Performance evaluation of microservices communication with REST, GraphQL, and gRPC

Muhammad Niswar, Reza Arisandy Safruddin, Anugrayani Bustamin, and Iqra Aswad

Abstract—**Microservice architecture has become the design paradigm for creating scalable and maintainable software systems. Selecting the proper communication protocol in microservices is critical to achieving optimal system performance. This study compares the performance of three commonly used API protocols: REST, GraphQL, and gRPC, in microservices architecture. In this study, we established three microservices implemented in three containers and each microservice contained a Redis and MySQL database. We evaluated the performance of these API protocols using two key performance metrics: response time and CPU Utilization. This study performs two distinct data retrieval: fetching flat data and fetching nested data, with a number of requests ranging from 100 to 500 requests. The experimental results indicate that gRPC has a faster response time, followed by REST and GraphQL. Moreover, GraphQL shows higher CPU Utilization compared to gRPC and REST. The experimental results provide insight for developers and architects seeking to optimize their microservices communication protocols for specific use cases and workloads.**

Keywords—**Microservices; API; gRPC; REST; GraphQL**

I. INTRODUCTION

SOFTWARE development using microservices architecture has changed the way we design applications. This architecture advocates breaking down complex applications into smaller, self-contained microservices. Each microservice has specific tasks and that can be managed and changed without affecting other components. It allows development teams to focus on specific aspects of the application, improving scalability, faster changes, and better fault isolation [1].

In microservice communication, two commonly used protocols are Representational State Transfer (REST) and Graph Query Language (GraphQL). REST has been one of the most widely used data exchange methods, which relies on a number of endpoints to access and manipulate data. Although REST remains popular, it comes with certain drawbacks, such as over-fetching or under-fetching data, where the retrieved data may exceed or fall short of actual needs. Addressing these drawbacks, GraphQL emerges as an attractive alternative. GraphQL allows clients to specify the data they needed [2] [3], overcoming REST's inefficiency problem and giving application developers more control.

Muhammad Niswar, Reza Arisandy Safruddin, Anugrayani Bustamin, Iqra Aswad are with Department of Informatics, Faculty of Engineering, Hasanuddin University, Gowa, South Sulawesi, Indonesia (e-mail: niswar@unhas.ac.id, rezaarisandy2525@gmail.com, anugrayani@unhas.ac.id, iqra@unhas.ac.id.)

In addition to REST and GraphQL, another data exchange method gaining attention today is Google Remote Procedure Call (gRPC). gRPC offers an efficient and versatile approach to communication among distributed services. Unlike the REST and GraphQL methods that utilize the HTTP/1 protocol, gRPC employs the HTTP/2 protocol and supports streaming data. gRPC simplifies remote procedure calls across various programming languages, delivering enhanced performance and speed in microservice communication [4].

In this study, we aim to evaluate and compare the performance of REST, gRPC, and GraphQL for data exchange within a microservice system under both fetching flat data and nested data. Our study includes a performance analysis with key performance metrics, including response time and CPU utilization. By evaluating these three communication protocols, we aim to assist developers and organizations in making informed decisions when designing and implementing microservices-based systems.

II. RELATED WORK

There have been many studies that compare the performance of REST and GraphQL. Reference [5] describes the performance of the REST and GraphQL in using the Ocelot and Hot Chocolate Application Programming Interface (API) gateways in the case of writing data and getting data. Reference [6] discusses the advantages and disadvantages of the REST and GraphQL. When dealing with data that undergoes frequent changes and needs to be handled efficiently with resource optimization in mind, GraphQL is the preferred choice. Reference [7] describes the REST as the appropriate selection for the data exchange method in situations where data is consistently accessed. Reference [8] [9] focuses on implementing GraphQL in a web application, which shifts from REST to GraphQL. Reference [10] [11] compares REST and GraphQL for API web design, focusing on response times and data sizes. Two NodeJS apps performed CRUD (Create, Read, Update, Delete) operations on MongoDB. There are no major differences for a few queries or resource removal. GraphQL outperformed REST when displaying data under heavy loads and for small data portions, while REST performed better for large data portions. Reference [12] compared the performance of REST and GraphQL architectural models in three different applications based on two metrics, i.e., response time and data transfer rate. It found that GraphQL improved performance in most

cases, except for workloads above 3,000 requests, where REST performed better. For smaller workloads (100 requests), both REST and GraphQL showed similar performance. Reference [13] compares REST and GraphQL for data communication in web applications. An experiment was conducted to assess the performance of both approaches when requesting nested objects. The results indicate that GraphQL outperformed REST in most scenarios. Reference [14] assesses these benefits in practice by migrating seven systems from standard REST-based APIs to GraphQL. The key finding is that GraphQL can significantly reduce the size of JavaScript Object Notation (JSON) documents returned by REST APIs, with a reduction of 94%.

In addition to studies on the performance of REST and GraphQL, there are several studies that discuss the performance of gRPC. Reference [15] explains microservices and gRPC, covering their workings, implementations, limitations, and applications. It relies on reliable online sources to demonstrate a microservice with gRPC servers. Reference [16] explores microservices architecture and its communication methods, primarily REST API and gRPC. It evaluates the pros and cons of both approaches and conducts a comparative analysis. It presents a decision-making framework for organizations to determine if adopting gRPC offers substantial benefits over REST for their architecture. Reference [17] explores the potential of gRPC for improving content delivery. The study aims to evaluate gRPC using the Goal Question Metric (GQM) methodology. The findings indicate that gRPC performs exceptionally well in scenarios involving mobile or IoT applications as clients. Reference [18] discusses load balancing challenges in gRPC microservices within Kubernetes using Golang. Reference [19] proposes a solution for building gRPC services using NodeJS as independent modules or components. Reference [20] focuses on analyzing emerging technologies for cross-process communication between Linux and Android-based platforms using the gRPC framework. The study involves developing applications in various object-oriented programming languages to perform remote procedure calls between a single-board computer and a smartphone. The performance of computational offloading for algorithms in each platform is evaluated through data analysis.

