JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 36 (2025) 350–356
Received: 21 May 2025
Accepted: 8 July 2025

Performance evaluation of REST and GraphQL API aproaches in data retrieval scenarios using NestJS

Ocena wydajności podejść API REST i GraphQL w scenariuszach pobierania danych z zastosowaniem NestJS

Kacper Stępień*, Maria Skublewska-Paszkowska

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The main aim of the study is to compare the performance of two API approaches, REST and GraphQL, in the context of data retrieval. Two applications with identical functionality have been developed in NestJS using a PostgreSQL database. Performance tests have been carried out using Grafana k6, simulating loads from 1,000 to 24,000 users. REST achieves better response times and throughput in simple queries from a single table. GraphQL shows better performance in scenarios involving complex queries from four related tables. In scenarios involving partial field selection, GraphQL returns significantly smaller responses – up to 94% smaller than REST. The results indicate that REST is more efficient in simple and high-load scenarios, while GraphQL performs better in complex data structures.

Keywords: REST; GraphQL; NestJS; API performance

Streszczenie

Celem badania jest porównanie wydajności dwóch podejść do projektowania API, REST i GraphQL, w kontekście pobierania danych. Opracowano dwie identyczne aplikacje w NestJS z bazą danych PostgreSQL. Testy wydajności przeprowadzono za pomocą narzędzia Grafana k6 przy obciążeniu od 1 000 do 24 000 użytkowników. REST uzyskał lepsze czasy odpowiedzi i przepustowość w prostych zapytaniach do jednej tabeli. GraphQL osiągnął lepsze wyniki w złożonych zapytaniach obejmujących cztery powiązane tabele. W przypadku pobierania tylko wybranych pól, rozmiar odpowiedzi GraphQL był znacznie mniejszy – nawet o 94% względem REST. Wyniki pokazują, że REST lepiej sprawdza się w prostych scenariuszach i przy dużym obciążeniu, a GraphQL w strukturach złożonych.

Słowa kluczowe: REST; GraphQL; NestJS; wydajność API

*Corresponding author

Email address: s95574@pollub.edu.pl (K. Stępień)

1. Introduction

Contemporary web applications are typically designed based on a multi-layer architecture that includes a frontend, a backend, and a database. The Application Programming Interface (API) is the main component of communication between these layers. Representational State Transfer (REST) is one of the most accepted API approaches, proposed by Roy Fielding in 2000 [1]. REST is a set of architectural principles based on the HTTP protocol and its standard operations, i.e., GET, POST, PUT, and DELETE. Due to its simplicity in deployment, broad reachability, and scalability, REST has become the de facto standard for API design.

However, REST also has certain drawbacks. Common problems are over-fetching, when the client receives more data than needed, and under-fetching, when needed data requires several requests. One response to this was GraphQL – an open standard developed by Facebook in 2015 [2].

GraphQL is a query language for APIs that enables clients to request only the data they need, providing a more efficient and flexible alternative [2]. It uses one endpoint and supports operations such as query, mutation, and subscription. Queries are executed via POST in the body of the request to enhance flexibility and prevent URL length issues. Being declarative, GraphQL reduces unnecessary data and gives more control over the structure received. This is especially beneficial in mobile apps and systems with many relationships.

Due to these characteristics, GraphQL is increasingly being adopted as an alternative to REST, especially in situations where complex, hierarchical data need to be fetched.

1.1. The aim and object of the research

The main aim of this study is to analyze and compare the performance of two API implementations – REST and GraphQL in terms of response time, data transfer size, and throughput. To achieve this, two applications were developed using the NestJS framework, both utilizing the same database. Performance tests were conducted using the Grafana k6 tool. Based on the reviewed literature comparing REST and GraphQL, the following research hypotheses were formulated and examined in this article:

- GraphQL provides shorter response times in queries involving complex, nested data structures,
- for simple single-record reads, REST and GraphQL demonstrate comparable performance.

1.2. Related work

There have been several comparisons of REST and GraphQL API performance in different contexts of use. Researchers have examined when GraphQL can be utilized as a substitute for REST and where using it is beneficial.

