Performance Evaluation of REST and GraphQL API Models in Microservices Software Development Domain

Mohamed S. M. Elghazal[1], Adel Aneiba[2][a] and Essa Q. Shahra[2][b]
[1]*Faculty of Information Technology Department of Computer Networks & Communications Department, University of Benghazi, Benghazi, Libya*
[2]*Faculty of Computing, Engineering and Built Environment, Birmingham City University, Birmingham B4 7RQ, U.K.*

Keywords: REST API, GraphQL AP, Microservice, Software Development.

Abstract: This study presents a comprehensive comparative analysis of REST and GraphQL API models within the context of microservices development, offering empirical insights into the strengths and limitations of each approach. The research explores the effectiveness and efficiency of GraphQL versus REST, focusing on their impact on critical software quality metrics and user experience. Using a controlled experimental setup, the study evaluates key performance indicators, including response time, data transfer efficiency, and error rates. The findings reveal that REST APIs demonstrate superior memory efficiency and faster response times, particularly under high-load conditions, making them a reliable choice for performance-critical microservices. On the other hand, GraphQL excels in offering greater flexibility for data retrieval, but exhibits higher response times and higher error rates when handling complex queries. This research provides a nuanced understanding of the trade-offs between the REST and GraphQL API interaction models, offering actionable guidance to developers and researchers in selecting the optimal API model for microservice-based applications. The insights are particularly valuable for balancing considerations such as performance, flexibility, and reliability in real-world implementations.

1 INTRODUCTION

In distributed systems, an application programming interface (API) allows distributed devices or services to exchange data regardless of operating system type (Efuntade et al., 2023). For example, an API allows any system to exchange the data and functionality of its services with other systems, devices, or even other applications within an organisation. As a result, organisations are increasingly sharing data, services, and sophisticated resources. This communication can take place locally, with external partners, or with the public. APIs are commonly used for the exchange of text, media, and more (Simon, 2024). Massive and complicated software products are groups of loosely linked services in a microservice architecture. Every service can be hosted separately from the others and is focused only on achieving one goal, excelling at that task (Wells, 2024). Figure 1 shows that a microservice architecture is a collection of loosely linked services that can be deployed independently, each focused on achieving specific business goals. The microservice architectural style is defined as a method of designing software as a collection of discrete services, each of which is executed in its own process and communicates through minimal means, most commonly an API for HTTP resources (Söylemez et al., 2024). These services are aligned with business characteristics and can be deployed independently using fully automated release tools. Various approaches have been proposed to increase the efficiency of data processing, some focusing on the forms of requests and responses, while others optimize the number of queries made over the network. A recent development is the use of a declarative data query approach, where client applications specify the data they need, improving communication with the server to retrieve data more efficiently (Acharya, 2024). Service providers and most software engineers have embraced the REST architectural style, largely due to its ease of use compared to previous web performance standards such as SOAP and RPC. Moreover, the success of REST can be attributed to the constraints imposed on how service components are built, potentially benefiting the over-

[a] https://orcid.org/0000-0002-8021-445X
[b] https://orcid.org/0000-0002-3668-6230

Elghazal, M. S. M., Aneiba, A. and Shahra, E. Q.
Performance Evaluation of REST and GraphQL API Models in Microservices Software Development Domain.
DOI: 10.5220/0013729900003985

In *Proceedings of the 21st International Conference on Web Information Systems and Technologies (WEBIST 2025)*, pages 83-91
ISBN: 978-989-758-772-6; ISSN: 2184-3252

all system when such constraints are applied (Singh et al., 2024).

gRPC is a high-performance framework that enables applications built using different technologies to communicate seamlessly. It is based on the concept of defining a service and the methods that can be called remotely, along with their input parameters and output formats. It originated from Google's internal project, called Stubby, which was designed for fast and efficient inter-service communication. gRPC builds on these principles, bringing them to the wider developer community as an open source solution. (Levandoski et al., 2024). In 2012, Facebook recognized the need to rebuild its native mobile apps, which were initially thin wrappers around mobile website views. A REST server existed, but performance was poor, and the apps frequently crashed. Developers sought to transform how data was delivered to client apps. Lee Byron, Nick Schrock, and Dan Schafer led the development of GraphQL, a query language that allows client/server applications to specify the capabilities and requirements of data models (Sheltami et al., 2018). GraphQL aims to improve the way consumers interact with remote systems by addressing several limitations faced by developers working with REST APIs. For instance, GraphQL allows clients to request only the data they need, whereas REST clients often have to make multiple requests to gather the necessary data (Vadlamani et al., 2021).