Our study focuses on the performance comparison of REST, GraphQL, and gRPC in microservice environments to provide valuable insights into their respective advantages and drawbacks. We aim to reveal which communication protocols operate efficiently across various scenarios and workloads.

III. APPLICATION PROGRAMMING INTERFACE (API) PROTOCOLS

API protocols are sets of rules, conventions, and standards that facilitate communication and interaction between diverse software programs and systems. These protocols define the structure and format of requests and responses, as well as the methods and rules for communication. The API acts as a bridge that allows developers to integrate functionality. The most commonly used API protocols are REST, GraphQL, and gRPC.

A. Representational State Transfer (REST)

REST is an API development architecture that provides client-server-based communication over the HTTP protocol. REST was first introduced by Roy Fielding in 2000 as his doctoral dissertation at the University of California [21]. REST uses the HTTP/1.1 protocol to send data from clients to servers. In systems that use REST, each service usually has a certain endpoint so that it can interact between services and exchange data. In REST, there are several methods that can be used, including GET, POST, PUT, and DELETE. The REST supports several formats for presenting data, such as JSON and XML. JSON is used more often because of its simplicity and efficiency. Figure 1 shows the REST communication model.
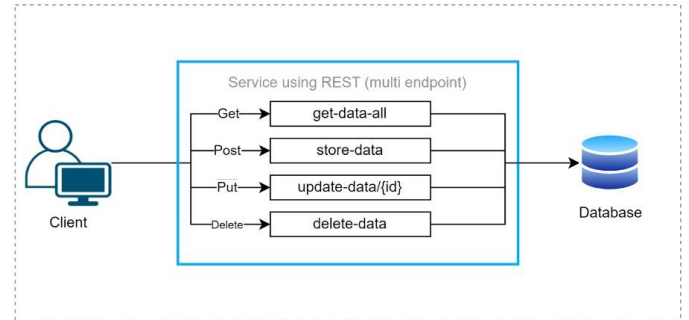

Fig. 1. REST Model

B. Graph Query Language (GraphQL)

GraphQL, a query language for APIs, was created by Facebook and used in communication between clients and servers [22]. The client requests data as needed with a query so the server can return a response according to the query request from the client. GraphQL offers an alternative solution to REST and allows developers to request specific data in a more efficient and flexible format. The background of the development of GraphQL was to meet Facebook's needs in handling complex data and to overcome the problems in the REST, such as over-fetching or under-fetching data. One of the main advantages of GraphQL is its flexibility. With GraphQL, clients can request multiple data sources in a single request, reducing the requests needed to retrieve the desired data. In addition, clients can validate their query requests by using clearly defined types before sending them to the server. Figure 2 shows the GraphQL communication model.

C. Google Remote Procedure Call (gRPC)

gRPC [23] is an open-source, high-performance framework for building efficient, distributed systems and microservices. It was developed by Google and designed to enable communication between applications and services in a way that is both language-agnostic and platform-independent. The gRPC allows applications to define their service methods and data structures using Protocol Buffers (protobufs), a language-neutral interface definition language. Based on these definitions, it generates client and server code in multiple programming languages. Clients and servers can then communicate
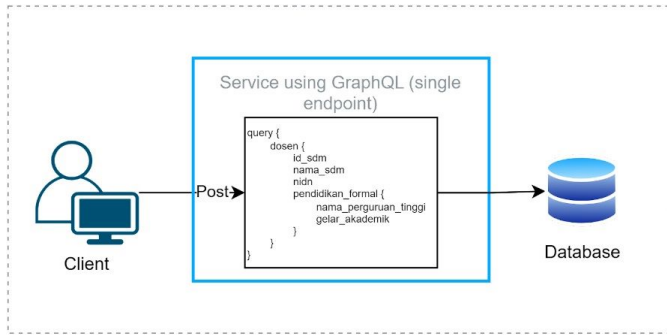
Fig. 2. GraphQL Model

using HTTP/2, benefiting from features like bidirectional streaming, multiplexing, and efficient serialization. It is a high-performance framework for building efficient and language-agnostic distributed systems, microservices, and APIs. Figure 3 shows the gRPC communication model.
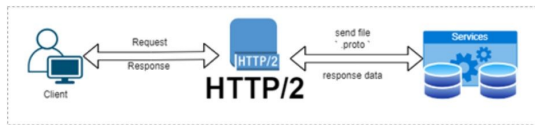

Fig. 3. gRPC Model

IV. SYSTEM DESIGN

In this study, we have developed microservices using Golang, with the case study of the Integrated Education Information System in the Ministry of Education and Culture Indonesia, known as SISTER. It is designed to manage and monitor resources in the education sector, including data related to academic institutions, research, and human resources. This system aims to improve the efficiency and effectiveness of educational management and administration.

Our study utilizes Hasanuddin University's SISTER data, specifically lecturer data and the lecturer's educational background. This research aims to assess the performance of REST, gRPC, and GraphQL. Our proposed service architecture consists of three services implemented in three containers, including authentication services, fetching lecturer profiles service (flat data), and fetching lecturer profile with educational background service (nested data). Each service contained a Redis and MySQL database. Figure 4 shows the architecture of our system.