The majority of works highlight REST's superiority when dealing with simple operations. Studies such as Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC [3] and Comparative Review of Selected Internet Communication Protocols [4] show that REST always yields faster response times. In Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System [5], REST outperformed GraphQL by 51% in response time and by 37% in throughput, which was confirmed in Performance Analysis of GraphQL and RESTful in SIM LP2M of Hasanuddin University [6].

While REST is superior in simple, single-entity queries, GraphQL certainly has its strengths in querying relational and complex data. The study Comparison of REST and GraphQL Interfaces for OPC UA [7] refers to the optimality of REST in basic operations, while GraphQL is superior in retrieving nested data. The results coincide with Performance Measurement of GraphQL API in Home ESS Data Server [8], which also shows GraphQL's performance in fetching dependent data with a single query.

Other research, including REST and GraphQL Comparative Analysis [9] and Performance Analysis of Database Access: Comparison of Direct Connection, ORM, REST API and GraphQL Approaches [10], confirms REST's better performance with high loads. Similarly, Implementation of REST API vs GraphQL in Microservice Architecture [11] concluded that REST performed higher throughput and better stability with high volumes of data.

GraphQL's ability to avoid over-fetching makes it better in scenarios where bandwidth optimization is critical. GraphQL for the Delivery of Bioinformatics Web APIs and Application to ZincBind [12] shows that GraphQL can readily reduce response size. This advantage is supported by An Initial Analysis of Facebook's GraphQL Language [13] and Experiences on Migrating RESTful Web Services to GraphQL [14], noting that clients receive just the necessary data, thereby improving performance. This has also been established in Design and Implementation of Online Legal Forum to Complain and Track UGC Cases using NextJs and GraphQL [15] and Migrating to GraphQL: A Practical Assessment [16].

Additional proof from the studies Comparison of REST and GraphQL Web Technology Performance [17] and Leveraging GraphQL for Large-Scale Queries on Digital Twins in Industry 4.0 [18] indicates that GraphQL is very efficient in the case of having a huge and complicated data structure. This is because it can return tailored responses assembled from various sources within one query.

GraphQL is also appreciated for its ability to support dynamic data structures, as referenced in A Low-Code Approach for Data View Extraction from Engineering Models with GraphQL [19], in which its flexibility was helpful for managing heterogeneous data. Although GraphQL is being ever more utilized, its adoption is often more demanding. GraphQL: A Systematic Mapping Study [20] and Can GraphQL Replace REST? A Study of Their Efficiency and Viability [21] notes that, while efficient, GraphQL may be more labor-intensive to set up. Moreover, Experiences on Migrating RESTful Web Services to GraphQL [14] highlights integration problems in distributed systems.

In short, each has its advantages and disadvantages. The complexity of the application, system load, and the specific requirements of the users should dictate when to utilize REST or GraphQL.

2. Research Methodology

2.1. Test Applications

To conduct the performance analysis, two applications with identical functionality were developed: one based on REST approach, and the other on GraphQL. Both were built using the NestJS framework (version 10.0.0). The GraphQL application utilized the @nestjs/apollo, @apollo/server, and dataloader packages to manage and execute queries. Data were stored in a PostgreSQL relational database (version 16.2), with TypeORM (version 0.3.20) as the ORM and the pg package (version 8.13.3) as the database driver. Both applications were designed using the Northwind schema, obtained from a public repository [22]. The database consists of 15 related tables, including orders, order_details, products and suppliers (Fig. 1). Only four of these tables were utilized for testing purposes, but the whole database was provided in its original format to ensure consistency and ease of configuration.

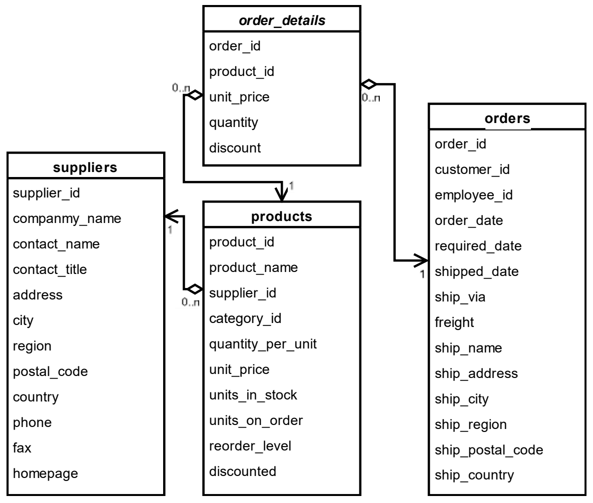

Figure 1: Fragment of the entity-relationship diagram (ERD) for the Northwind database.

