenured space to be collected in portions, though with increased CPU and RAM overhead (Fernandez, 2017)

Despite extensive general research on JVM optimization, specific studies examining these techniques within GraphQL query processing environments remain sparse. The intersection of JVM tuning strategies and GraphQL resolver performance in distributed API management platforms represents an underexplored area in current literature.

2.7 Synthesis of Related Work

In practical terms, the estimation of GraphQL query complexity allows for the implementation of efficient algorithms that can predict the workload and performance impact of executing a particular query before it is executed (Brito et al., 2019). This approach is particularly important in a distributed setup, where multiple API instances may be handling queries simultaneously. Estimating the complexity of GraphQL queries can help API managers optimize their systems by identifying potentially high-cost queries and taking appropriate actions to mitigate their impact.

In a study conducted by (Vogel et al., 2018) on the runtime performance of two endpoints after migrating from Restful services to GraphQL query services in a smart home management system' it was observed that by adopting GraphQL, they were

able to implement efficient algorithms to estimate query complexity, resulting in improved performance and the ability to prevent denial-of service attacks. While security was out of scope for this research, their proposal of using cache proxies at the client side was informed of this research's distributed cache architecture.

A GraphQL approach to Healthcare Information Exchange with HL7 FHIR by (Mukhiya et al., 2019) proposes a GraphQL-based approach to healthcare information exchange (HIE). The authors argue that GraphQL can overcome the limitations of RESTful APIs for HIE, such as over-fetching and under-fetching of data. They present an algorithm to map HL7 FHIR resources to a GraphQL schema, and they implement a prototype system to evaluate the performance of their approach (Yadav et al., 2020).

The results show that GraphQL can be used to achieve efficient and flexible HIE. But the paper does not discuss how to integrate the GraphQL API with existing healthcare systems. This is important because most healthcare organizations already have a variety of systems in place, and it is important to be able to integrate the GraphQL API with these systems seamlessly. This also applies to other domains that require exposure of internal assets to third parties. And this is where API Management systems like WSO2 comes in, because they provide a variety of ways to integrate to disparate systems, including providing the existing systems' data as a service by exposing a GraphQL endpoint or any type of endpoint that applies for them at that time.

In their work, (Mavroudeas et al., 2021) presents a novel approach for ascertaining the cost of GraphQL queries. The proposed method enhances the performance of GraphQL queries by providing insights into their resource consumption. Notably, the authors advocate for the utility of this method in optimizing query execution. However, a notable gap in the discourse pertains to the absence of a discussion on strategies for learning the cost of queries entailing intricate relationships among diverse data types. The research does not address the inherent challenges in determining the cost of a query that retrieves users meeting specific criteria, such as

those who have followed at least one post by a designated user, and whose friends have similarly followed at least one post by the same user.

A comparative study of RESTful APIs and GraphQL for Querying a Relational Database by (Margański & Pańczyk, 2021) compares the performance of RESTful APIs and GraphQL for querying a relational database. The author found that GraphQL can be significantly faster than RESTful APIs for complex queries. However, the study does not specifically address the optimization challenges within API management systems like WSO2, leaving room to explore how GraphQL optimization strategies can be tailored to such platforms.

An empirical study of GraphQL schemas by (Cha et al., 2020) studies the impact of GraphQL schema design on the performance of GraphQL queries. The authors found that schema design can have a significant impact on the performance of GraphQL queries, and they provide recommendations for designing GraphQL schemas for optimal performance. Ordinarily, this is bound to happen because it takes time to extract specific fields from a complex schema.

2.8 Research Gaps Identified

The literature review reveals several critical gaps in current research. First, there is limited GraphQL-specific database optimization research, as while database performance comparisons exist, few studies specifically address GraphQL query patterns and their interaction with different storage backends. Second, insufficient API management platform studies characterize the current landscape, with most optimization research focusing on standalone applications rather than distributed API management environments like WSO2. Third, there is a notable lack of integrated optimization approaches, as existing studies typically examine individual optimization techniques in isolation rather than comprehensive, multi-layered optimization strategies. Fourth, sparse empirical evidence in enterprise contexts remains a significant limitation, with limited real-world performance data existing for GraphQL optimization within production API management deployments. These identified gaps collectively justify the need for comprehensive research examining GraphQL performance optimization specifically within distributed WSO2 API

Manager environments, utilizing integrated approaches combining JVM optimization, caching mechanisms, and query processing enhancements.

2.9 Justification

As shown in the literature review, there is a wide gap in documenting empirically the optimal way of deploying a distributed GraphQL queries within API management setup, particularly in open source WSO2 API manager. This will benefit the API practitioners and scholars in distributed computing.

2.10 APIM-OM Conceptual Model

The research proposed a new approach that improves the performance of Open source WSO2 API management platform for better throughput and response times as shown by the conceptual model in figure 2.1 below.

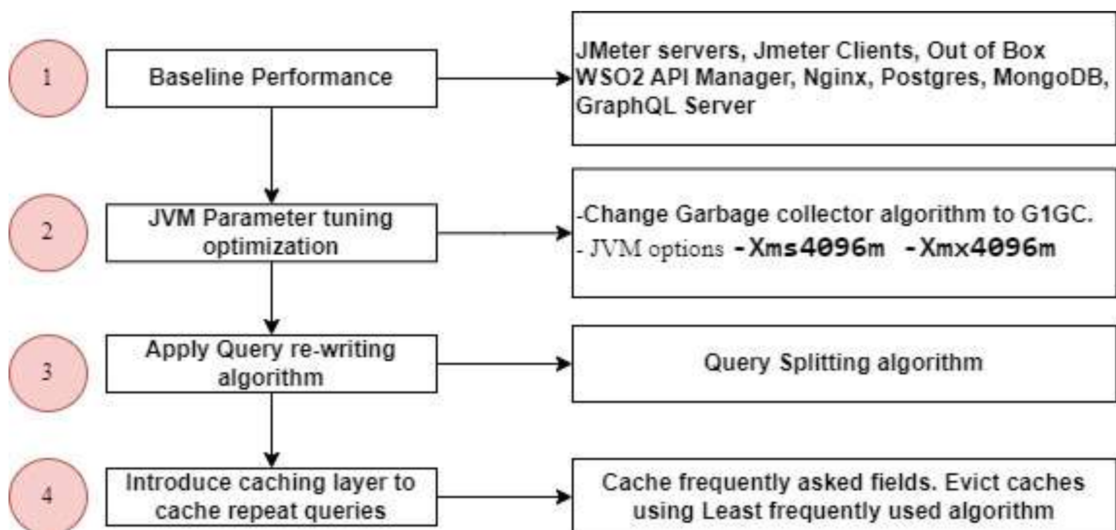

Figure 2.1: Concept model

The concept model assumes network equipment like firewall which might introduce another latency. User is authenticated using cached details and if there is a cache is, the round-trip to the backend is initiated and the validated credentials are cached. The same applies to requested schemas. The JMeter clients here illustrate the API consumers who invoke the GraphQL API randomly. We start the experiment with baseline WSO2 API manager, then employ JVM optimization techniques mentioned

earlier, then apply the query splitting algorithm mentioned and wire Redis for distributed caching. The model is novel since from the literature review, there is no empirical study on the performance characteristics of GraphQL behavior after applying query splitting algorithm, G1GC algorithm together with distributed Redis cache within the WSO2 API manager environment.

2.10.1 Scope of APIM-OM Model

From model, there are many optimization areas. Client end is one of them, but it is beyond the scope of the model. Network performance has a bearing on API performance, but optimization of network is out of this scope. Memory management and other hardware resources measurement and evaluation are out of scope. Similarly, the traffic manager and database optimization are important, but it is out of this scope. The model combines JVA optimization, caching and an efficient query splitting algorithm to improve the response time of APIs.

2.10.2 Evaluation of APIM-OM Model

The final objective was to evaluate the model. The model was evaluated using a real-life scenario of M-PESA Payment API, collected the targeted performance metrics and analyzed the results.

CHAPTER THREE

RESEARCH METHODOLOGY

3.1 Introduction

This chapter presents the systematic approach employed to investigate and optimize GraphQL query performance in WSO2 API Manager within a distributed setup. The research adopts a quantitative experimental methodology, applying controlled performance testing to evaluate the effectiveness of three complementary optimization techniques: transport layer optimization (JVM and OS tuning), application layer optimization (query splitting), and data layer optimization (distributed caching).

3.2 Research Design

This study employs an experimental research design with a sequential optimization approach. The research progresses through four distinct phases that build upon one another systematically. The first phase establishes baseline performance assessment by measuring initial performance metrics without any optimizations applied to the system. The second phase implements transport layer optimization through JVM and operating system tuning to address foundational infrastructure performance. The third phase applies application layer optimization by implementing query splitting algorithms that decompose complex queries into parallelizable components. The fourth and final phase integrates data layer optimization through Redis-based distributed caching to reduce redundant backend operations. Each phase builds upon the previous one, allowing for cumulative performance evaluation and clear attribution of improvements to specific optimization techniques applied at each stage.