GraphQL is a distinct and entirely new method for interacting with APIs. It is an API query and information retrieval approach that is open-source. It provides a customizable, query-based language for interacting with APIs, offering flexibility and a single endpoint. Software components using GraphQL communicate with the server only to retrieve the specific data they request, with queries designed to interact with the web for exactly what is needed—nothing more. GraphQL was publicly released in 2015 as an alternative to existing REST services. On November 7, 2018, Facebook handed over the development of GraphQL to the newly established GraphQL Foundation, which is managed by the non-profit Linux Foundation. As seen, the tech sector is continuously evolving, prompting developers to find solutions that balance flexibility and functionality. While REST APIs offer simplicity and familiarity, their rigid structure can become problematic. In REST, the API provider dictates what data is sent, often leading to over-fetching or under-fetching (Leng et al., 2024). This model worked in the past, but the rise of complex, client-driven applications necessitates a different approach.

GraphQL addresses the shortcomings of REST by

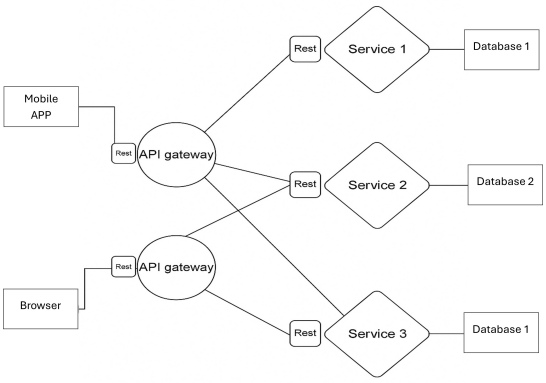

Figure 1: Cloud computing services model.

empowering clients with control over data retrieval. Clients can specify exactly what data they need in a single query, which is crucial in dynamic tech environments. This flexibility can improve application performance, streamline development, and enhance API self-documentation. However, it is important to recognize the added complexity of GraphQL, potential security risks, and the need for custom caching solutions.

The aim of this study is to conduct a comprehensive comparative analysis of REST and GraphQL API models within the context of microservices development, providing empirical insights into the strengths and limitations of each approach.

This paper is organized as follows: Section 2 reviews the related work. Section 3 introduces the proposed system model. Section 4 discuss the implementation and presents the results. Finally, Section 5 concludes the paper.

2 RELATED WORK

GraphQL is a modern class of web-based data access interfaces that offers an alternative to REST-based interfaces. Due to its advantages over REST, GraphQL has gained significant attention and has been adopted by a growing number of users since its official launch (Hartig and Pérez, 2017). Some research studies have evaluated the performance of both REST and GraphQL, but many of these studies rely on simple software development implementations, resulting in either ambiguous outcomes or basic network service functions being tested, such as reading, inserting, modifying, and removing data (Mikuła and Dzieńkowski, 2020). The results demonstrated that both services performed equally well in adding, modifying, and removing data. However, under high load, performance variations were observed for dis-

play functions. While GraphQL may not consistently outperform REST in scenarios involving over-fetching, it demonstrates comparable performance for single requests, positioning it as a viable alternative to REST. Furthermore, in cases of under-fetching, GraphQL proves to be more efficient than REST, as it allows clients to retrieve precisely the data they need in a single query. In environments where under-fetching occurs frequently, GraphQL emerges as a more practical and efficient API architecture compared to REST (Vadlamani et al., 2021).

Authors in (Lawi et al., 2021) found that REST was 51% faster than GraphQL when evaluating the performance of the two systems in a monolithic architectural application . However, this result is specific to monolithic architectures. C. Pautass et al. in (Pautasso et al., 2008) attempted to compare GraphQL and REST but did not consider the microservices domain. When microservices are not fully considered, conclusions about REST can be questionable, as an architecture style is defined by the combination of constraints, not just individual ones. Metrics are implementation-dependent and, thus, not always suitable for judging performance across different domains. In (Brito and Valente, 2020), they involved 22 graduate and undergraduate students, aimed to compare the effort and time required to develop queries for a web service using REST and GraphQL. The experiment took place in a controlled setting, with participants randomly assigned to either the REST or GraphQL group. The results indicated that implementing remote service queries using REST required significantly more effort than with GraphQL, particularly for complex endpoints and multiple inputs, leading to increased development time and effort.