2.2. Testing Environment

The experiments were conducted using a single laptop that served both as the server running the tested applications and as the client executing the load tests. Utilizing a single machine helped eliminate the influence of network latency on the results. The hardware specifications of the test device are presented in Table 1. Since the operating system was Windows 11, both the applications and the test software were run under the Windows Subsystem for Linux (WSL), with Node.js version 20.19.0 on Ubuntu.

WSL was chosen for its superior handling of a high number of concurrent network connections, which was essential when simulating multiple virtual users. Additionally, due to Node.js's single-threaded execution model, both applications were launched in cluster mode using PM2, with four instances each. This setup enabled more effective use of the multi-core processor.

Table 1: Computer specification

Component	Specification
Processor	AMD Ryzen 7 7745HX 5.1 GHz, 8 cores, 16 threads
RAM	32 GB DDR5
Graphics Card	Nvidia RTX 4070 8GB
Storage	1 TB SSD
Operating System	Windows 11

2.3. Research Tool

To test the performance, the open-source tool Grafana k6 (version 0.57.0) was used. It allows users to model load-testing scenarios through JavaScript. The group feature in k6 enables packaging multiple requests into a single block for testing, providing the combined execution time. This feature is most helpful in simulating REST scenarios that comprise multiple consecutive requests. k6 has extensive configuration settings, such as the number of virtual users, test duration, and load stages, thus facilitating the execution of various performance testing methodologies.

2.4. Research Scenarios

The study involved two main test scenarios that varied in request complexity and structure. A detailed overview of these scenarios is presented below:

1. Fetching data from a single table orders in three variants: 1, 100, and 500 records. Tested implementations: REST, GraphQL (full structure), GraphQL with limited fields (only ID, ~7% of structure).
2. Fetching data from four related tables: orders, order_details, products, and suppliers. Tested implementations: REST (multiple endpoints), REST (combined endpoint), GraphQL (full structure), GraphQL with a reduced field set (~45% of structure).

Each scenario was executed under the following load conditions: 1,000, 2,000, 4,000, 6,000, 8,000, 10,000, 12,000, 14,000, 16,000, 18,000, 20,000, 22,000, and 24,000 concurrent virtual users. Each test ran for a fixed duration of one minute at a constant number of virtual users. During each test, a user waited exactly one second after receiving a response before sending the next request. After the one-minute period had elapsed, no new requests were generated, but all in-progress requests were allowed to complete, slightly extending the total duration.

To ensure consistent results, each configuration was executed 6 times and average values across all runs were calculated. This practice minimized the impact of random fluctuations in performance and allowed measurement repeatability. All tests were conducted automatically using custom Bash scripts, with a 10-second delay between iterations to allow for system stabilization. The laptop remained continuously plugged into power during tests, and all unnecessary applications were closed.

3. Results

3.1. Results of the First Test

The first test focused on retrieving data from a single table (orders) in three variations: retrieving 1, 100, and 500 records. The GraphQL implementation included a variant with a limited field set, retrieving only the order ID – approximately 7% of the full data schema. The results showing average response time versus the number of concurrent users are presented in Figures 2, 3, and 4.

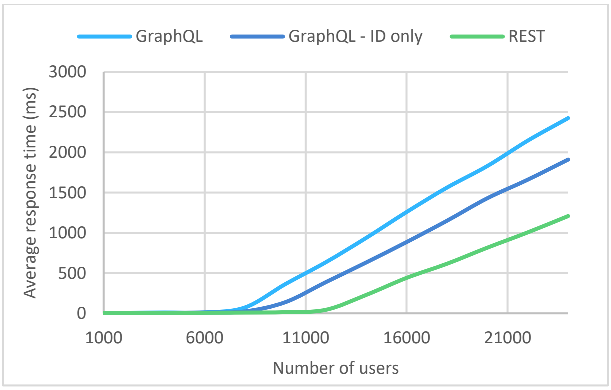

Figure 2: Average response time for a single-record query depending on the number of concurrent users.