The evaluation comprises two distinct data retrieval, i.e., fetching flat data and nested data. Fetching flat data refers to JSON structures where all the data is organized at the same level, typically using simple key-value pairs. On the other hand, fetching nested data refers to JSON structures where data is organized hierarchically, with one or more items containing other items as properties or elements.

This study utilizes Redis and MySQL as Database Management Systems, with MySQL as the long-term storage solution and Redis as the in-memory storage system. Being

an in-memory database, Redis excels in read-heavy operations and is ideal for our use cases demanding low-latency data access. Initially, we imported the SISTER data, available at http://sister.unhas.ac.id/ws.php/1.0 into the MySQL database. This imported data includes lecturer profile data totaling 2,221 entries and lecturer profiles with educational backgrounds, which amounts to 6,197 entries. Subsequently, Redis exported the SISTER data from the MySQL database. The service fetches data from Redis rather than directly from the MySQL database in the data retrieval process. In cases where Redis lacks the required data, the service retrieves it from the MySQL database and caches it within Redis. During testing, data retrieval occurs from Redis. Figure 5 illustrates the data fetching process within the services.

Figure 6 shows that JSON represents flat data. It includes a single object within an array, with several key-value pairs. Figure 7 shows that JSON represents nested data within the "pendidikan_formal" array to represent the lecturer's educational backgrounds. The data in Figures 6 and 7 are dummy data, used to avoid exposing sensitive information.

V. PERFORMANCE EVALUATION AND RESULT

Performance evaluation has been conducted to assess the impact of data fetching load on response time and CPU performance. This evaluation aims to evaluate the data exchange with REST, gRPC, and GraphQL to determine the most suitable approach for both flat data and nested data cases. We used Apache JMeter for API load testing. The Apache JMeter application is open source software designed to conduct load test on functional capabilities and assess performance [24].

A. Concurrent Requests Evaluation

In concurrent requests evaluation, multiple clients initiate requests concurrently, ranging from 100 to 500 requests, to assess response times and CPU utilization under these simultaneous load conditions. This approach allows us to gauge how the system performs when subjected to varying levels of concurrent user activity. Response time measurements were conducted for both the fetching flat data and nested data. Each evaluation was carried out over ten iterations. The average response time ($aveRT$) can be calculated using equation (1) :

$$aveRT = \frac{1}{n}\sum_{i=1}^{n}(t_{resp_i} - t_{req_i}) \qquad (1)$$

In equation (1), n represents the total number of requests, t_{req_i} is the request time of the i-th request, and t_{resp_i} is the the response time of the i-th request. The equation calculates the average time interval required for the client to receive a response from the sent request.

Figure 8 shows the average response time for fetching flat data. For REST, the average response times increase as the number of requests increases, ranging from 1,113.33 ms for 100 requests to 4,009.83 ms for 500 requests. gRPC offers significantly lower response times, with averages ranging from 233.84 ms for 100 requests to 2,606.59 ms for 500 requests. On the other hand, GraphQL shows the highest response times,
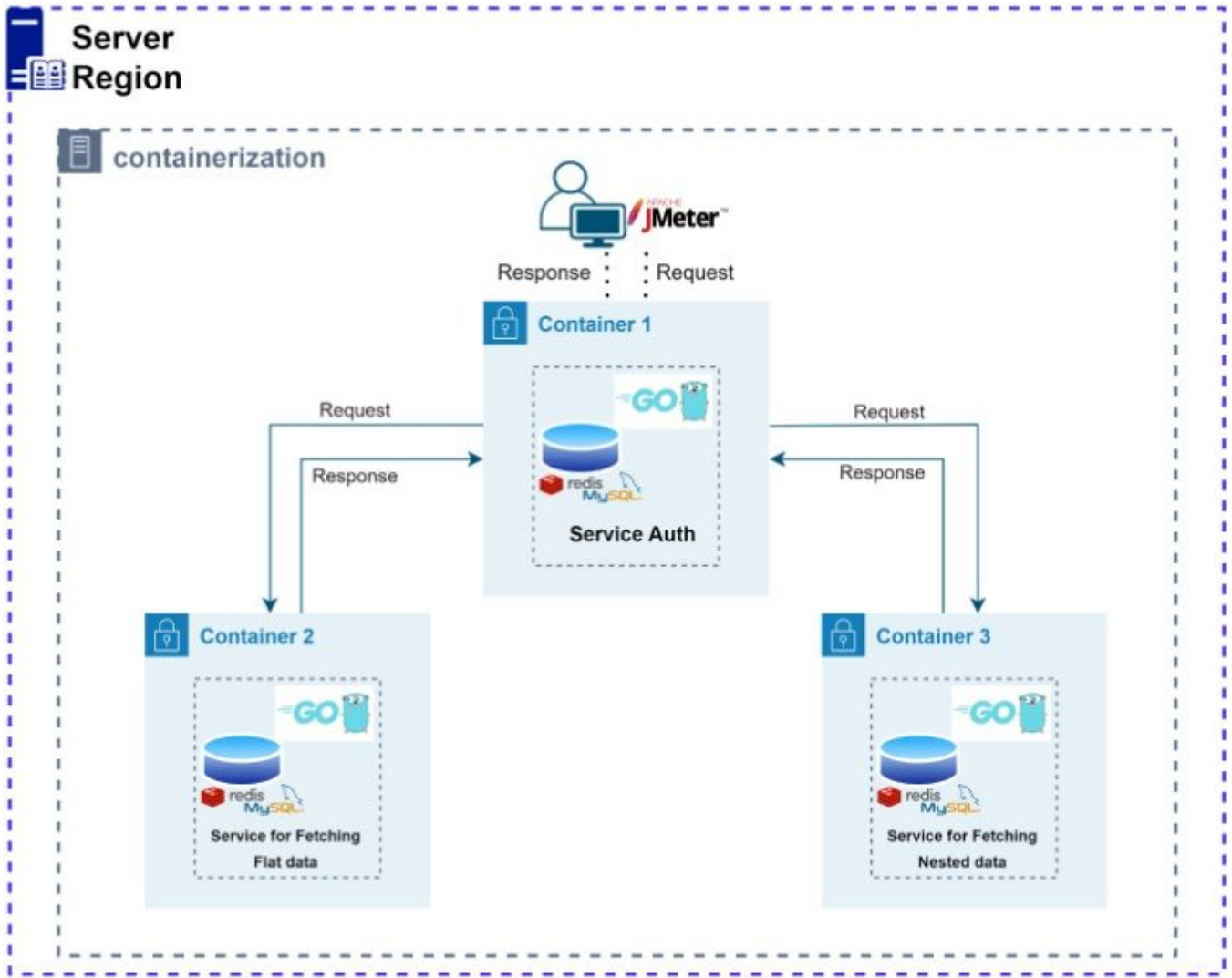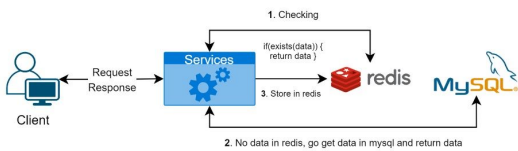
Fig. 4. System Architecture