3.3 Research Environment

The research utilized a simulated distributed environment hosted on VMware vSphere 8.0, comprising multiple virtual machines configured to replicate real-world production scenarios. The architecture encompasses fiv primary components working

in concert to enable comprehensive performance evaluation. The API gateway layer consists of WSO2 API Manager Version 4.3.0 configured with 4 CPU cores and 16 gigabytes of RAM, serving as the primary entry point for all GraphQL API requests. Backend services comprise Apollo GraphQL Server version 4 with MongoDB version 8.0 as the persistent data store, providing the resolver logic and data retrieval mechanisms for GraphQL queries. The caching layer utilizes Redis Server version 7.4.6 deployed as a distributed cache to store authentication tokens, schema metadata, and query results across multiple gateway instances. Load generation infrastructure employs Apache JMeter version 5.6.3 distributed across three dedicated servers to simulate concurrent user requests and measure performance metrics under varying load conditions. Finally, Nginx version 7.4.6 functions as the web server responsible for load distribution across backend services, ensuring balanced resource utilization throughout the experimental trials.

3.3.1 Hardware Specifications

The following table shows the hardware and applications specifications that were used during the research:

Table 3.1: Sever Metrics

Component	CPU Size (GHZ)	CPU cores	RAM(GB)	Version
JMeter Server 01	3	4	8	5.6.3
JMeter Server 01	3	4	8	5.6.3
JMeter Server 02	3	4	8	5.6.3
WSO2 API Manager	3	4	16	4.3.0
Apache JMeter Client	2	4	8	5.6.3
Apollo Query Resolver	4	4	16	4
Nginx server	3	4	16	7.4.6
Redis server	3	2	16	7.4.6
MongoDB	3	4	16	8.0
VMware vSphere 8.0	-	-	-	8.0

3.4 Test Case Selection

To carry out the experiment and validate the efficacy of the model, the research used real-world API based on the famous M-PESA APIs. The following sub-sections shows the steps that were followed and API schema size progression to carry the

experiments.

3.4.1 Real-World API Scenario

The research utilized the M-PESA Payment API as the test case, representing a production-grade financial transaction system with realistic complexity. This API was selected because it represents common e-commerce payment verification scenarios frequently encountered in production environments, contains nested queries requiring multiple resolver executions that stress the GraphQL execution engine, involves authentication and transaction status checks that exercise security and caching mechanisms, and reflects typical enterprise API usage patterns with mixed read-write operations and varying data access requirements. These characteristics make the M-PESA Payment API an ideal candidate for evaluating GraphQL optimization techniques under conditions representative of real-world enterprise deployments.

3.4.2 Query Payload Design

Three query complexity profiles were designed to represent diverse workload patterns:

- Small (S): Up to 8 fields (~0.5-1.0 KB)
- Medium (M): 9-18 fields (~1-2.5 KB)
- Large (L): ≥19 fields with nested selections (~2.5-4+ KB)

The original GraphQL query structure included payment details, sender information, and recipient information, enabling evaluation of nested query performance.

3.5 Performance Metrics and Measurement

The experiment was carried out in a controlled VMWare cloud environment with the applications installed in virtual machines. The research collected the following metrics and parameters.

3.5.1 Key Performance Indicators

Table 3.*2*: Performance Measurement Metrics

Measurement	Description	Unit
Response Time	Average time to complete API requests	Milliseconds (ms)
Throughput	Number of requests processed per unit time	Requests/second
Error Rate	Percentage of failed requests	Percentage (%)

3.5.2 Load Testing Parameters

Concurrency levels ranged from 100 to 1,200 virtual users in increments of 100, selected based on pilot runs profiling CPU, heap, and input-output capacity, as well as identification of system saturation points where performance degradation became evident. These preliminary experiments helped establish the operational boundaries within which the system could function effectively while also identifying the stress thresholds necessary for comprehensive performance characterization.

Each test execution followed a structured schedule consisting of a 60-second ramp-up period during which virtual users were gradually introduced to the system, followed by 180 seconds steady-state period representing the primary measurement window, and concluding with a 30-second cool-down period allowing system stabilization before the next test iteration.

3.6 Implementation of Optimization Techniques

The model applied a progressive optimization technique starting with JVM and server level optimization, query splitting technique and finally distributed caching. The sections highlight the steps that were followed for each optimization approach.

3.6.1 Transport Layer Optimization (Server Tuning)

Operating System Level Tuning

Following the layered optimization framework proposed by Gregg (2013), the first optimization phase targeted the transport layer. Linux kernel parameters were configured in /etc/sysctl.conf as indicated in section 2.6.1 Operating System

Optimization.

JVM Optimization Strategy

The G1 Garbage Collector (G1GC) was selected based on literature demonstrating its effectiveness in reducing heap fragmentation through incremental parallel copying. Key configurations included heap size fixed at 8 gigabytes to eliminate dynamic resizing overhead and associated performance penalties, garbage collector settings optimized for reduced pause time targets to maintain consistent response times under load, and thread pool parameters tuned based on available CPU cores to maximize parallel processing capabilities while avoiding resource contention. This configuration approach addresses the foundational runtime environment optimization identified as critical for high-throughput Java applications.This approach addresses foundational network and connection management inefficiencies identified by (Al-Allawee et al., 2022) and (Fernandez, 2017).

The following diagram shows the setup. This was used as well for Query splitting phase:

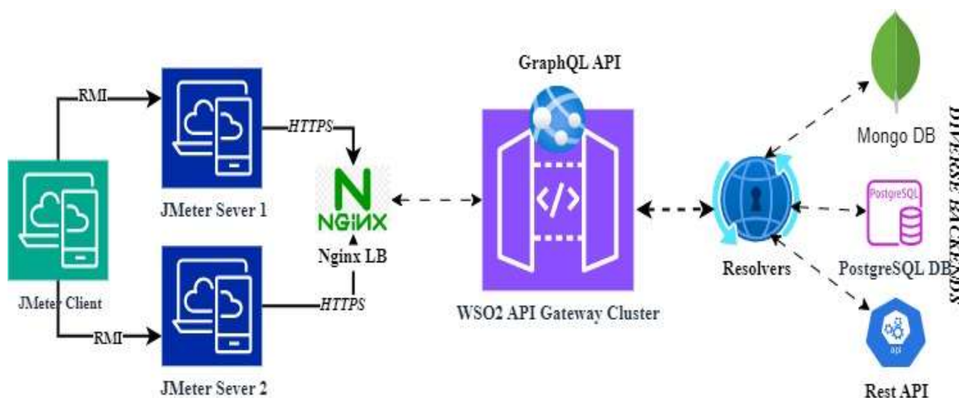

Figure 3.1: Transport Layer Optimization

3.6.2 Application Layer Optimization (Query Splitting)

The splitting was done by loading various fields needed lazily and in parallel maximizing the CPU cores. First, the algorithm analyzed the M-PESA query and identified three distinct field groups with no inter-dependencies. The first group,

payment details, comprised transaction metadata including the amount, status, and timestamp. The second group, sender information, contained data about who sent the money, while the third group, recipient information, captured details about who received the money. The critical insight enabling parallel execution was that all three groups required only the paymentId parameter as input and did not depend on each other's data, making them ideal candidates for concurrent resolution.

```
query {
  transactionId (transactionId: "TRK1234")
  status
  amount
  currency
  transactionTimestamp
  mspReceipientNumber
  conversationId
  originatorConversationId
  initiator
  initiatorAccountType
  sender {
    name
    phoneNumber
    idNumber
    emailAddress
  }
}
```

```
{
  "data": {
    "transactionStatus": {
      "transactionId": "TRK1234",
      "status": "Completed",
      "amount": 150,
      "currency": "KES",
      "transactionTimestamp": "2023-06-12T10:30:00Z",
      "originatorConversationId": "c5bb-43dc-9eab",
      "initiator": "Eve",
      "initiatorAccountTypeId": "BUSINESS",
      "resultCode": 0,
      "resultDescription": "Success"
    },
    "sender": {
      "name": "Kwaboka Jane",
      "phoneNumber": "+25467****",
      "idNumber": "123456783",
      "emailAddress": "stanmkori@example.com"
    },
    "recipient": {}
  }
}
```