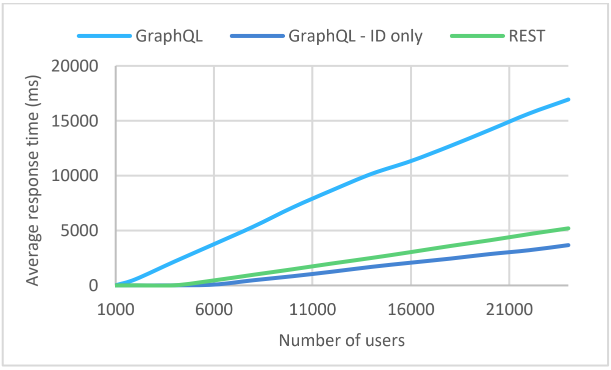

Figure 3: Average response time for a 100-record query depending on the number of concurrent users.

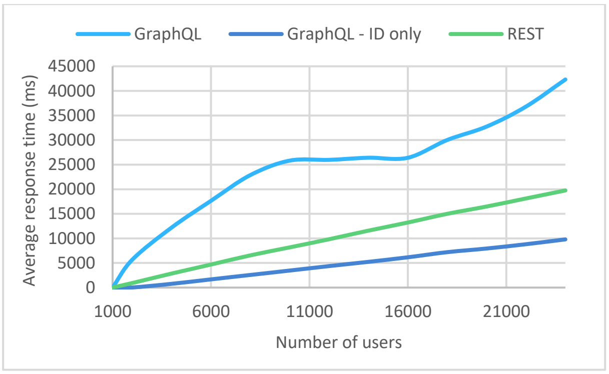

Figure 4: Average response time for a 500-record query depending on the number of concurrent users.

For single-record queries, all approaches exhibited similar performance under low load. With a group of 1,000 users, the average response times were as follows: REST – 3.37 ms, GraphQL – 4.49 ms, and GraphQL with a limited data scope – 3.63 ms. Beginning from 8,000 users, however, the performance of full GraphQL began to degrade significantly, with an average response time of 72.83 ms, while REST maintained 9.67 ms – over seven times faster. Under the maximum load of 24,000 users, the gap narrowed: GraphQL – 2,425 ms, REST – 1,208 ms. Notably, the limited GraphQL query had an average response time approximately 30% lower than the full query. Even with significantly reduced payload, GraphQL still performed worse than REST, which consistently provided the lowest response times across concurrency levels. The maximum throughput achieved in this scenario was as follows: REST – 10,896 requests per second, GraphQL (ID only) – 8,598 requests per second, and full GraphQL – 7,279 requests per second. The dependency of the number of users on throughput is presented in Figure 5.

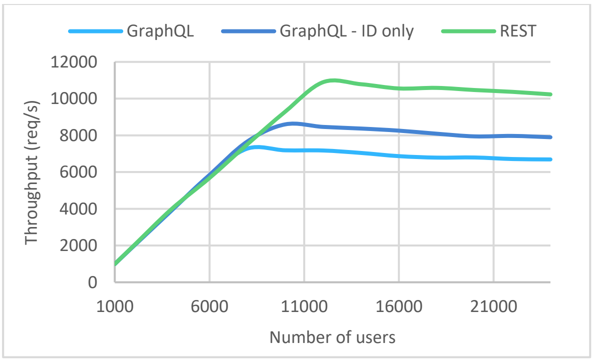

Figure 5: Throughput for 1-record queries depending on the number of concurrent users.

In the 100-record variant, complete GraphQL requests exhibited longer response times from the start. At 2,000 users, the average response time was 555 ms, compared to 6.72 ms for REST and 6.53 ms for the GraphQL (ID only). As user load increased, response times rose almost linearly across all approaches, with the biggest rise recorded for full GraphQL query. Under the maximum load of 24,000 users, the response times averaged: GraphQL – 16,940 ms, REST – 5,200 ms, and GraphQL (ID only) – 3,671 ms. The limitedGraphQL query was approximately 85% faster than full GraphQL query and nearly 30% faster than REST. This implies that the amount of data retrieved via GraphQL has a great impact on its operational cost and could be the key determining factor in overtaking the conventional REST approach. The maximum throughput achieved was as follows: REST with 3,928 requests per second, GraphQL (ID only) with 5,343 requests per second, and full GraphQL with 1,240 requests per second. Figure 6 shows the correlation between user load and throughput.