Fig. 5. Data Fetching Process

Fig. 6. JSON for Fetching Flat Data

Fig. 7. JSON for Fetching Nested Data

with averages increasing from 3,852.07 ms for 100 requests to 21,148.14 ms for 500 requests. In summary, gRPC provides the fastest response times, followed by REST, while GraphQL lags with substantially slowest response times, particularly as the request volume increases.

Figure 9 shows the average response time for fetching nested data. For REST, as the number of requests increased from 100 to 500, the average response times grew from 5,201.39 ms to 16,646.55 ms. In the case of gRPC, the response times also increased with more requests, ranging
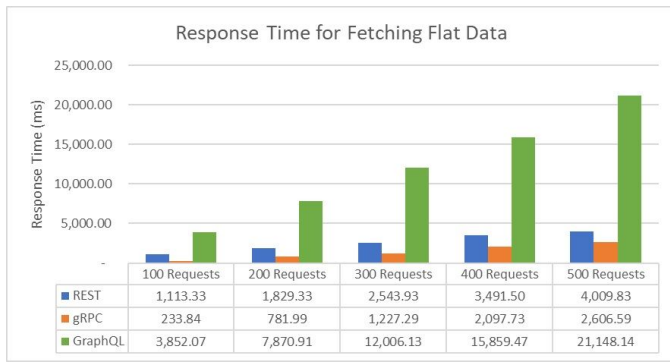
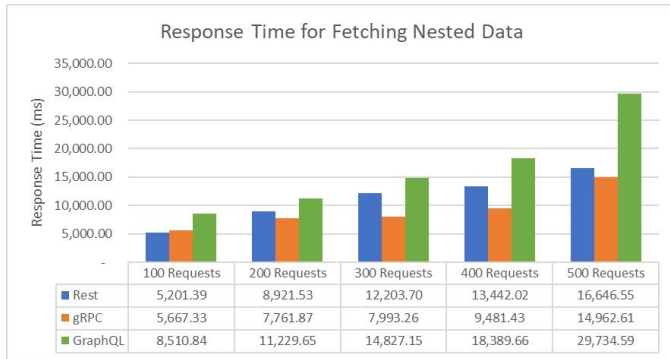Fig. 8. Average response time for Fetching Flat Data

	100 Requests	200 Requests	300 Requests	400 Requests	500 Requests
REST	1,113.33	1,829.33	2,543.93	3,491.50	4,009.83
gRPC	233.84	781.99	1,227.29	2,097.73	2,606.59
GraphQL	3,852.07	7,870.91	12,006.13	15,859.47	21,148.14

Fig. 9. Average response time for Fetching Nested Data

	100 Requests	200 Requests	300 Requests	400 Requests	500 Requests
Rest	5,201.39	8,921.53	12,203.70	13,442.02	16,646.55
gRPC	5,667.33	7,761.87	7,993.26	9,481.43	14,962.61
GraphQL	8,510.84	11,229.65	14,827.15	18,389.66	29,734.59

from 5,667.33 ms to 14,962.61 ms. GraphQL showed the highest response times, averaging 8,510.84 ms to 29,734.59 ms as the number of requests increased. Overall, REST had the lowest response times, followed by gRPC, while GraphQL exhibited the slowest response times, particularly with a larger number of requests.

We also measured CPU utilization to assess the impact of data fetching load on CPU performance. We aim to gauge how the act of fetching data, whether it involves retrieving flat or nested data, influences the CPU utilization. We examine CPU performance across a range of scenarios, each involving a varying number of data retrieval requests, spanning from 100 to 500 requests.
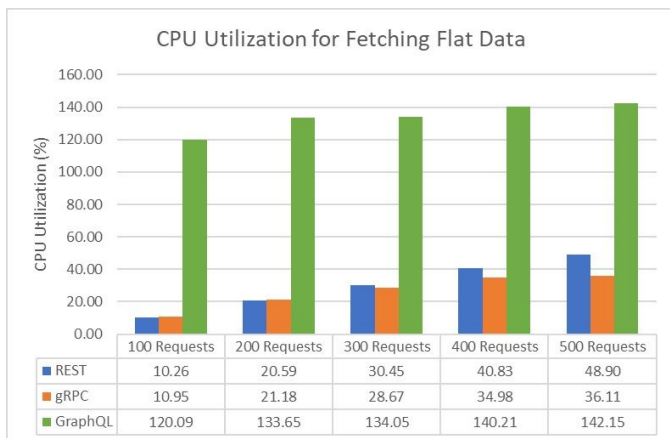