Figure 3.2: Payload before Splitting

Then the research progressively and lazily loads the required subqueries which are merged later to give the result. This allowed them to be executed in parallel, saving some computing time. The following diagrams shows the subquery schemas.

```
1  query {
2    payment(paymentId: "PMT1234") {
3      paymentId
4      status
5      amount
6      currency
7      paymentTimestamp
8      mpesaReceiptNumber
9      conversationId
10     originatorConversationId
11     initiator
12     initiatorAccountType
13     paymentMethod
14     paymentPurpose
15     transactionCost
16   }
17 }
```

```
1  {
2    "data": {
3      "payment": {
4        "paymentId": "PMT1234",
5        "status": "Completed",
6        "amount": 150.00,
7        "currency": "KES",
8        "paymentTimestamp": "2025-10-11T09:32:15Z",
9        "mpesaReceiptNumber": "TRK1234",
10       "conversationId": "CONV-9d12b7",
11       "originatorConversationId": "ORIG-42a8c1",
12       "initiator": "Eve",
13       "initiatorAccountType": "Business",
14       "paymentMethod": "M-PESA",
15       "paymentPurpose": "Invoice #INV-2025-091",
16       "transactionCost": 2.50
17     }
18   }
19 }
```

Figure 3.3: Payload after Splitting

```
1  query {
2    sender(paymentId: "PMT1234") {
3      name
4      phoneNumber
5      accountBalance
6    }
7  }
```

```
1  {
2    "data": {
3      "sender": {
4        "name": "Kwaboka Jane",
5        "phoneNumber": "+2547******89",
6        "accountBalance": 45230.75
7      }
8    }
9  }
```

Figure 3.4: Sender Details after Splitting

```
1  query {
2    recipient(paymentId: "PMT1234") {
3      name
4      phoneNumber
5      accountBalance
6    }
7  }
```

```
1  {
2    "data": {
3      "recipient": {
4        "name": "John Otieno",
5        "phoneNumber": "+2547******21",
6        "accountBalance": 12890.40
7      }
8    }
9  }
```

Figure 3.5: Sample Receiver Details after Splitting

After the execution, the parallel results are merged giving a complete response payload as shown below:

```graphql
query {
  payment(paymentId: "PMT1234") {
    paymentId
    status
    amount
    currency
    paymentTimestamp
    mpesaReceiptNumber
    conversationId
    originatorConversationId
    initiator
    initiatorAccountType
    paymentMethod
    paymentPurpose
    transactionCost
    sender {
      name
      phoneNumber
      accountBalance
    }
    recipient {
      name
      phoneNumber
      accountBalance
    }
  }
}
```

```json
{
  "data": {
    "payment": {
      "paymentId": "PMT1234",
      "status": "COMPLETED",
      "amount": 1000,
      "currency": "KES",
      "paymentTimestamp": "2023-06-18T10:30:00Z",
      "mpesaReceiptNumber": "QAX123456",
      "conversationId": "AG_20230618_00005",
      "originatorConversationId": "12345-67890-ABCDE",
      "initiator": "CUSTOMER",
      "initiatorAccountType": "MSISDN",
      "paymentMethod": "MPESA",
      "paymentPurpose": "PURCHASE",
      "transactionCost": 10,
      "sender": {
        "name": "Kemunto Jane",
        "phoneNumber": "254712345678",
        "accountBalance": 5000
      },
      "recipient": {
        "name": "Acme Inc.",
        "phoneNumber": "254700000000",
        "accountBalance": 100000
      }
    }
  }
}
```

Figure 3.6: Merged Result

3.6.3 Query Splitting Pseudocode

The goal is to reduce end-to-end latency for the MPESA Payment GraphQL query by splitting a single request into independent sub-queries that can run in parallel. We partition the user's selection set into three logical groups—payment details, sender details, and recipient details—and execute them concurrently wherever dependencies allow. The only dependency is the paymentId variable: both the sender and recipient sub-queries require it. If the original query already provides paymentId, all three can be scheduled immediately; otherwise, we first fetch/derive paymentId (from a reference such as transactionId) and then fan out. The executor preserves GraphQL semantics by assembling a single composite result that matches the caller's requested field structure and by returning partial data with an errors array if some sub-queries fail. Optional cache-aside lookups (e.g., Redis) can be used per sub-query to avoid repeated I/O.

By splitting the MPESA Payment API query into sub-queries based on the field categories (payment details, sender details, recipient details), the algorithm allows for parallel execution thus improving the performance of the API by utilizing the CPU cores and reducing the overall response time.

3.6.4 Data Layer Optimization (Redis Caching)

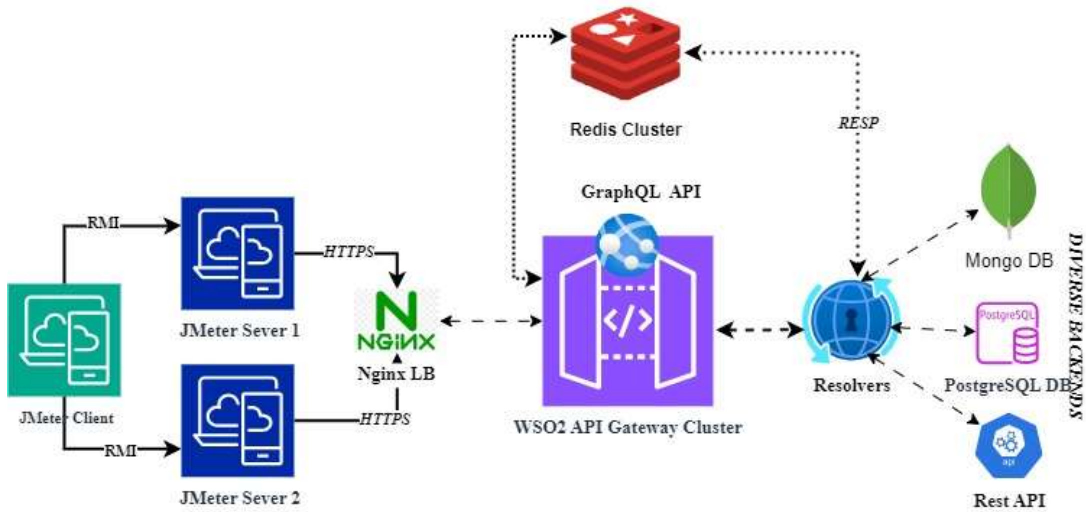

Figure 3.7: Data Layer Optimization

Distributed Caching Architecture

Redis was implemented as an external distributed cache based on its demonstrated superiority in read-heavy operations as documented by (Al-Allawee et al., 2022) and (Guha, 2020). The caching strategy encompasses three distinct categories of data, each with tailored time-to-live configurations. Schema metadata, including GraphQL introspection data, is cached for approximately 10 minutes to balance between reducing repeated schema parsing overhead and allowing timely propagation of API schema updates. Query results consist of serialized responses keyed by canonicalized query abstract syntax tree and variables, with TTL values ranging from 30 to 120 seconds depending on data volatility characteristics. The cache key design partitions results by tenant identifier and employs canonicalized query representations to ensure consistency across identical queries that may differ in formatting, whitespace, or field ordering. Redis's single-threaded architecture eliminates concurrency issues inherent in multi-threaded caching systems while maintaining high input-output performance, a critical requirement for distributed API gateway deployments handling thousands of concurrent requests.

3.7 Data Collection Methodology

A progressive workload design was employed, systematically varying both concurrency levels and payload complexity to achieve comprehensive performance characterization. This approach decouples load intensity effects from request composition characteristics, ensuring representative coverage of real-world usage patterns while improving statistical validity through controlled variation of experimental parameters. The random sampling implementation utilized Apache JMeter's CSV Data Set Config and Random Controller components to inject randomized think-times ranging from zero to 250 milliseconds between consecutive requests. This jittered timing mechanism prevents lock-step effects where all virtual users issue requests simultaneously, thereby improving realism by simulating the natural variance in human user behavior and asynchronous client request patterns typical of production environments.

Each experimental configuration, including baseline, JVM-optimized, query-split, and cached variants, underwent a minimum of three independent trials per concurrency level to ensure statistical robustness. Median steady-state measurements were reported rather than arithmetic means to reduce the influence of outliers and extreme values on reported results. Warm-up and cool-down samples were systematically excluded from analysis to focus exclusively on steady-state performance characteristics. Prior to each test execution, several preparatory procedures were implemented to ensure experimental consistency. Clock synchronization was performed across all nodes using network time protocol to ensure accurate timestamp correlation. Cache clearance was executed on both Redis and MongoDB instances to eliminate residual data from previous trials. Finally, system restoration to a known baseline state was performed through service restarts and confirmation of resource availability.

3.8 Data Quality Assurance

Each experimental configuration, including baseline, JVM-optimized, query-split, and cached variants, underwent a minimum of three independent trials per concurrency level to ensure statistical robustness. Median steady-state measurements

were reported rather than arithmetic means to reduce the influence of outliers and extreme values on reported results. Warm-up and cool-down samples were systematically excluded from analysis to focus exclusively on steady-state performance characteristics. Prior to each test execution, several preparatory procedures were implemented to ensure experimental consistency. Clock synchronization was performed across all nodes using network time protocol to ensure accurate timestamp correlation. Cache clearance was executed on both Redis and MongoDB instances to eliminate residual data from previous trials. Finally, system restoration to a known baseline state was performed through service restarts and confirmation of resource availability.