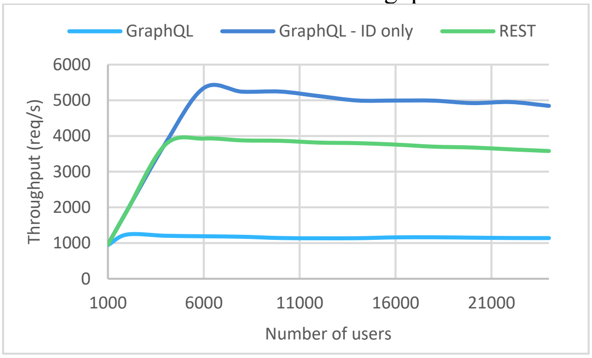

Figure 6: Throughput for 100-record queries depending on the number of concurrent users

The most demanding variant involved retrieving 500 records. Under a load of 24,000 users, full GraphQL queries averaged over 42 s, REST reached 19.7 s, and GraphQL (ID only) achieved 9.8 s. The optimized GraphQL query achieved a response time approximately 85% shorter than the full GraphQL query and 58% shorter than REST. Unlike the 100-record case, the performance gap between optimized GraphQL and REST increased from 30% to over 58%. These results suggest that as the size of the queried dataset increases, the advantages of GraphQL with a reduced field set become more pronounced when compared to both traditional REST and full GraphQL query. The maximum throughput achieved in this instance was: full GraphQL – 284 requests/s, GraphQL (ID only) – 2,174 requests/s, and REST – 1,012 requests/s. The relationship between user load and throughput is illustrated in Figure 7.

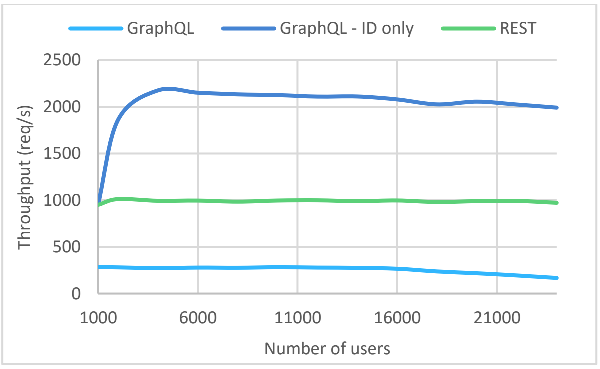

Figure 7: Throughput for 500-record queries depending on the number of concurrent users.

Apart from response time and throughput, another important factor in the comparison is the volume of data exchanged between the client and the server. Table 3 presents the data transferred per request according to the number of records retrieved. For single-record queries, the differences were relatively small. For the 100-record retrieval, the size of the REST response was 34 kB, and that of the complete GraphQL response was 38 kB. The GraphQL query limited to identifiers was significantly smaller – only 2.2 kB, approximately 6% of the REST response size. In the 500-record variant, the total size of the GraphQL response was 188 kB, REST – 167 kB, and the limited GraphQL query – only 9.8 kB. This shows that limiting the number of fields in GraphQL requests can reduce response size over tenfold.

It is also worth noting that the size of requests sent by the client is generally larger in GraphQL than in REST, because of GraphQL's structured nature for queries. However, since response size typically dominates total data transfer, well-optimized GraphQL queries can significantly reduce overall network consumption.

Table 2: Volume of data sent and received per request depending on the number of records.

Records	Data	REST	GraphQL	GraphQL (ID)
1	Sent	92 B	574 B	277 B
	Received	571 B	658 B	298 B
100	Sent	103 B	610 B	313 B
	Received	34 kB	38 kB	2,2 kB
500	Sent	103 B	610 B	313 B
	Received	167 kB	188 kB	9.8 kB

3.2. Results of the Second Test

In the second scenario, data was fetched from four related tables: orders, order_details, products and suppliers. In the standard REST approach, retrieving all data required eight separate requests. Alternatively, a REST variant with a single aggregated endpoint was tested, merging data from all four tables into one response. In GraphQL, data was retrieved via a single query in two forms: a full query covering the entire structure and a reduced version limited to approximately 45% of the fields. Figure 8 shows average response times by user load, while throughput values are shown in Figure 9.