	100 Requests	200 Requests	300 Requests	400 Requests	500 Requests
REST	10.26	20.59	30.45	40.83	48.90
gRPC	10.95	21.18	28.67	34.98	36.11
GraphQL	120.09	133.65	134.05	140.21	142.15

Fig. 10. Average CPU Utilization for Fetching Flat Data

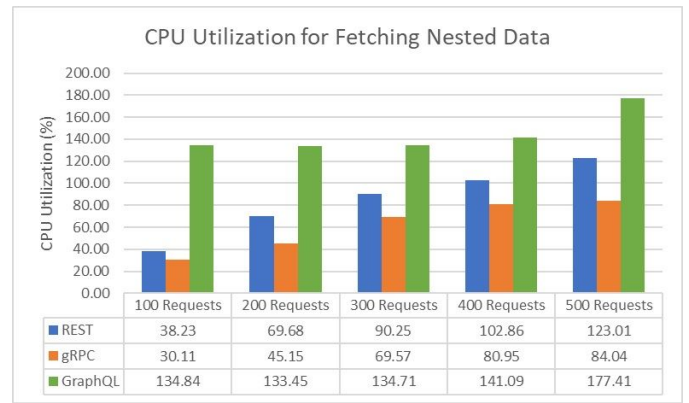|	100 Requests	200 Requests	300 Requests	400 Requests	500 Requests
REST	38.23	69.68	90.25	102.86	123.01
gRPC	30.11	45.15	69.57	80.95	84.04
GraphQL	134.84	133.45	134.71	141.09	177.41

Fig. 11. Average CPU Utilization for Fetching Nested Data

Figure 10 shows the average CPU utilization for fetching flat data. For REST requests, as the number of requests increased from 100 to 500, CPU utilization gradually increased from 10.26% to 48.90%. With gRPC requests, CPU utilization also increased with the number of requests, going from 10.95% to 36.11%. However, for GraphQL requests, CPU utilization exhibited a different trend, starting remarkably high at 120.09% for 100 requests and gradually increasing to 142.15% for 500 requests. These figures highlight the varying CPU resource demands of different data request protocols. GraphQL was notably more resource-intensive than REST and gRPC, showing increased linear CPU utilization with increasing request loads.

Figure 11 shows the average CPU utilization for fetching nested data. For REST, CPU utilization increased from 38.23% at 100 requests to 123.01% at 500 requests. gRPC had lower CPU utilization, starting at 30.11% and reaching 84.04% at 500 requests. In contrast, GraphQL showed significantly higher CPU utilization, exceeding 100% even at 100 requests and peaking at 177.41% at 500 requests. This indicates that the workload is being distributed across at least two CPU cores, suggesting higher processing demands for GraphQL queries than REST and gRPC as the request load increased.

B. Consecutive Requests Evaluation

In the consecutive request evaluation, clients initiate requests consecutively for five minutes with a varying number of requests, from 100 to 500 requests, to measure response time and CPU utilization during the test.

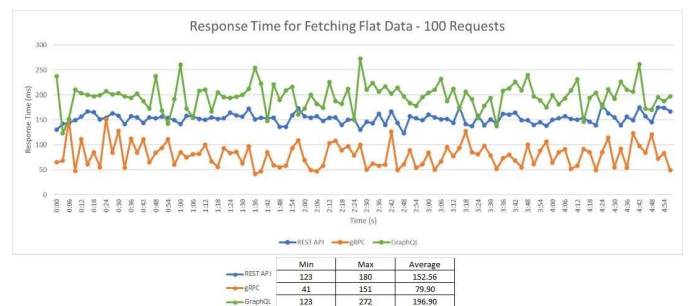|	Min	Max	Average
REST API	123	180	152.56
gRPC	41	151	79.90
GraphQL	123	272	196.90

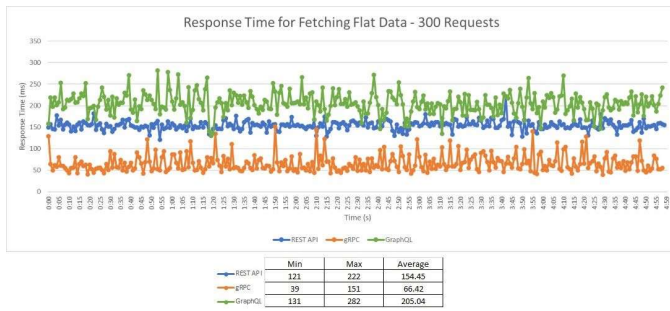Fig. 12. Response time during five minutes for Fetching Flat Data (100 requests)

Fig. 13. Response time during five minutes for Fetching Flat Data (300 requests)
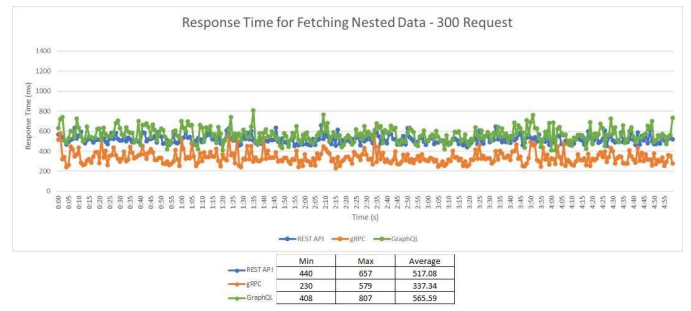