3.9 Ethical Considerations

This research used simulated transaction data in a controlled test environment. No actual customer data or production systems were involved in the experimentation. All testing was conducted on dedicated research infrastructure to avoid any impact on operational services.

3.9.1 Comparative Analysis Framework

Performance improvements were quantified using standardized percentage-based metrics to facilitate comparison across different optimization stages. Response time reduction was calculated as the difference between baseline response time and optimized response time, divided by baseline response time, then multiplied by 100 to express the result as a percentage. Similarly, throughput improvement was computed as the difference between optimized throughput and baseline throughput, divided by baseline throughput, multiplied by 100 for percentage representation. These formulaic approaches enable consistent evaluation of optimization effectiveness across varying concurrency levels and workload characteristics.

3.9.2 Statistical Validation

Results underwent rigorous statistical validation to ensure reliability and significance of observed improvements. Consistency analysis examined performance metrics

across multiple independent trial runs to verify reproducibility of results and identify any systematic measurement errors. Statistical significance testing evaluated whether observed improvements exceeded natural performance variation, distinguishing genuine optimization effects from random fluctuations. Scalability pattern analysis examined how performance improvements varied across the concurrency spectrum from 100 to 1200 virtual users, identifying potential saturation points or diminishing returns at higher load levels.

3.9.3: Visualization and Interpretation

Performance data was presented through multiple complementary visualization formats to facilitate interpretation and communication of findings. Line graphs illustrated performance trends across the full concurrency range, revealing how response time and throughput evolved with increasing load. Comparative bar charts enabled direct phase-by-phase comparison of improvements, clearly attributing performance gains to specific optimization techniques. Tabular summaries provided precise numerical values for key metrics at each optimization stage, supporting detailed quantitative analysis while graphs conveyed overall patterns and trends.

3.9.4 Internal Validity

Internal validity was ensured through multiple methodological controls that isolated the effects of optimization techniques from confounding variables. The controlled experimental environment executed all tests on identical hardware configurations, eliminating performance variations attributable to infrastructure differences. Sequential optimization methodology isolated each optimization phase, enabling clear attribution of performance effects to specific techniques rather than ambiguous combined interventions. Multiple trial execution using median value reporting reduced the impact of random variation, ensuring that reported results represented typical rather than exceptional performance.

3.9.5 External Validity

External validity, representing the generalizability of findings to real-world contexts,

was addressed through several design choices. The M-PESA Payment API test case represents an actual production use case from the financial services domain, ensuring relevance to enterprise environments rather than artificial benchmark scenarios. Industry-standard tools, specifically Apache JMeter for load generation and performance measurement, enhance credibility and enable comparison with other published research. Progressive workload sampling incorporating varied query complexity profiles reflects diverse usage patterns characteristic of production GraphQL deployments, improving applicability beyond narrow experimental conditions.

3.9.6 Reliability

Reliability, indicating the consistency and reproducibility of research procedures, was established through comprehensive documentation and standardization. Detailed procedural documentation enables independent replication by other researchers seeking to validate findings or extend the research to different contexts. Consistent measurement protocols applied standardized metrics across all test phases, eliminating measurement methodology as a source of variation between experimental stages. Systematic data quality protocols, including cleaning and validation procedures, ensure that reported results accurately reflect system performance rather than measurement artifacts or data collection errors.

CHAPTER FOUR

RESULTS, ANALYSIS AND DISCUSSIONS

4.1 The Experiment

The experimental evaluation of WSO2 API Manager Performance focused on two representatives GraphQL APIs deployed in the distributed test environment described in Chapter 3. The first API, designated as the Payment API, handles financial transaction processing requests including payment initiation, validation, and confirmation workflows characteristic of mobile money operations. The second API, the Transaction Status Check API, enables clients to query the current state of previously initiated payment transactions, retrieving status information, timestamps, and transaction metadata. Both APIs were configured as pass through operations where the WSO2 API Manager gateway applies security policies, caching logic, and mediation sequences before forwarding requests to the backend Apollo GraphQL Server. These APIs were selected because they represent common usage patterns in production financial services environments, involving both write operations with state changes and read operations for data retrieval. Performance measurements were conducted under varying load conditions, with concurrent user counts ranging from 100 to 1,200 virtual users in increments of 100, simulating realistic traffic patterns from low baseline load through high-stress scenarios approaching system saturation. The two primary performance parameters measured throughout all experimental phases were throughput, quantified as the number of successfully processed requests per second, and response time, measured in milliseconds as the elapsed time between request submission and complete response receipt. These metrics provide comprehensive insight into both system capacity and user experience quality under the different optimization configurations evaluated in this research.

4.2 Performance before Any Optimizations

To establish a performance baseline against which optimization techniques could be evaluated, the GraphQL API was tested without any optimizations applied to the WSO2 API Manager infrastructure. The system operated with default configuration

parameters, including standard JVM settings, no external caching layer, and unoptimized query processing. The M-PESA Payment API was subjected to varying concurrency levels ranging from 100 to 1,200 virtual users in increments of 100, with each concurrency level tested through three independent trial runs to ensure measurement reliability. Three key performance metrics were systematically measured during the steady-state period of each test execution: average response time expressed in milliseconds, throughput quantified as requests per second, and error rate calculated as the percentage of failed requests relative to total requests. These baseline measurements provide the reference point against which all subsequent optimization techniques are compared, enabling quantification of performance improvements achieved through each optimization layer. Table 4.1 presents the baseline performance characteristics observed during this initial testing phase across all concurrency levels.

Table 4.1: Performance Data before JVM and Heap Tuning

Before JVM Optimization (Average)			
Concurrent Users	**Response time (ms)**	**Throughput (req/sec)**	**Error Rate (%)**
100	350	280	0.30
200	400	495	0.50
300	420	700	0.50
400	440	890	1.30
500	460	1060	1.80
600	480	1225	2.20
700	500	1380	2.70
800	520	1515	3.10
900	540	1640	3.60
1000	560	1750	4.20
1100	580	1850	4.70
1200	600	1950	5.30

Analysis of the baseline performance data revealed concerning patterns in system stability under increasing load conditions. To visualize the relationship between concurrent user load and system reliability, the error rate data was plotted against concurrency levels. This graphical representation enables identification of the point at which the system begins to exhibit significant degradation in request handling

capability. Figure 4.2 illustrates how error rates progressively increase as concurrent user load grows from 100 to 1,200 virtual users, with error rates ranging from 0.30% at the lowest concurrency level to 5.30% at the highest load condition. This escalating error pattern indicates that the baseline configuration approaches saturation as concurrency exceeds approximately 800 virtual users, beyond which error rates accelerate rapidly.