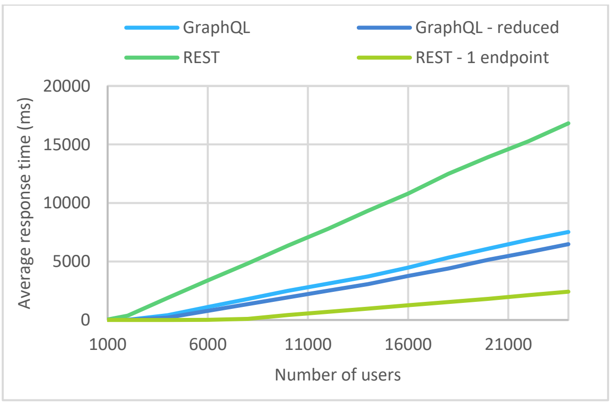

Figure 8: Average response time depending on the number of users – complex query retrieving data from four related tables.

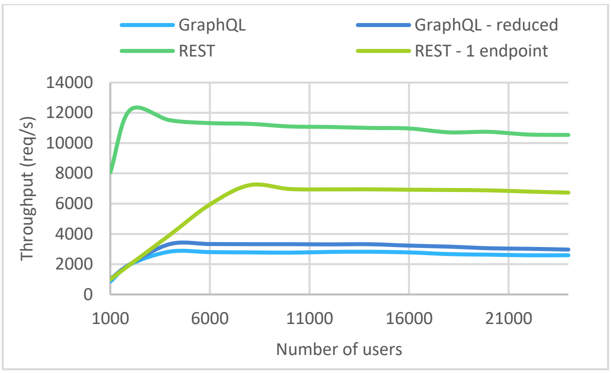

Figure 9: Throughput depending on the number of users – complex query retrieving data from four related tables.

Compared to standard REST, the full GraphQL query demonstrated an average response time reduction of 65%, and the reduced GraphQL query showed a reduction of up to 72%. The best result was achieved by REST with the single aggregated endpoint, which was, on average, 92% faster than standard REST. Under the maximum load of 24,000 users, traditional REST averaged 16,806 ms, while the full GraphQL query averaged 7,520 ms, the limited GraphQL query averaged 6,578 ms, and REST with a single endpoint averaged 2,420 ms.

The results confirm that in scenarios requiring data retrieval from multiple tables, GraphQL offers a significant advantage over classical REST – particularly in terms of response time and data transfer efficiency, especially when the scope of requested fields is properly limited.

The data size for sent and received payloads in the second scenario is presented in Table 4. The largest request was generated by the full GraphQL variant (1.7 kB), and the smallest – REST with a single endpoint (107 B). Responses in REST were the largest (4 kB), while the limited GraphQL variant sent only 1.2 kB. REST with a single endpoint returned 2.5 kB, and the full GraphQL query – 2.4 kB. These findings show that restricting the output of GraphQL responses, along with consolidating data into a single REST endpoint, can greatly minimize the data being transferred.

Table 3: Volume of data sent and received per request.

Data	REST	GraphQL	GraphQL (selected fields)	REST - one end-point
Sent	739 B	1.7 kB	822 B	107 B
Re-ceived	4 kB	2.5 kB	1.2 kB	2.4 kB

4. Discussion

The aim of this study is to compare the performance of REST and GraphQL in relation to query complexity and server load. The experiments were based on two test scenarios representing typical use cases: simple data retrieval from a single table, and complex queries involving multiple related tables. As a result, two research hypotheses were examined.

The first hypothesis assumed that GraphQL provides shorter response times in queries involving complex, nested data structures. The test results fully confirmed this assumption. In the second scenario, which queried data from the orders, order_details, products, and suppliers tables, standard REST required eight separate requests, resulting in an average response time of 16.8 seconds under a load of 24,000 users. In comparison, a full GraphQL query achieved 7.5 seconds, while the limited query (covering about 45% of fields) achieved 6.48 seconds. This represents a reduction of 55% and 61% respectively, compared to REST. REST with a single aggregated endpoint achieved 2.42 seconds, though this approach requires additional server-side logic, which may reduce flexibility. These findings are supported by previous studies. For instance, [7] indicates that GraphQL performs better in retrieving multiple related resources, while [8] demonstrates that a single GraphQL query outperforms REST in multi-relation scenarios. The results of this study align with the conclusions of articles [16, 18], which emphasize GraphQL's strengths in large and deeply nested data structures.