Interestingly, the experiment also showed that even participants with no prior knowledge of GraphQL could easily implement queries, suggesting that GraphQL has a lower learning curve compared to REST, making it more accessible to developers (Amareen et al., 2024). However, one limitation of the experiment was the lack of consideration for microservices as a constraint. Brito argues that this limitation is not significant since most developers work with monolithic architectures. Nevertheless, the results could differ if microservices were taken into account. In (Erlandsson and Remes, 2020) Brito's experiment provides valuable insights into the relative merits of REST and GraphQL, suggesting that GraphQL is a more efficient and easier-to-use option for developing remote service queries. However, it is important to note that these findings may not be generalizable to all development contexts. Despite these limitations, the experiment provides valuable insights into the relative merits of REST and GraphQL. The results suggest that GraphQL is a more efficient and easier-to-use option for developing remote service queries. The asynchronous JavaScript and XML apps designed by Mesbah and van Deursen are also relevant here. Their work connects to Single Page Applications (SPAs) (Mesbah and Van Deursen, 2007), the successors to what they referred to as "AJAX applications," which often utilize GraphQL. They precisely define factors to consider when developing such apps using the REST method, such as delta communication. However, rather than viewing these systems as variations of existing models, they propose a new architectural style that addresses multiple factors previously covered by older models. Systems using SPAs can be seen as composites, for example, employing the Model-View-Controller (MVC) style for the front end and REST for the backend. Shaw et al. (Shaw and Clements, 1997) Presents a classification of architectural styles based on their fundamental elements, including components and connectors, as well as their handling of control mechanisms, data management, and the interplay between control and data aspects. However, this classification is disconnected from the specific requirements of applications, focusing only on technical features like the layout or structure of the final system. This makes it difficult for a designer to choose an architecture, and thus, their classification methodology is not appropriate for comparing GraphQL and REST models, as it does not offer guidance on when to use each for creating an API. Vohra and Manuaba This research aims to fill this gap by conducting a quantitative comparison of REST and GraphQL in a scalable microservices environment. It will explore performance under real-world conditions, including complex queries and varying levels of concurrency, providing developers with the empirical data necessary to make informed decisions about API models in microservices.

Authors in (Vohra and Manuaba, 2022) explored performance comparisons between REST and GraphQL within a microservices architecture framework, using a freelance marketplace as their testbed. They employed Ocelot for REST and HotChocolate for GraphQL as gateways, using "JMeter" to simulate various user scenarios (Niswar et al., 2024). Performance parameters like average response time and throughput were analyzed to highlight differences between REST and GraphQL under diverse scenarios. While they found similar performance for POST and PUT queries, significant discrepancies arose in GET queries that retrieved data from multiple services. Despite the valuable insights from Vohra and Manuaba's study, several limitations must be considered. Firstly,

the study was conducted in a local system rather than a cloud-based environment, which could restrict scalability and real-world applicability. Secondly, the simplified freelance marketplace scenario might not fully capture the complexity of real-world microservices architectures, where interconnected services of varying complexities could affect REST and GraphQL APIs differently. Lastly, the reliance on a limited number of predefined test cases might constrain the comprehensiveness of the findings, potentially missing a full range of scenarios encountered in real-world applications. Despite significant theoretical comparisons between REST and GraphQL, empirical research on their performance within microservices architectures remains limited. Existing studies often focus on monolithic architecture and simulated environments, which do not accurately reflect the complexities of real-world distributed systems.

3 PROPOSED SYSTEM MODEL

The system architecture is built to compare the performance of REST and GraphQL APIs model in a microservices environment. Both the REST and GraphQL services are developed using Go (Golang) and utilize the GORM library to interact with a centralized PostgreSQL database. The architecture consists of three key layers:

1. REST Service Layer: It interacts with the PostgreSQL database using GORM to perform various database operations. The REST API exposes multiple endpoints that allow clients to request resources or perform operations on data.
2. GraphQL Service Layer: Like the REST service, it communicates with the PostgreSQL database using GORM, but provides more flexibility in terms of data fetching, allowing clients to define complex queries and mutations.
3. Data Layer: A PostgreSQL database serves as the shared data layer for both REST and GraphQL services. This setup ensures consistency in data management and allows for a fair comparison of the two services' performance when interacting with the same data source.

The proposed system model is Illustrated in Figure 2, which outlines the flow of communication between different system components. The system is tested using various tools such as K6 for load testing and Postman for manual testing. A Python script is employed for analysis, generating insights into performance metrics like response time, memory consumption, and error rates under different testing scenarios.