Fig. 14. Response time during five minutes for Fetching Flat Data (500 requests)
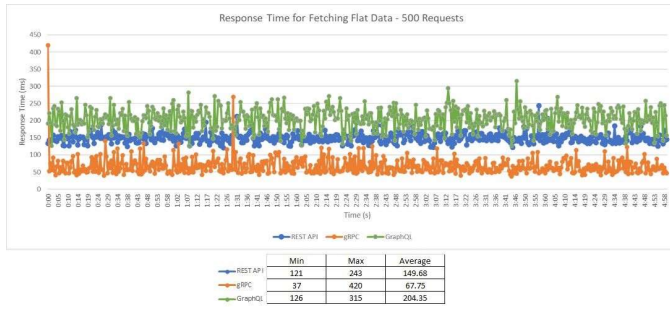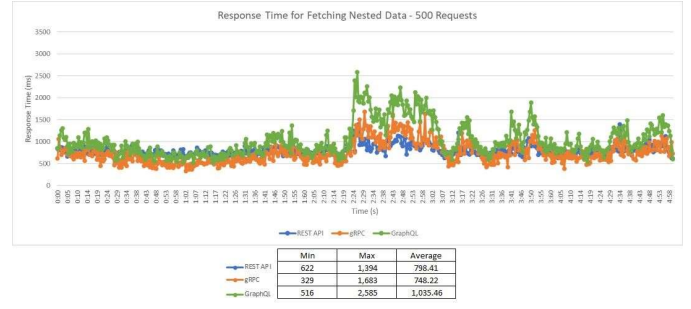
Figures 12, 13, and 14 show that the response time of gRPC is faster than REST and GraphQL during five minutes measurements for fetching flat data. For 100 requests (Figure 12), gRPC performed the fastest, with an average response time of 79.90 ms, followed by REST at 152.56 ms and GraphQL at 196.90 ms. When the number of requests increased to 300 (Figure 13), gRPC remained the fastest with 66.42 ms, while REST and GraphQL showed slight increases in response times with an average of 154.45 ms and 205.04 ms, respectively. However, when the number of requests further increased to 500 (Figure 14), GraphQL had the highest average response time at 204.35 ms, while gRPC and REST had response times of 67.75 ms and 149.68 ms, respectively, with gRPC being the fastest.

Fig. 15. Response time during five minutes for Fetching Nested Data (100 requests)

Figures 15, 16, and 17 show that the response time of gRPC is faster than REST and GraphQL during five minutes measurements for fetching nested data. For 100 requests

Fig. 16. Response time during five minutes for Fetching Nested Data (300 requests)

Fig. 17. Response time during five minutes for Fetching Nested Data (500 requests)

(Figure 15), gRPC performed the fastest, with an average response time of 437.03 ms, followed by REST at 510.47 ms and GraphQL at 589.25 ms. When the number of requests increased to 300 (Figure 16), gRPC remained the fastest with 337.34 ms, while REST and GraphQL showed slight increases in response times with an average of 517.08 ms and 565.59 ms, respectively. However, when the number of requests further increased to 500 (Figure 17), GraphQL had the highest average response time at 1,035.46 ms, while gRPC and REST had response times of 748.22 ms and 798.41 ms, respectively, with gRPC being the fastest.

We also measured the CPU utilization for five minutes for each data fetching scenario with a different number of requests (100, 300, and 500 Requests).
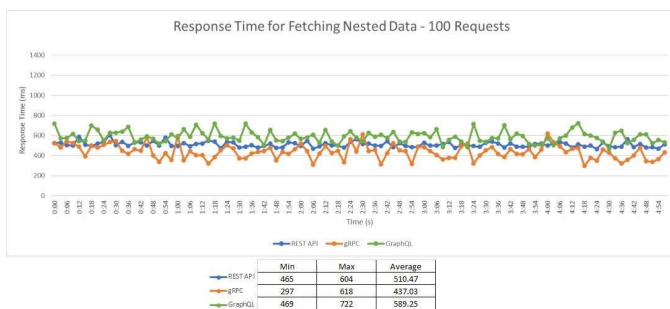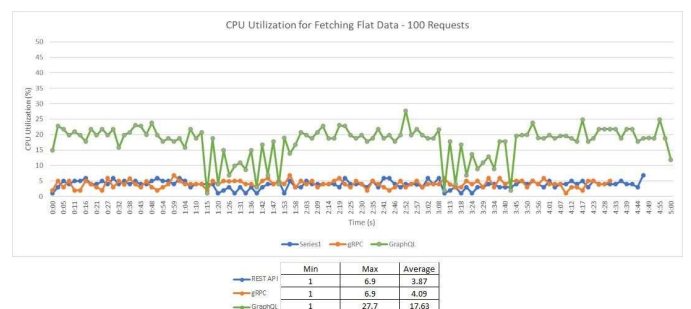

Fig. 18. CPU Utilization during five minutes for Fetching Flat Data (100 requests)