The second hypothesis stated that for simple single reads, REST and GraphQL offer comparable performance. This was only partially confirmed. For 1,000 users, the response time were similar: REST – 3.37 ms, full GraphQL – 4.49 ms. However, as the number of concurrent users increased, the differences became more pronounced. At 24,000 users, REST reached 1,208 ms, full GraphQL – 2,425 ms, and the optimized GraphQL version – 1,910 ms. On average, REST was 50% faster than full GraphQL and 36% faster than the optimized variant. The more records were requested, the more REST outperformed GraphQL. For example, when querying 500 records, REST achieved 19.76 seconds, while GraphQL took 42.32 seconds – a difference of 115%. These delays in GraphQL are caused by the need to execute a separate resolver for each record and the overall complexity of query processing on the server side. These findings are consistent with results from [3, 5, 6, 7, 17], where REST was consistently identified as a more efficient solution for simple operations, especially under high load. For instance, in [5], REST achieved a 51% lower response time, and 37% higher throughput compared to GraphQL for repeated queries to the same data. Similarly, [17] emphasized REST's stability and predictability in high-load environments.

The benefits of reducing the scope of data in GraphQL were also noted by the authors of [12-14] and [15], who reported that tailoring the response to the client's exact needs can reduce payload sizes by several times, a finding confirmed in this study. In the 500-record scenario, the full REST response size was 167 kB, while a GraphQL response limited to a single field was only 9.8 kB, representing just 6% of the REST response.

It is also worth considering the throughput results, i.e., the number of requests processed by the system per second. In simple query scenarios, REST achieved the highest throughput under a 24,000-user load: 10,541 req/s, compared to 6,688 req/s for full GraphQL and 7,897 req/s for the optimized variant. In the complex query scenario, standard REST again reached 10,541 req/s, while full GraphQL and the reduced version achieved 2,591 and 2,965 req/s, respectively. Interestingly, the single-endpoint REST approach, despite offering the fastest response times, had lower throughput (6,726 req/s), likely due to additional server-side computation. These results suggest that REST is more scalable in terms of requests handled per second, especially under high load and when large volumes of data are involved.

To conclude, the study confirms that REST performs better in simple queries and under heavy load, while GraphQL is more efficient in scenarios that require complex, related data retrieval. It is also worth noting that GraphQL, when used with a reduced field set, may outperform REST even in simple cases – particularly in terms of response size and flexibility. The findings are consistent with the latest literature and suggest that the choice of technology should be guided by the application's nature and functional requirements.

5. Conclusions

This study evaluated the performance of REST and GraphQL API approaches in scenarios involving both simple and complex data retrieval. The conducted experiments confirmed that GraphQL is better suited for complex queries involving multiple related tables, while REST remains the more efficient choice for simple operations, particularly under heavy load.

All tests were performed on a single local machine using four PM2-managed instances of the backend application, simulating basic horizontal scalability. However, this setup has limitations, the results may not fully represent production systems deployed in distributed or cloud-native environments. Additionally, production-specific mechanisms such as caching, authentication, and rate limiting were intentionally excluded to focus on raw API performance.

Future works should include testing in production-like environments, such as distributed or cloud-native deployments, to better reflect real-world conditions. In this study, the GraphQL implementation was based on Apollo Server, which is currently the most popular solution in the ecosystem. However, Apollo may not be the most performant under high load, and future work could compare it with alternative GraphQL servers such as Mercury.

References

[1] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD dissertation, University of California, Irvine, 2000.

[2] P. Linjanja, Scalable Application Development with NestJS. Leverage REST, GraphQL, microservices, testing, and deployment for seamless growth, Pact Publishing, 2025.

[3] M. Niswar, R. A. Safruddin, A. Bustamin, I. Aswad, Performance evaluation of microservices communication with REST, GraphQL, and gRPC, Int. J. Electron.

Telecommun. 70(2) (2024) 429–436, https://doi.org/10.24425/ijet.2024.149562.

[4] L. Kamiński, M. Kozłowski, D. Sporysz, K. Wolska, P. Zaniewski, R. Roszczyk, Comparative review of selected Internet communication protocols, Found. Comput. Decis. Sci. 48(1) (2023) 39–56, https://doi.org/10.2478/fcds-2023-0003.