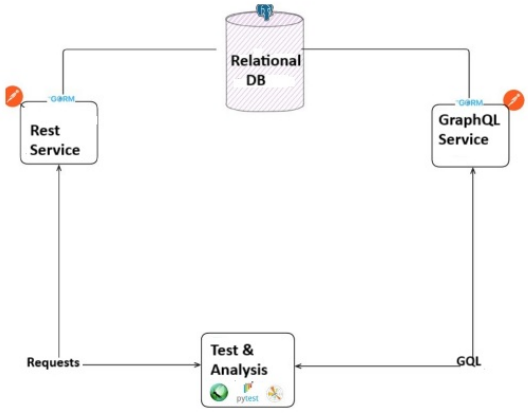

Figure 2: Proposed system model.

The results are analyzed to compare the efficiency and scalability of both API models within the same microservices architecture.

4 EXPERIMENT SETUP AND ANALYSIS PROCEDURES

4.1 Experiment Setup

1. Microservices Development: Two microservices were implemented, each employing a distinct architectural style—one based on REST and the other on GraphQL. Both services were developed in Golang, leveraging its performance and efficiency for backend development, and integrated with PostgreSQL to provide robust and scalable data management across both implementations. The microservices and databases were deployed on separate virtual machines running Ubuntu 22.04 LTS, each provisioned with 4 vCPUs, 8 GB RAM, and 50 GB SSD storage. These machines were connected over a dedicated gigabit Ethernet network to minimize latency and ensure consistent throughput during testing. This deployment approach addressed limitations identified in earlier work that relied solely on local, single-machine tests, allowing for a more realistic evaluation of distributed system performance.
2. Performance Testing Framework: Testing was conducted using the Grafana K6 framework, which enabled the simulation of real-world usage scenarios to evaluate system performance. The tests scaled up to 1,000 Virtual Users (VUs), effectively simulating varying levels of user load to ensure the microservices could handle diverse operational demands.

3. Test Scenarios: The performance evaluation was designed to assess the system under varied operational loads while accounting for differences in data-retrieval behavior between REST and GraphQL. Notably, REST often required multiple requests to retrieve the same data due to underfetching. For example, in the Heavy Consumer Scenario, a REST client typically made two sequential requests of 150bytes each, whereas GraphQL could retrieve the equivalent dataset in a single 300-byte query. In some cases, REST responses also included unused fields (overfetching), leading to slightly larger payload sizes than strictly necessary. These distinctions are important when comparing network utilization, latency, and backend workload.
   - The Light Consumer Scenario focused on read-intensive operations, simulating data fetching with up to 1,000 Virtual Users (VUs) ramping up in one minute, holding steady for 30 seconds, and then gradually ramping down. This scenario aimed to assess API efficiency in managing concurrent read requests.
   - Concentrated on write-intensive operations, starting two minutes into the test and following the same ramping profile as the Light Consumer Scenario. It measured latency and error rates during peak insertion loads. Network payload sizes were comparable between REST and GraphQL for writes, as these operations typically involved a single structured request.
   - Emphasized complex data-retrieval involving nested or relational queries. Beginning at the four-minute mark, this scenario highlighted a key difference: REST often required multiple endpoint calls (e.g., two $\times$ 150-byte requests) to assemble the complete dataset, while GraphQL could perform the same retrieval in a single composite query (300bytes). This allowed direct analysis of query efficiency and backend processing costs.
   - Focused on frequent modifications to existing data, starting six minutes into the test with the same ramping structure. While both REST and GraphQL updates were typically single-request operations, REST payloads tended to contain some redundant data fields due to overfetching in update responses. This scenario evaluated efficiency, consistency, and error management under high update loads.

4.2 Analysis Procedures

1. Key Metrics Evaluated

Table 1: REST API vs. GraphQL API results comparison.

Metric	REST API	GraphQL API
Average Request Duration	247.17 ms	915.14 ms
Error Rate	6.86% 15,462 failed out of 225,262 requests	5.85% 8,613 failed out of 138,492 requests
Requests Completed	225, 262 471 requests /second	147,327 312 request /second
Request Sending Duration	1.23 ms (avg), 24.86 second (max)	477.14 Mic sec(avg), 1.74 seconds (max)
Request Receiving Duration	44.46 ms(avg), 1 minute, 3 seconds	1.64 ms (avg), 3.96 seconds (max)
Total VUs	1,000 max VUs	1,000 max VUs
Test Duraion	8 minutes	8 minutes

   - Response Time: Measures efficiency in delivering prompt results under varying loads.
   - Memory Consumption: Tracks resource usage to evaluate scalability under high-demand conditions.
   - Error Rate: Monitors system reliability, stability, and ability to handle complex queries without failure.