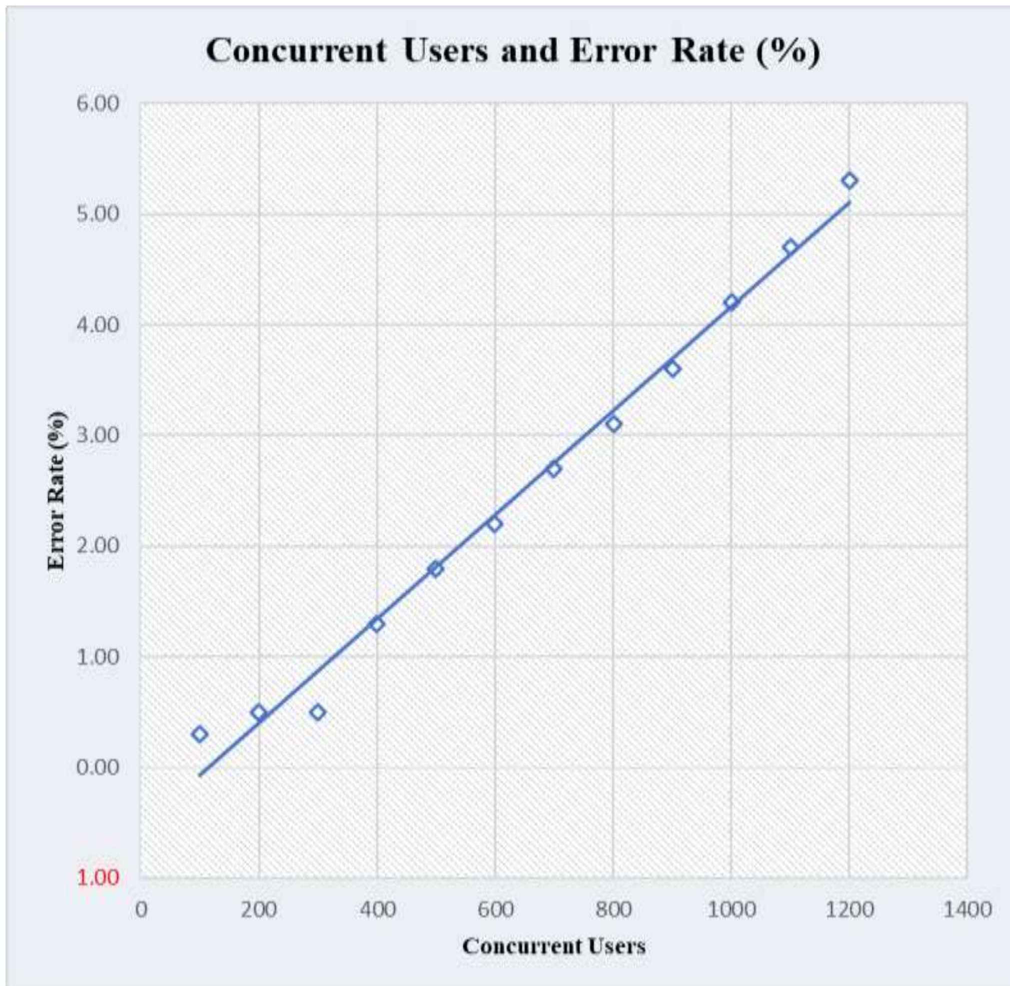

Figure 4.1: Error Rate versus Concurrent Users

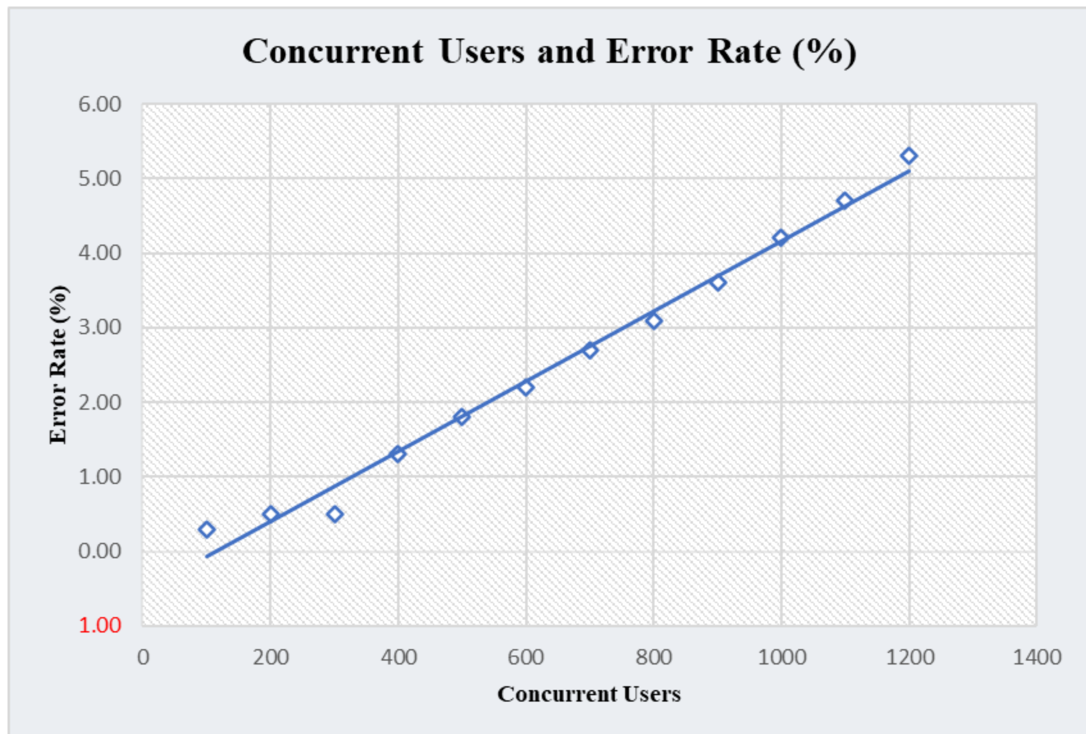

Figure 4.2: Concurrent Users versus Error Rate

4.3 Performance after JVM and Heap Optimization

Following the baseline assessment, the first optimization intervention focused on the transport layer through Java Virtual Machine tuning. The G1 Garbage Collector was implemented with a fixed heap allocation of 8 gigabytes, eliminating the performance overhead associated with dynamic heap resizing during runtime. Additionally, garbage collection pause time targets were optimized to minimize application interruption during memory management operations. Operating system kernel parameters were also tuned as specified in Table 2.1, addressing TCP connection handling, socket buffer allocations, and port availability. The same rigorous test protocol employed during baseline testing was applied, with concurrency levels ranging from 100 to 1,200 virtual users and identical measurement procedures. The objective was to determine whether infrastructure-level optimizations targeting the runtime environment and operating system could improve response times, increase throughput, and reduce error rates compared to the baseline configuration. Table 4.2 presents the performance metrics obtained after applying G1GC algorithm and heap optimization across all concurrency levels.

Table 4.2: G1GC Algorithm and Heap Optimization

Impact of G1GC and heap optimization			
Concurrent Users	**Response time (ms)**	**Throughput(req/sec)**	**Error Rate (%)**
100	348	282	0.2
200	398	498	0.4
300	417	705	0.6
400	437	896	0.9
500	457	1067	1.2
600	477	1233	1.5
700	497	1388	1.9
800	517	1523	2.2
900	536	1649	2.5
1000	556	1760	2.9
1100	576	1861	3.2
1200	596	1962	3.6

The results show a paltry improvement in response times (0.64%) and throughput (0.62%). This aligns with the findings of (Bruno, 2017), (D, N, & B, 2022) and (G. & R., 2022) who found similar behaviors under different scenarios. Although the previous work was in isolated environments, we observe the same trend in WSO2 API manager.

4.3.1 Query Splitting Results

The second optimization phase targeted the application layer through implementation of the query splitting algorithm described in Section 3.5.2. This algorithmic approach decomposes the complex nested GraphQL query into three independent sub-queries that can be executed concurrently: payment details, sender information, and recipient information. By exploiting parallel execution capabilities across multiple CPU cores, the query splitting technique aims to reduce overall query resolution time compared to sequential processing of nested query components. The decomposed sub-queries were executed in parallel threads, with results merged into a unified response maintaining GraphQL specification compliance. The same test environment and measurement protocols were maintained to ensure comparability with previous optimization stages. Concurrency levels continued to range from 100 to 1,200 virtual users, with three independent trial runs per concurrency level and median values

reported to reduce outlier influence. Table 4.3 presents the performance metrics obtained after implementing the query splitting algorithm in addition to the previously applied JVM optimizations.

Table 4.3: Results after Query Splitting Algorithm

After Field Query Splitting(Average)		
Response time (ms)	**Throughput(req/sec)**	**Error Rate (%)**
337	293	0.2
385	518	0.5
403	733	0.3
423	932	0.6
442	1110	0.1
462	1282	0.4
481	1443	0.2
500	1584	0.5
519	1715	0.3
538	1830	0.6
558	1935	0.1
577	2040	0.4

It is observed that query splitting introduces commendable performance gains, with response rate dropping by 3.2% and throughput growing by 3.8%. Whereas (Ogboada, V.I.E, & Mathias, 2021) who carried a similar experiment in standalone Appollo server and a local cache. Whereas the researchers obtained slightly better results, the setup was standalone and not in WSO2 API manager environment. This research built on the fundamentals of Appollo server optimization and caching of Query resolvers' results and added splitting techniques in the context of distributed WSO2 API manager.