[5] A. Lawi, B. L. Panggabean, T. Yoshida, Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system, Computers 10(11) (2021) 138, https://doi.org/10.3390/computers10110138.

[6] D. A. Hartina, A. Lawi, B. L. E. Panggabean, Performance analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University, In 2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT) IEEE (2018) 237–240, https://doi.org/10.1109/EIConCIT.2018.8878524.

[7] R. Ala-Laurinaho, J. Mattila, J. Autiosalo, J. Hietala, H. Laaki, K. Tammi, Comparison of REST and GraphQL Interfaces for OPC UA, Computers 11(5) (2022) 65, https://doi.org/10.3390/computers11050065.

[8] E. Lee, K. Kwon, J. Yun, Performance Measurement of GraphQL API in Home ESS Data Server, In 2020 International Conference on Information and Communication Technology Convergence (ICTC) IEEE (2020) 1929–1931, https://doi.org/10.1109/ictc49870.2020.9289569.

[9] P. Margański, B. Pańczyk, REST and GraphQL comparative analysis, J. Comput. Sci. Inst. 19 (2021) 89–94, https://doi.org/10.35784/jcsi.2473.

[10] Y. Marchuk, I. Dyyak, I. Makar, Performance Analysis of Database Access: Comparison of Direct Connection, ORM, REST API and GraphQL Approaches, In 2023 IEEE 13th International Conference on Electronics and Information Technologies (ELIT) IEEE (2023) 174–176, https://doi.org/10.1109/ELIT61488.2023.10310748.

[11] N. Vohra, 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) 45–50, https://doi.org/10.1109/ICIMTech55935.2022.9915244.

[12] S. M. Ireland, A. C. R. Martin, GraphQL for the delivery of bioinformatics web APIs and application to ZincBind, Bioinform. Adv. 1(1) (2021) vbab023 https://doi.org/10.1093/bioadv/vbab023.

[13] O. Hartig, J. Pérez, An initial analysis of Facebook's GraphQL language, Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW), CEUR Workshop Proceedings 1912 (2017) 1–10.

[14] M. Vogel, S. Weber, 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, Cham (2018) 283–295, https://doi.org/10.1007/978-3-319-91764-1_23.

[15] O. Bhamare, P. Gite, A. Lohani, K. Choudhary, J. Choudhary, Design and Implementation of Online Legal Forum to Complain and Track UGC Cases using NextJs and GraphQL, In 2023 10th International Conference on Signal Processing and Integrated Networks (SPIN) IEEE (2023) 230–234, https://doi.org/10.1109/SPIN55947.2023.10074892.

[16] G. Brito, T. Mombach, M. T. Valente, Migrating to GraphQL: A practical assessment, In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) IEEE (2019) 140–150, https://doi.org/10.1109/SANER.2019.8667986.

[17] M. Mikuła, M. Dzieńkowski, Comparison of REST and GraphQL web technology performance, J. Comput. Sci. Inst. 16 (2020) 309–316, https://doi.org/10.35784/jcsi.2077.

[18] N. Braunisch, T. Reiplinger, R. Lehmann, Leveraging GraphQL for Large-Scale Queries on Digital Twins in Industry 4.0, In 2024 IEEE International Conference on Industrial Technology (ICIT) IEEE (2024) 1–6, https://doi.org/10.1109/ICIT58233.2024.10541016.

[19] I. Koren, N. Jansen, J. Michael, B. Rumpe, E. Böse, A Low-Code Approach for Data View Extraction from Engineering Models with GraphQL, In 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) IEEE (2023) 888–892, https://doi.org/10.1109/MODELS-C59198.2023.00139.

[20] A. Quiña-Mera, P. Fernández, J. M. García, A. Ruiz-Cortés, GraphQL: A systematic mapping study, ACM Comput. Surv. 55(10) (2023) 1–35, https://doi.org/10.1145/3561818.

[21] S. L. Vadlamani, B. Emdon, J. Arts, 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) 10–17, https://doi.org/10.1109/SERIP52588.2021.00009.

[22] A PostgreSQL port of the Northwind database, https://github.com/pthom/northwind_psql, [11.05.2025].