2. Performance Assessment Goals
   - Provide a comprehensive evaluation of the system's using the key metrics evaluation mentioned above. .
   - Identify the suitability of the microservices for high-demand environments.

5 RESULTS

Table 1 provides a side-by-side comparison of the REST and GraphQL APIs outputs aspects:

5.1 Individual Metric

The Average Request Duration is another critical metric that highlights the efficiency of both API models. As shown in Figure 3, the average request duration for GraphQL is significantly higher than that of

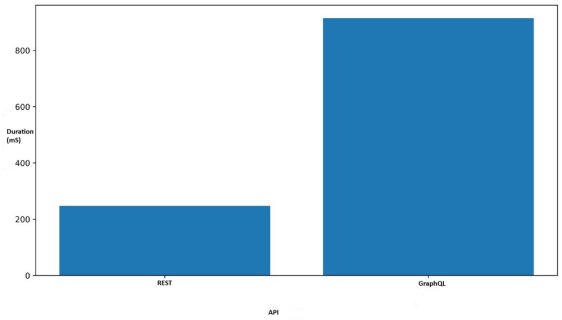

Figure 3: Average request duration by API model.

REST, with GraphQL averaging around 850 ms compared to REST's average of approximately 250 ms. This substantial difference can be attributed to the increased complexity of GraphQL queries, which often involve fetching nested and specific data structures that require more server-side processing time. REST, by contrast, operates with predefined endpoints that are optimized for quicker data retrieval, leading to lower request duration. The longer request duration in GraphQL demonstrates a trade-off between flexibility and performance. While GraphQL offers more detailed and precise data querying capabilities, this comes at the cost of increased server-side processing time, especially when handling complex queries. REST, with its simpler and more straightforward data fetching mechanism, is able to provide faster response times, making it more suitable for applications where low-latency interactions are critical.

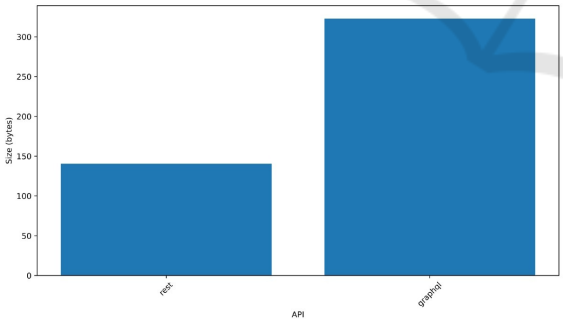

Figure 4: Average Request Size of Rest and GraphQL Models.

Figure 4, shows a significant difference in Average Request Size between REST and GraphQL. REST requests were smaller on average, with sizes typically around 150 bytes. In contrast, GraphQL requests were more than twice as large, averaging over 300 bytes. This increase in size for GraphQL is due to its ability to allow clients to request complex and nested data in a single query. While REST requests are generally simpler and more streamlined due to the fixed structure of endpoints, GraphQL's flexibility comes at the cost of larger payloads. This finding suggests that while GraphQL offers greater query flexibility, it may introduce inefficiencies in terms of network bandwidth when dealing with large or frequent requests.

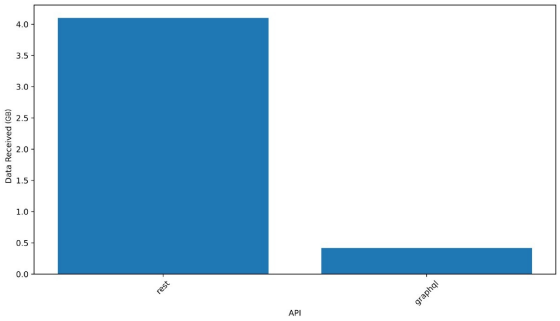

Figure 5: Data received comparison between the two models.

Figure 5, illustrated Total Data Received by the client from the server also varied significantly between REST and GraphQL. REST APIs received far more data, with the total data received nearing 4 GB, while GraphQL received significantly less data—close to a few hundred MB. This disparity is primarily due to the over-fetching inherent in REST, where endpoints return a fixed set of data regardless of how much is actually needed by the client. In contrast, GraphQL's ability to request only the data that is required leads to smaller data transfers, particularly in complex queries where only a subset of fields is needed. This result highlights the efficiency of GraphQL in reducing unnecessary data transmission, making it more suitable in bandwidth-constrained environments.