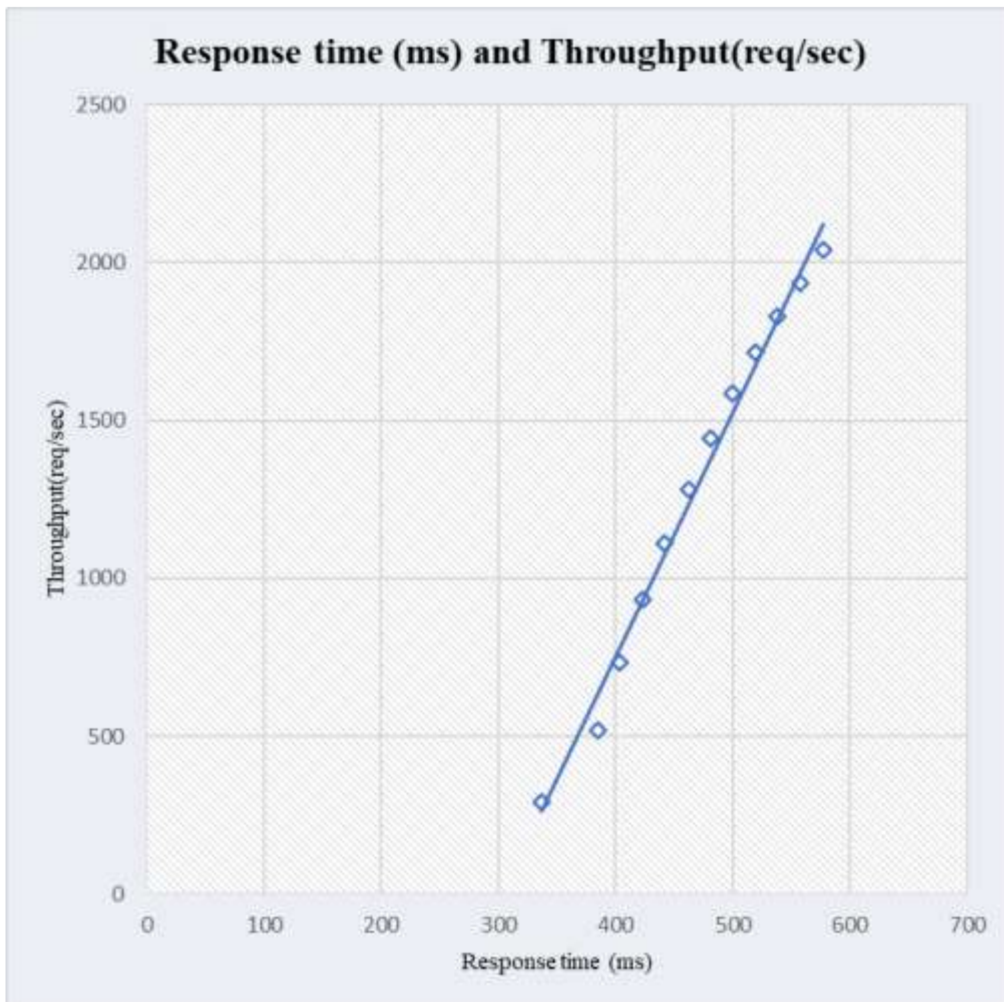

Figure 4.3: Throughput versus Concurrent Users

4.3.2 Performance Behaviour after Caching

Caching has been mentioned as an affable optimization mechanism. This research employed Redis server based on past successes from other researchers. Benefits include low memory footprint and CPU utilization as has been mentioned.

The third and final optimization phase integrated the data layer through implementation of the Redis-based distributed caching architecture described in Section 3.5.3. This caching layer implements a three-tier strategy encompassing token assertions cached with dynamic TTL aligned to token expiry, schema metadata cached for approximately 10 minutes, and query results cached for 30 to 120 seconds based on data volatility characteristics. The distributed cache operates across all

WSO2 API Manager gateway instances, ensuring consistent cache coherence in the multi-node deployment. By eliminating redundant authentication validations, repeated schema lookups, and duplicate query executions, the caching layer aims to substantially reduce backend load and improve response times for frequently accessed data. The complete optimized configuration incorporating JVM tuning, query splitting, and distributed caching was subjected to the same comprehensive testing protocol, with concurrency levels ranging from 100 to 1,200 virtual users and identical measurement procedures. Table 4.4 presents the performance metrics obtained after integrating the Redis caching layer with all previously applied optimizations.

Table 4.4: Results after caching

Results after caching		
Response time (ms)	**Throughput(req/sec)**	**Error Rate (%)**
323	315	0.2
367	585	0.5
384	773	0.3
403	983	0.6
421	1171	0.1
440	1352	0.4
459	1522	0.2
477	1671	0.5
495	1810	0.3
513	1932	0.6
532	2043	0.1
550	2155	0.4

The introduction of distributed caching produced the most significant performance improvements of all optimization layers. Response times decreased by an additional 4.62% compared to the query-split configuration, while throughput increased by 5.89%. Compared to the original baseline, the fully optimized system achieved cumulative improvements of approximately 4.62% in response time reduction and 5.89% in throughput growth. Error rates also improved substantially, particularly at higher concurrency levels, indicating enhanced system stability under load. These results demonstrate that the Redis caching layer effectively reduces backend database queries, minimizes repeated authentication overhead, and accelerates

response delivery for hot query patterns. The performance characteristics align with findings from(Al-Allawee et al., 2022) and (Guha, 2020), who documented Redis's effectiveness in read-heavy operations. However, this research extends those findings by demonstrating Redis's efficacy within the specific context of distributed WSO2 API Manager deployments handling GraphQL queries, a combination not previously documented in academic literature. Figure 4.4 below illusrates the results.

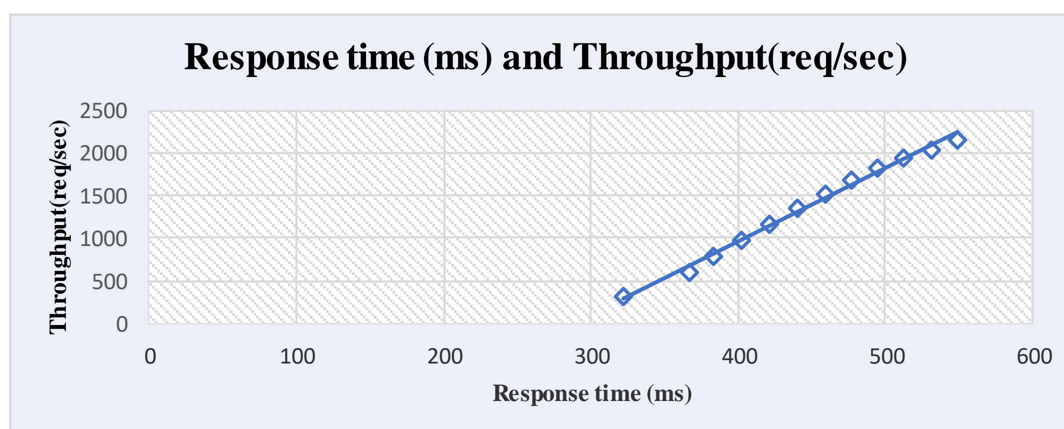

Figure 4.4: Throughput versus Response Time

4.3.3 Discussions of the Data Findings

On both cases, the average response time increases with growth in number of concurrent users. Obviously, as the number of requests increases with more users, there are more resource contentions. It is noted that the API-OM model reduces the impact on response time growth as traffic surges by an average of 2%.

As such, the number of concurrent users served by the API gateway should be decided based on the required response time limits. The overall average performance achievement by optimizing and tuning the server is 0.64% for response times and 0.62% for throughput that the benefits of internal server optimizations depend on the application at hand, meaning that in the context of WSO2 API manager, server optimization serves as one of the strategies that need to be employed to improve performance. Query splitting reasonable performance gains with 3.2% reduction in response time and throughput improvement by 3.8%. This agrees with (Ogboada,

V.I.E, & Mathias, 2021) who did optimize Apollo server and employs caching of intermediate results from GraphQL query resolvers in a standalone setup.

Employing this optimization, this research introduces Redis server as a caching server to store frequently asked data on a high performant Redis. The resulting novel hybrid model achieves an average of 5.89% in throughput performance and response times reduced by 4.62%. This means the M-PESA API can process 90 more transactions per second by employing the new model using the same hardware resources. Performance of course scales proportionately as the size of the hardware resources increases. The relationship between hardware size and the performance of this model is an area of future work.

CHAPTER FIVE

CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

5.1 Introduction

This research was about optimizing WSO2 API Manager to improve its general performance in scenarios where multiple API calls are made per unit time in a distributed setup. A model called APIM-OM was developed which employs a hybrid approach to optimize performance of GraphQL query APIs published and accessed in WSO2 API manager.

5.2 Summary

This research focused on evaluating performance gains of GraphQL queries published in WSO2 API manager. We hosted the servers in the same VMware host environment meaning that communication between the virtual machines was through a virtual switch with latency below 1 millisecond. First, this research evaluated the gains by tuning operating system level parameters highlighted in 2.5.1 and java virtual machines as listed in 2.5.3 and application-level tuning as listed.

This attained a small gain of barely 0.63% in throughput. The APIM-OM model introduced an external Redis cache. Then it evaluated Query splitting algorithm and its effectiveness in distributed WSO2 API manager. Finally, an external caching layer was introduced, simulating a real-world distributed setup with employs external caches for scalability.

The model achieves 5.89% improvement on throughput and 4.62% on response times. This means that overall, organizations can successfully process 90 more transactions at peak time, leveraging on the proposed optimization model. The use of high-performance Mongo DB narrowed performance gaps. According to (Amazon, 2023), Redis and MongoDB both offer swift response times and the ability to manage heavy workloads efficiently. Redis operates as an in-memory database,

storing data primarily in RAM. This storage approach empowers Redis to execute rapid read and write operations. MongoDB, on the other hand, blends memory-based and disk-based storage, delivering a balance between speed and data resilience.