Figures 18, 19, and 20 show the CPU utilization of three API protocols during five minutes measurement for fetching flat data. With 100 requests (Figure 18), REST had the lowest
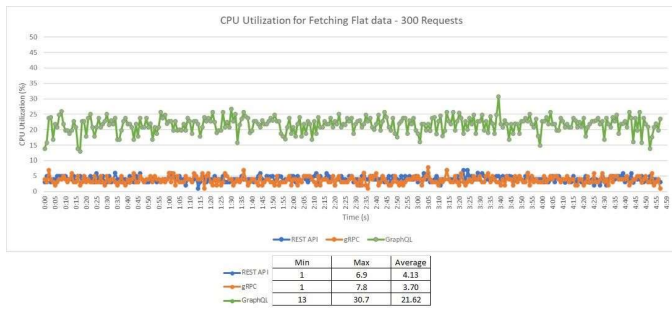
Fig. 19. CPU Utilization during five minutes for Fetching Flat Data (300 requests)
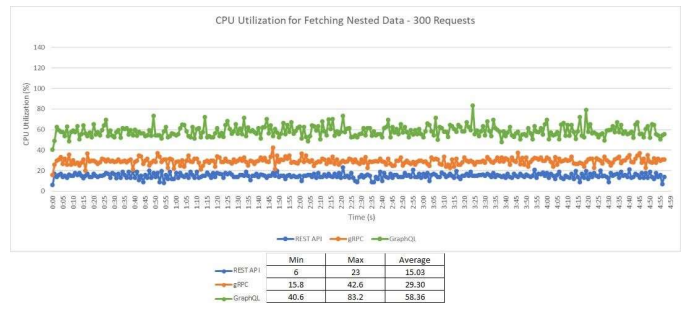

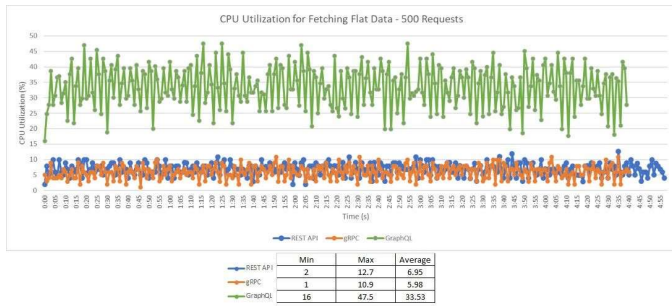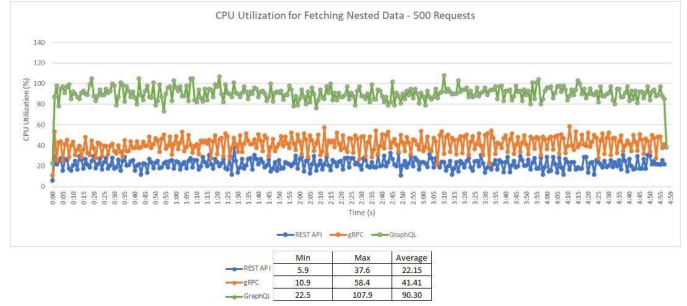Fig. 20. CPU Utilization during five minutes for Fetching Flat Data (500 requests)

CPU utilization with an average of 3.87%, gRPC was slightly higher with an average of 4.09%, and GraphQL had the highest utilization, averaging 17.63%. As the request count increased to 300 (Figure 19), REST's CPU utilization increased with an average of 4.13%, gRPC decreased with an average of 3.70%, and GraphQL spiked with an average of 21.62%. At 500 requests (Figure 20), REST's CPU utilization increased with an average of 6.95%, gRPC with an average of 5.98%, and GraphQL had the highest CPU utilization, averaging 33.53%. These results suggest that GraphQL places a heavier load on the CPU as the number of requests grows compared to REST and gRPC.
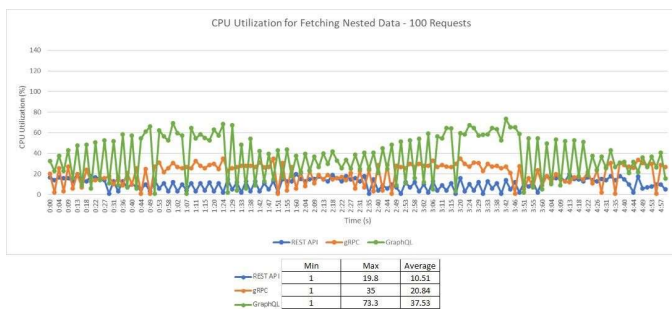

Fig. 21. CPU Utilization during five minutes for Fetching Nested Data (100 requests)

Figures 21, 22, and 23 show the CPU utilization of three API protocols during five minutes measurement for fetching nested data. For 100 requests (Figure 21), REST had the lowest CPU utilization with an average of 10.51%, followed by gRPC with an average of 20.84%, and GraphQL had

Fig. 22. CPU Utilization during five minutes for Fetching Nested Data (300 requests)

Fig. 23. CPU Utilization during five minutes for Fetching Nested Data (500 requests)

the highest, averaging 37.53%. As the request load increased to 300 (Figure 22) and 500 (Figure 23) requests, the CPU utilization also increased across all three protocols. REST maintained the lowest utilization, gRPC in the middle, and GraphQL consistently had the highest CPU utilization with an average of 90.30% for 500 requests.

The performance evaluation shows that gRPC outperformed REST and GraphQL in terms of response time, while REST demonstrated lower CPU utilization compared to the others. gRPC's superior performance can be largely attributed to its use of the HTTP/2 protocol, which offers several advantages over the HTTP/1 protocol utilized by both REST and GraphQL. HTTP/2 enables more efficient data exchange through features like multiplexing and reduced latency, making it a key factor in gRPC's enhanced performance when compared to its counterparts.

VI. CONCLUSION

Microservice architecture is now the prevailing framework for developing software systems that are both scalable and easy to maintain. The selection of the proper communication protocol within microservices is essential for attaining the best possible system performance. This research evaluates the performance of API protocols: REST, gRPC, and GraphQL in a microservices-based system using Redis and MySQL as databases. Two distinct data retrieval were examined: fetching flat data and nested data. Based on the evaluation of response time and CPU utilization for fetching flat and nested data scenarios, gRPC had the fastest response time, while REST showed the lowest CPU utilization compared to gRPC and GraphQL. gRPC's use of HTTP/2, with its

multiplexing feature, allows multiple requests over a single connection, making it highly efficient for concurrent remote procedure calls. These findings provide insights for choosing API protocols in microservices environment.