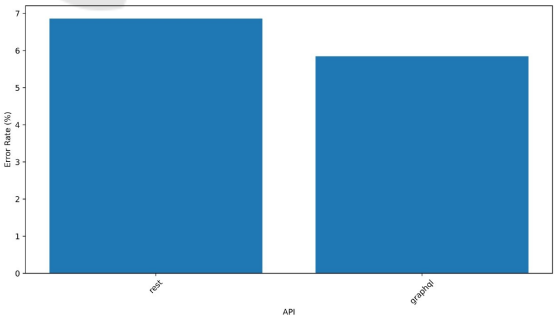

Figure 6: Error rate comparison between the two API models.

Figure 6, presents the Error Rate comparison between REST and GraphQL. REST showed a slightly higher error rate, around 7%, while GraphQL's error rate was slightly lower, approximately 6%. This dif-

ference suggests that REST experiences more challenges when handling high traffic or more complex operations. GraphQL's ability to handle more specific queries might reduce the likelihood of errors, as it allows clients to retrieve exactly what they need, reducing the complexity of server-side operations. However, it is important to note that this difference is relatively small, and both API models performed reliably under typical load conditions.

5.2 Time Series Analysis for REST and GraphQL API Models

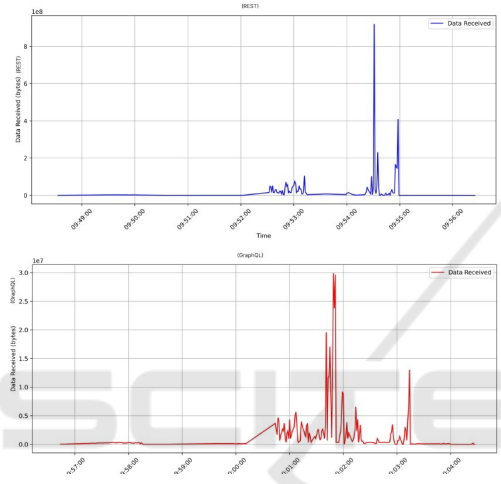

Figure 7: Comparison of Data Received (REST vs. GraphQL).

The data received by the REST and GraphQL APIs during the performance tests demonstrates the different approaches each API takes in handling data retrieval. As shown in Figure 7, the REST API consistently processes larger volumes of data compared to GraphQL. This is a direct result of REST's static nature, where predefined endpoints often return all available data fields, regardless of whether the client requires them. In contrast, GraphQL's flexibility allows clients to specify exactly what data they need, reducing the amount of unnecessary data transferred.

This observation highlights the efficiency of GraphQL in minimising data transfer, which is particularly advantageous in bandwidth-constrained environments or applications where only partial data is frequently needed. However, REST's approach can be beneficial in scenarios where full data responses are consistently required, as it simplifies server-side implementation and reduces client-side logic.The volume of data handled by each API model during the performance tests, highlighting the more efficient data transfer exhibited by GraphQL in complex querying scenarios.

The comparison of data sent between the REST and GraphQL APIs reveals similar trends to the data received metrics. The REST API tends to send more data, as it returns complete responses based on predefined structures. GraphQL, on the other hand, exhibits more efficient data transfer by sending only the data explicitly requested by theclient. This leads to a smaller data footprint in GraphQL, especially in complex scenarios where clients can avoid fetching unnecessary fields.

This characteristic of GraphQL presents a clear advantage in applications where precise data fetching is required, such as mobile apps or environments with limited resources. However, the simplicity of REST's data transmission model makes it more predictable and easier to manage in scenarios where uniform data payloads are necessary. As shown in Figure 8, the amount of data sent through the REST and GraphQL APIs demonstrates a significant contrast, particularly when handling requests involving multiple nested data structures. The Figure emphasizes GraphQL's capability to send precise data based on client requirements, unlike REST, which often results in over-fetchin.

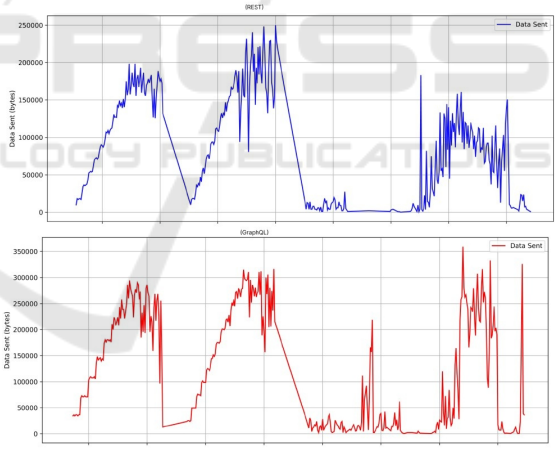

Figure 8: Comparison of Data Sent (REST vs. GraphQL).