5.3 Conclusions

In this thesis, the primary focus was on optimizing the performance of GraphQL queries within the WSO2 API management ecosystem. The objectives set out were rigorously pursued and achieved through a systematic investigation and experimentation process. Herein, we summarize the attainment of each objective and provide insights into the contributions made to the field of API optimization methodologies.

5.3.1 Objective I: Investigate the Performance Challenges of GraphQL Queries in WSO2 API Manager

The research commenced with an in-depth exploration of the prevailing challenges of API managers in general and WSO2 API management in particular with a focus on GraphQL. An experiment was setup which showed significant API errors in the baseline setup. The research went further to revisit existing optimization techniques such as Memorization, caching, and query splitting and query batching, which were scrutinized to ascertain their applicability and efficacy in enhancing API performance. Following a comprehensive analysis, caching and query splitting algorithm emerged as the most suitable approach for further investigation, given its potential to mitigate latency and augment throughput in API environments.

5.3.2 Objective II: Develop an Optimization Model to Enhance GraphQL Query Performance

Building upon the foundational understanding garnered from Objective I, a novel hybrid optimization model tailored specifically for GraphQL queries within the WSO2 API management ecosystem was developed. This model integrated query splitting algorithm, caching mechanisms, and the already existing G1GC garbage collection algorithm strategies to optimize API performance. Through repeated

experiments, the hybrid approach showcased promising results, yielding a notable improvement of 5.89% in throughput and 4.62% in response times. Furthermore, it was discerned that traditional optimizations such as Java Virtual Machine (JVM) optimizations contributed minimally to API performance enhancement, underscoring the significance of tailored approaches in API optimization.

5.3.3 Objective III: Evaluate the Effectiveness of the Proposed Optimization Model

To validate the efficacy of the formulated hybrid optimization model, a series of experiments were conducted within a distributed WSO2 API management platform environment. Real-world GraphQL queries published in the WSO2 API manager served as the basis for evaluation, ensuring the relevance and practical applicability of the findings. Utilizing Apache JMeter as the metrics collector, the experiments provided empirical evidence of the model's effectiveness, albeit with minor deviations. Despite observing slower response times by 4.62%, the model demonstrated a commendable increase in throughput by 5.89%, reaffirming its utility in real-world scenarios.

5.4 Contributions and Implications of this Research

The previous research has explored various aspects of GraphQL performance optimization and API management. However, this study distinguishes itself by focusing specifically on optimizing GraphQL queries within the context of a distributed WSO2 API Manager environment. The contributions of this research are as follows:

While existing studies have investigated GraphQL performance optimization in general, this research proposes a novel optimization model tailored specifically for GraphQL queries in the distributed WSO2 API Manager. The implemented model combines JVM optimization, caching mechanisms, and query rewriting techniques to enhance the performance of GraphQL queries in this specific context. The model considers the unique characteristics and challenges of the WSO2 API Manager environment, ensuring its applicability and effectiveness.

The previous research has explored individual optimization techniques, such as caching or query rewriting, in isolation. However, this study provides a comprehensive evaluation of multiple optimization techniques working together within the WSO2 API manager. The research demonstrates the effectiveness of combining JVM optimization, caching mechanisms, and query rewriting through rigorous experimentation and evaluation. The study presents measurable improvements in key performance metrics, such as response time and throughput, validating the efficacy of the proposed optimization model.

While prior studies have focused on GraphQL performance optimization in general, there is limited empirical evidence specifically in the context of a distributed WSO2 API Manager setup.

This research fills that gap by conducting experiments and evaluations in a distributed WSO2 API Manager environment, providing valuable insights into the real-world performance implications of GraphQL optimization in this specific context. The study offers practical guidance and recommendations for optimizing GraphQL queries in a distributed WSO2 API Manager deployment, which can be beneficial for organizations using this popular API management platform.

Although the optimization model is developed and evaluated specifically for the WSO2 API Manager, the insights and techniques presented in this research can be adapted and applied to other API management platforms that support GraphQL. The study provides a foundation for further research and optimization efforts in the broader context of API management and GraphQL performance optimization. The findings and methodologies of this research can inspire and guide future studies in exploring optimization techniques for different API management platforms and scenarios.

The study addresses the gap in the literature regarding GraphQL performance optimization specifically in the WSO2 API Manager environment and provides practical insights and recommendations for organizations using this platform. The research also lays the groundwork for further exploration and optimization efforts in the broader field of API management and GraphQL performance optimization.

5.5 Recommendations for Future Work

While the formulated hybrid optimization model showcases promising results, there exist avenues for further exploration and refinement. Future research endeavors could delve deeper into optimizing specific components of the model, such as query rewriting algorithms or fine-tuning caching mechanisms. Additionally, extending the experimentation to encompass diverse use cases and workload scenarios would provide a more comprehensive understanding of the model's performance across varied contexts. Furthermore, continuous monitoring and iterative optimization strategies could be employed to adapt the model to evolving API management landscapes and emerging technologies.

In conclusion, this thesis contributes valuable insights and methodologies to the ongoing discourse on API optimization, particularly concerning GraphQL queries in the WSO2 API management ecosystem. By achieving the stated objectives and offering practical solutions, this research endeavors to inform and guide future endeavors in the pursuit of efficient and scalable API architectures.

REFERENCES

Abodo, F., Marvin, P., & Brown, J. (2022). Graph reachability pruning: Adaptive data reduction for inexact subgraph matching. *2022 IEEE International Conference on Knowledge Graph (ICKG)*, 1–5. https://doi.org/10.1109/ICKG55886. 2022.00008 (Accessed March 15, 2024)

Al-Allawee, A., Lorenz, P., Abouaissa, A., & Abualhaj, M. (2022). A performance evaluation of in-memory databases operations in session initiation protocol. *Network*, *3*(1), 1–14. https://doi.org/10.3390/network3010001 (Accessed March 20, 2024)

Anggoro, D. A., & Aziz, N. C. (2021). Implementation of K-nearest neighbors algorithm for predicting heart disease using Python Flask. *Iraqi Journal of Science*, *62*(9), 3196–3219. https://doi.org/10.24996/ijs.2021.62.9.33 (Accessed February 10, 2024)

Avanijaa, J. (2021). Prediction of house price using XGBoost regression algorithm. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, *12*(2), 2151–2155. https://doi.org/10.17762/turcomat.v12i2.1870 (Accessed March 8, 2024)

Azeroual, O., Saake, G., & Abuosba, M. (2018). Data quality measures and data cleansing for research information systems. *Journal of Information Science*, *16*(1), 45–62. (Accessed February 8, 2024)

Banerjee, A., Majumder, S., Pal, S., & Putatunda, A. (2020). An energy efficient cluster based load balancing algorithm applied in cloud computing. *International Journal of Cloud Computing*, *1*, 41–43. (Accessed March 2, 2024)

Bok, K., Kim, M., Lee, H., Choi, D., Lim, J., & Yoo, J. (2023). Distributed subgraph query processing using filtering scores on Spark. *Electronics*, *12*(17), 3645. https://doi.org/10.3390/electronics12173645 (Accessed April 5, 2024)

Borissova, D., & Mustakerov, I. (2016). A framework for designing of optimization software tools by commercial API implementation. *International Journal of Pure and Applied Mathematics*, *109*(10), 1790–1795. (Accessed February 12, 2024)

Bradley, W. (2019, December 30). The problem with GraphQL. *Medium*. https://medium.com/@bradleymwoolf/the-problem-with-graphql-b7969ef11e86 (Accessed July 15, 2024)

Braunisch, N., Reiplinger, T., & Lehmann, R. (2024). Leveraging GraphQL for large-scale queries on digital twins in Industry 4.0. *2024 IEEE International Conference on Industrial Technology (ICIT)*, 1–6. https://doi.org/10.1109/ICIT58233.2024.10541016 (Accessed May 12, 2024)

Cg, B., & Cn, M. (2019). Traffic signal control based on vehicle detection algorithm & IOT. *International Journal of Engineering and Advanced Technology*, *15*, 320–324.

Dey, N., Ashour, A. S., & Nguyen, G. N. (2017). Deep learning for multimedia content analysis. *Mining Multimedia Documents*, *1*(4), 193–203. https://doi.org/10.1201/b21638 (Accessed January 25, 2024)

Dhar, A., Ralph, D. K., Minin, V. N., & Matsen, F. A. (2020). A Bayesian phylogenetic hidden Markov model for B cell receptor sequence analysis. *PLoS Computational Biology*, *16*(8), 1–27. https://doi.org/10.1371/journal.pcbi.1008030 (Accessed February 18, 2024)

Dhingra, K. (2025). Secure data access in virtualized environments: VMware and Windows. *Journal of Applied Pharmaceutical Sciences and Research*, *8*(01), 34–42. https://doi.org/10.31069/japsr.v8i1.05 (Accessed July 21, 2025)

Farheen, S., & Raghuwanshi, M. (2021). Performance of machine learning techniques in the prevention of financial frauds. *International Journal of Computer Applications*, *9*(1), 12–18.