REFERENCES

[1] I. Karabey Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *Journal of Systems and Software*, vol. 180, p. 111014, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001114

[2] G. S. M. Diyasa, G. S. Budiwitjaksono, H. A. Ma'rufi, and I. A. W. Sampurno, "Comparative analysis of rest and graphql technology on nodejs-based api development," *Nusantara Science and Technology Proceedings*, pp. 43–52, Apr. 2021. [Online]. Available: https://nstproceeding.com/index.php/nuscientech/article/view/322

[3] M. Vesić and N. Kojić, "Comparative analysis of web application performance in case of using rest versus graphql," in *Proceedings of the Fourth International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture (ITEMA), Online-Virtual*, 2020, pp. 17–24. [Online]. Available: https://doi.org/10.31410/ITEMA.2020.17

[4] Y. Lee and Y. Liu, "Using refactoring to migrate rest applications to grpc," in *Proceedings of the 2022 ACM Southeast Conference*, ser. ACMSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 219–223. [Online]. Available: https://doi.org/10.1145/3476883.3520220

[5] N. Vohra and I. B. K. Manuaba, "Implementation of rest api vs graphql in microservice architecture," in *2022 International Conference on Information Management and Technology (ICIMTech)*. IEEE, 2022, pp. 45–50. [Online]. Available: https://doi.org/10.1109/ICIMTech55957.2022.9915098

[6] S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, "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)*. IEEE, 2021, pp. 10–17. [Online]. Available: https://doi.org/10.1109/SER-IP52554.2021.00009

[7] A. Lawi, B. L. Panggabean, and T. Yoshida, "Evaluating graphql and rest api services performance in a massive and intensive accessible information system," *Computers*, vol. 10, no. 11, p. 138, 2021. [Online]. Available: https://doi.org/10.3390/computers10110138

[8] B. Lama, "Implementing graphql in existing rest api," B.S. thesis, Universitat Politècnica de Catalunya, 2019.

[9] M. Vogel, S. Weber, and C. Zirpins, "Experiences on migrating restful web services to graphql," in *Service-Oriented Computing– ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13–16, 2017, Revised Selected Papers*. Springer, 2018, pp. 283–295. [Online]. Available: https://doi.org/10.1007/978-3-319-91764-1_23

[10] P. Margański and B. Pańczyk, "Rest and graphql comparative analysis," *Journal of Computer Sciences Institute*, vol. 19, pp. 89–94, 2021. [Online]. Available: https://doi.org/10.35784/jcsi.2473

[11] M. Mikuła and M. Dzieńkowski, "Comparison of rest and graphql web technology performance," *Journal of Computer Sciences Institute*, vol. 16, pp. 309–316, 2020. [Online]. Available: https://doi.org/10.35784/jcsi.2077

[12] M. Seabra, M. F. Nazário, and G. Pinto, "Rest or graphql? a performance comparative study," in *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, 2019, pp. 123–132. [Online]. Available: https://doi.org/10.1145/3357141.3357149

[13] M. D. C. França and E. da Silva, "Performance evaluation of rest and graphql apis searching nested objects," *Anais do Computer on the Beach*, vol. 11, no. 1, pp. 237–244, 2020. [Online]. Available: https://doi.org/10.14210/cotb.v11n1.p237-244

[14] G. Brito, T. Mombach, and M. T. Valente, "Migrating to graphql: A practical assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 140–150. [Online]. Available: https://doi.org/10.1109/SANER.2019.8667986

[15] H. Vo, "Applying microservice architecture with modern grpc api to scale up large and complex application," 2021. [Online]. Available: https://urn.fi/URN:NBN:fi:amk-2021060314024

[16] M. Stefanic, "Developing the guidelines for migration from restful microservices to grpc," *Masaryk University, Faculty of Informatics, Brno*, pp. 1–81, 2021. [Online]. Available: https://is.muni.cz/th/ozxws/

[17] B. P. Rebrošová, "grpc layer for content delivery in kentico kontent," Master's thesis, Masaryk University, 2021. [Online]. Available: https://is.muni.cz/th/d1f9l/Masters_Thesis.pdf

[18] K. Nieman and S. Sajal, "A comparative analysis on load balancing and grpc microservices in kubernetes," in *2023 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, 2023, pp. 322–327. [Online]. Available: https://doi.org/10.1109/IETC57902.2023.10152023

[19] M. Vasiljević, A. Manasijević, A. Kupusinac, Ć. Sukić, and D. Ivetić, "One solution of component based development in nodejs for modularization of grpc services and rapid prototyping," *SAR J*, vol. 2, pp. 181–185, 2019. [Online]. Available: https://doi.org/10.18421/SAR24-06

[20] M. Araújo, M. E. Maia, P. A. Rego, and J. N. De Souza, "Performance analysis of computational offloading on embedded platforms using the grpc framework," in *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, 2020, pp. 1–8. [Online]. Available: https://hal.science/hal-02495252

[21] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UC Irvine, 2000. [Online]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

[22] The GraphQL Foundation. (2015) "graphql",. Accessed: September 26, 2023. [Online]. Available: https://graphql.org/

[23] Louis Ryan (Google). (2015) "grpc",. Accessed: September 16, 2023. [Online]. Available: https://grpc.io

[24] The Apache Software Foundation. (n.d) "apache jmeter",. Accessed: October 23, 2022. [Online]. Available: https://jmeter.apache.org/