The error rates observed during testing indicate that GraphQL encounters more frequent errors per second compared to REST, particularly under heavy load and when resolving complex, nested queries. This is largely due to the additional computational overhead required by GraphQL to dynamically resolve requests and gather data from multiple sources. In contrast, REST's simpler, more linear request-response model results in fewer errors, making it more reliable under high-stress conditions.The frequency of errors encountered per second by each API model is visualized in Figure 9. The results demon-

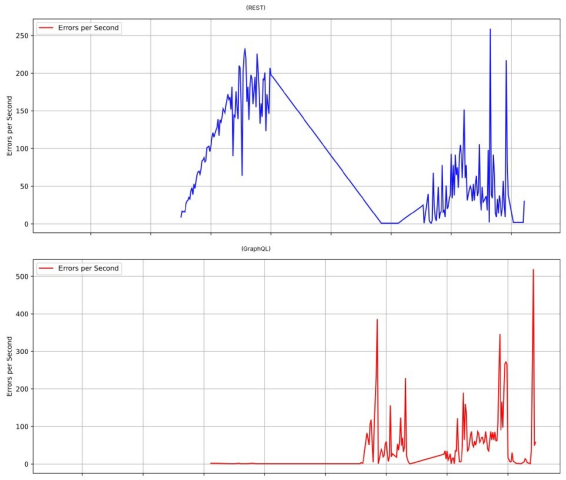

Figure 9: Comparison of errors per second (REST vs. GraphQL)).

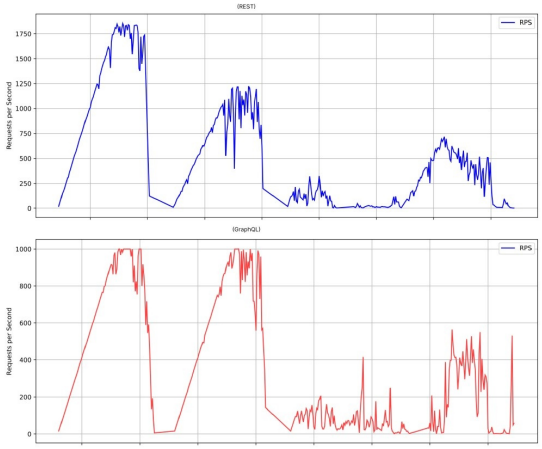

Figure 10: Comparison of requests per second (REST vs. GraphQL).

strates that GraphQL, though more flexible, tends to generate more errors under higher loads, especially when dealing with intricate data queries. This difference underscores the importance of optimizing GraphQL resolvers and schema design to mitigate error rates in production environments. While GraphQL offers flexibility, it requires careful management to avoid performance bottlenecks and error spikes during peak usage. REST, with its more robust error handling under load, is better suited for applications where reliability is prioritized over flexibility. The requests per second (RPS) metric provides insight into the throughput capabilities of each API. The REST API consistently handles a higher number of requests per second compared to GraphQL. This is expected, given REST's simpler request-response model, where requests are statically defined and require less computational overhead to fulfill. In contrast, GraphQL's dynamic nature, where the client defines the shape of the response, introduces additional processing time, reducing its overall memory.

This observation suggests that REST is better suited for high-throughput scenarios where performance and scalability are critical. GraphQL, while capable of handling complex queries, may struggle to maintain high RPS in applications with intense traffic unless optimizations are made to streamline query resolution. For developers, this highlights the trade-off between flexibility and performance in API design, with REST being the preferred option for performance-critical applications.

The comparative analysis shows that REST handles a higher volume of requests per second more efficiently, while GraphQL's performance fluctuates depending on the complexity of the queries being processed, as illustrated in Figure 10.