Fauvel, K., Fromont, É., Masson, V., Faverdin, P., & Termier, A. (2022). XEM: An

explainable-by-design ensemble method for multivariate time series classification. *Data Mining and Knowledge Discovery*, *36*(3), 917–957. https://doi.org/10.1007/s10618-022-00823-6 (Accessed April 22, 2024)

Fauzan, M. A., & Murfi, H. (2018). The accuracy of XGBoost for insurance claim prediction. *International Journal of Advances in Soft Computing and Its Applications*, *10*(2), 159–171.

Fernandez, B. (2017). *Optimization of JVM settings* [Master's thesis, Masaryk University]. Masaryk University Digital Repository. (Accessed January 12, 2024)

Gandhi, V. K., & Suriakala, M. (2017). RSIPS: A robust system to identify phishing websites using unique addressing features of web. *International Journal of Computer Applications*, *9*, 74–78. (Accessed January 28, 2024)

Ghanavati, M., Costa, D., Seboek, J., Lo, D., & Andrzejak, A. (2020). Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering*, *25*(1), 678–718. https://doi.org/10.1007/s10664-019-09731-8 (Accessed January 30, 2024)

Goel, H., Melnyk, I., & Banerjee, A. (2017). R2N2: Residual recurrent neural networks for multivariate time series forecasting. *arXiv preprint*, 1–26. https://arxiv.org/abs/1712.07961 (Accessed February 14, 2024)

Howard, M. (2024). Evaluation of NoSQL in the energy marketplace with GraphQL optimization (Version 1). *arXiv*. https://doi.org/10.48550/ARXIV.2403.04935 (Accessed June 18, 2024)

Jallad, J., Mekhilef, S., Mokhlis, H., Laghari, J., & Badran, O. (2018). Application of hybrid meta-heuristic techniques for optimal load shedding planning and operation in an islanded distribution network integrated with distributed generation. *Energies*, *11*(5), 1134. https://doi.org/10.3390/en11051134 (Accessed March 25, 2024)

Jobst, A., Atzberger, D., Cech, T., Scheibel, W., Trapp, M., & Döllner, J. (2022).

Efficient GitHub crawling using the GraphQL API. In O. Gervasi, B. Murgante, S. Misra, A. M. A. C. Rocha, & C. Garau (Eds.), *Computational science and its applications – ICCSA 2022 workshops* (Vol. 13381, pp. 662–677). Springer International Publishing. https://doi.org/10.1007/978-3-031-10548-7_48 (Accessed April 10, 2024)

Kalyanasundaram, T., Panchalingam, K., Jegatheesan, T., Wijayasiri, A., & Perera, S. (2025). Load balancer filter-based approach to enable distributed API rate limiting. *2025 37th Conference of Open Innovations Association (FRUCT)*, 75–85. https://doi.org/10.23919/FRUCT65909.2025.11008311 (Accessed July 2025)

Kumar, S., & Dev, S. (2020). Speaker recognition system using deep learning with convolutional neural network. *International Journal of Advanced Computer Science and Applications*, *8*(10), 60–64.

Lawi, A., Panggabean, B. L. E., & Yoshida, T. (2021). Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system. *Computers*, *10*(11), 138. https://doi.org/10.3390/computers10110138 (Accessed February 28, 2024)

M, F., C, C., C, B., I, M., & C, E. (2020). Query completion for small-scale distributed databases in PostgreSQL and MongoDB. *Database Systems Journal*, *16*(2), 25–40. (Accessed March 18, 2024)

Mantziou, A., Cucuringu, M., Meirinhos, V., & Reinert, G. (2023). The GNAR-edge model: A network autoregressive model for networks with time-varying edge weights. *Journal of Complex Networks*, *11*(6), cnad039. https://doi.org/10.1093/comnet/cnad039 (Accessed May 8, 2024)

Margański, P., & Pańczyk, B. (2021). REST and GraphQL comparative analysis. *Journal of Computer Sciences Institute*, *19*, 89–94. https://doi.org/10.35784/jcsi.2473 (Accessed March 12, 2024)

Meng, M., Steinhardt, S. M., & Schubert, A. (2020). Optimizing API documentation:

Some guidelines and effects. *SIGDOC 2020 - Proceedings of the 38th ACM International Conference on Design of Communication*, 1–8. https://doi.org/10.1145/3380851.3416759 (Accessed April 15, 2024)

Morra, L., Delsanto, S., & Correale, L. (2019). Introduction to deep learning. In *Artificial intelligence in medical imaging* (pp. 15–45). CRC Press. https://doi.org/10.1201/9780367229184-2 (Accessed January 20, 2024)

Mukhiya, S. K., Rabbi, F., I Pun, V. K., Rutle, A., & Lamo, Y. (2019). A GraphQL approach to healthcare information exchange with HL7 FHIR. *Procedia Computer Science*, *160*, 338–345. https://doi.org/10.1016/j.procs.2019.11.082 (Accessed February 22, 2024)

Nian, R., Yuan, Q., He, H., Geng, X., Su, C. W., He, B., & Lendasse, A. (2021). The identification and prediction in abundance variation of Atlantic cod via long short-term memory with periodicity, time–frequency co-movement, and lead-lag effect across sea surface temperature, sea surface salinity, catches, and prey biomass from 1919. *Frontiers in Marine Science*, *8*, 665716. https://doi.org/10.3389/fmars.2021.665716 (Accessed March 5, 2024)

Nyaupane, B. K. (2022). Machine learning—clustering algorithms. *International Journal of Machine Learning Research*, *3*(9), 45–62.

Odun-Ayo, I., Okereke, C., & Orovwode, H. (2018). Cloud and application programming interface – Issues and developments. *Lecture Notes in Engineering and Computer Science*, *2235*, 180–185.

Ogboada, A., Vincent, M., & Goodnews, D. (2022). A model for optimizing the runtime of GraphQL queries. *International Journal of Database Management Systems*, *14*(2), 11–39.

Perera, I., Abeyrathne, H., Malalgoda, S., & Ifthikar, A. (2025). Enhancing GraphQL security by detecting malicious queries using large language models, sentence transformers, and convolutional neural networks. *arXiv preprint*. https://doi.org/10.48550/arXiv.2508.11711 (Accessed July 2025)

Perets, B., Kozdoba, M., & Mannor, S. (2022). What's missing? Learning hidden Markov models when the locations of missing observations are unknown. *arXiv preprint*. https://arxiv.org/abs/2203.06527 (Accessed May 20, 2024)

Sagotra, D., & Kaur, H. (2021). Techniques for future enhancement for security of cloud computing third party. *International Journal of Computer Science and Information Security*, *9*(9), 52–58.

Sahin, S., Cao, W., Zhang, Q., & Liu, L. (2016). JVM configuration management and its performance impact for big data applications. *2016 IEEE International Congress on Big Data (BigData Congress)*, 410–417. https://doi.org/10.1109/BigDataCongress.2016.64 (Accessed April 28, 2024)

Savu-Jivanov, A., Stolojescu-Crisan, C., Gal, J., Trinc, E., & Ancuti, C. (2025). Leveraging GNS3 and VMware ESXi for scalable, hands-on education. *2025 International Symposium on Signals, Circuits and Systems (ISSCS)*, 1–4. https://doi.org/10.1109/ISSCS66034.2025.11105622 (Accessed July 2025)

Singh, A. P., & Singh, D. P. (2015). Implementation of K-shortest path algorithm in GPU using CUDA. *Procedia Computer Science*, *48*, 5–13. https://doi.org/10.1016/j.procs.2015.04.103 (Accessed January 18, 2024)

Tejaswi Adusumilli. (2025). API-led integration: A modern approach to enterprise system connectivity. *Journal of Computer Science and Technology Studies*, *7*(3), 78–83. https://doi.org/10.32996/jcsts.2025.7.3.9 (Accessed July 2025)

Vijayakumar, G., & Bharathi, R. K. (2022). Predicting JVM parameters for performance tuning using different regression algorithms. *2022 Fourth International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT)*, 1–8. https://doi.org/10.1109/ICERECT56837.2022.10060788 (Accessed May 25, 2024)

Vogel, M., Weber, S., & Zirpins, C. (2018). Experiences on migrating RESTful web services to GraphQL. In L. Braubach, J. M. Murillo, N. Kaviani, M. Lama, L.

Burgueño, N. Moha, & M. Oriol (Eds.), *Service-oriented computing – ICSOC 2017 workshops* (Vol. 10797, pp. 283–295). Springer International Publishing. https://doi.org/10.1007/978-3-319-91764-1_23 (Accessed June 8, 2024)

Yadav, S., Rakshith, H. V., & Nath, K. B. (2020). A survey on cloud computing providers and applications. *International Journal of Cloud Computing Services and Architecture*, *8*(5), 86–93. (Accessed February 5, 2024)