6 CONCLUSION

This study presented a comprehensive and empirical evaluation of REST and GraphQL API models within the dynamic landscape of microservices development. The primary objective was to provide a detailed comparison of these two paradigms based on critical performance metrics, including response time, memory consumption, and error rates. Through a series of meticulously designed experiments that leveraged real-world scenarios in a social media system, this research effectively highlighted the unique strengths and limitations of each API model in managing complex data interactions. The experimental results revealed that REST consistently demonstrated superior performance in terms of average request duration and error rates, especially under high-load conditions, underscoring its suitability for performance-critical applications. Conversely, GraphQL excelled in offering a significantly higher degree of flexibility in data fetching, allowing for more granular and precise data retrieval. This capability reduces issues such as over-fetching or under-fetching, enhancing the developer experience in applications where precise data control is a priority. The findings of this study offer valuable practical insights for both developers and researchers. By presenting robust performance data and clearly articulating the trade-offs between REST and GraphQL, the research provides actionable guidance on selecting the most appropriate API model for microservices-based systems. This work enriches the growing body of knowledge on API technologies and empowers the development community to make informed decisions tailored to real-world performance demands and specific system requirements.

REFERENCES

Acharya, K. (2024). Chat application through client server management system project. *Authorea. July 29, 2024. DOI: https://doi. org/10.22541/au. 172228527.74316529/v1*.

Amareen, S., Dector, O. S., Dado, A., and Bosu, A. (2024). Graphql adoption and challenges: Community-driven insights from stackoverflow discussions. *arXiv preprint arXiv:2408.08363*.

Brito, G. and Valente, M. T. (2020). Rest vs graphql: A controlled experiment. In *2020 IEEE international conference on software architecture (ICSA)*, pages 81–91. IEEE.

Efuntade, O. O., Efuntade, A. O., and FCIB, F. (2023). Application programming interface (api) and management of web-based accounting information system (ais): Security of transaction processing system, general ledger and financial reporting system. *Journal of Accounting and Financial Management*, 9(6):1–18.

Erlandsson, P. and Remes, J. (2020). Performance comparison: Between graphql, rest & soap.

Hartig, O. and Pérez, J. (2017). An initial analysis of facebook's graphql language.

Lawi, A., Panggabean, B., and Yoshida, T. (2021). Evaluating graphql and rest api services performance in a massive and intensive accessible information system. computers 2021, 10, 138.

Leng, J., Guo, J., Xie, J., Zhou, X., Liu, A., Gu, X., Mourtzis, D., Qi, Q., Liu, Q., Shen, W., et al. (2024). Review of manufacturing system design in the interplay of industry 4.0 and industry 5.0 (part i): Design thinking and modeling methods. *Journal of Manufacturing Systems*, 76:158–187.

Levandoski, J., Casto, G., Deng, M., Desai, R., Edara, P., Hottelier, T., Hormati, A., Johnson, A., Johnson, J., Kurzyniec, D., et al. (2024). Biglake: Bigquery's evolution toward a multi-cloud lakehouse. In *Companion of the 2024 International Conference on Management of Data*, pages 334–346.

Mesbah, A. and Van Deursen, A. (2007). An architectural style for ajax. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 9–9. IEEE.

Mikuła, M. and Dzieńkowski, M. (2020). Comparison of rest and graphql web technology performance. *Journal of Computer Sciences Institute*, 16:309–316.

Niswar, M., Safruddin, R. A., Bustamin, A., and Aswad, I. (2024). Performance evaluation of microservices communication with rest, graphql, and grpc. *International Journal of Electronics and Telecommunication*, 70(2):429–436.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs." big"'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814.

Shaw, M. and Clements, P. (1997). A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 6–13. IEEE.

Sheltami, T. R., Shahra, E. Q., and Shakshuki, E. M. (2018). Fog computing: Data streaming services for mobile end-users. *Procedia computer science*, 134:289–296.

Simon, F. M. (2024). Escape me if you can: How ai reshapes news organisations' dependency on platform companies. *Digital Journalism*, 12(2):149–170.

Singh, S. P., Mehta, A., and Vasudev, H. (2024). Application of sensitivity analysis for multiple attribute decision making in lean production system. *Engineering Management Journal*, pages 1–24.

Söylemez, M., Tekinerdogan, B., and Tarhan, A. K. (2024). Microservice reference architecture design: A multi-case study. *Software: Practice and Experience*, 54(1):58–84.

Vadlamani, S. L., Emdon, B., Arts, J., and Baysal, O. (2021). Can graphql replace rest? a study of their efficiency and viability. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 10–17. IEEE.

Vohra, N. and Manuaba, I. B. K. (2022). Implementation of rest api vs graphql in microservice architecture. In *2022 International Conference on Information Management and Technology (ICIMTech)*, pages 45–50. IEEE.

Wells, S. (2024). *Enabling microservice success: managing technical, organizational, and cultural challenges*. " O'Reilly Media, Inc.".