Data and Metadata. 2025; 4:199
doi: 10.56294/dm2025199

ORIGINAL

Efficiency study of GraphQL and REST Microservices in Docker containers: A computational experiment

Estudio de eficiencia de Microservicios GraphQL y REST en contenedores Docker: Un experimento computacional

Antonio Quiña-Mera[1,2], Zamia Marlene Guitarra de la Cruz[1], Cathy Guevara-Vega[1,2]

[1]Universidad Técnica del Norte, Facultad de Ingeniería en Ciencias Aplicadas. Ibarra, Ecuador.
[2]Universidad Técnica del Norte, Grupo de Investigación Ciencias en Red eCIER. Ibarra, Ecuador.

Cite as: Quiña-Mera A, Guitarra De la Cruz ZM, Guevara-Vega C. Efficiency study of GraphQL and REST Microservices in Docker containers: A computational experiment. Data and Metadata. 2025; 4:199. https://doi.org/10.56294/dm2025199

Submitted: 14-05-2024 **Revised:** 21-09-2024 **Accepted:** 20-02-2025 **Published:** 21-02-2025

Editor: Dr. Adrián Alejandro Vitón Castillo

Corresponding Author: Antonio Quiña-Mera

ABSTRACT

Introduction: in the constant evolution of technology, implementing new services in computer systems is crucial. However, the integration of these services presents problems and certain challenges in the deployment of applications. Technologies such as Docker and microservices architectures are alternatives to alleviate such integration. The aim was to compare the performance efficiency between microservices architectures implemented with GraphQL and REST, deployed in Docker and localhost environments.
Method: a computational experiment was conducted following the Wholin methodology to compare the performance efficiency of microservices architectures. The experimental design consisted of deploying both a GraphQL API and a REST API with identical functionalities in Docker containers and a localhost environment. Both APIs were consumed under controlled complexity and data volume conditions, ensuring a fair evaluation.
Results: the experiment showed that the average response time in the Docker environment was significantly lower compared to the localhost environment. Also, the GraphQL API outperformed the REST API. In addition, a research artifact including all the study materials was published on Zenodo to support the replicability of the experiment.
Conclusions: the architecture deployed in Docker is more efficient for microservices execution, particularly when GraphQL is used.

Keywords: Microservices; Graphql; REST; Computational Experiment; Docker.

RESUMEN

Introducción: en la constante evolución de la tecnología, la implementación de nuevos servicios en los sistemas informáticos es crucial. Sin embargo, la integración de estos servicios presenta problemas y ciertos retos en el despliegue de aplicaciones. Tecnologías como Docker y las arquitecturas de microservicios son alternativas para paliar dicha integración. El objetivo fue comparar la eficiencia de rendimiento entre arquitecturas de microservicios implementadas con GraphQL y REST, desplegadas en entornos Docker y localhost.
Método: se realizó un experimento computacional siguiendo la metodología Wholin para comparar la eficiencia de rendimiento de arquitecturas de microservicios. El diseño experimental consistió en desplegar una API GraphQL y una API REST con idénticas funcionalidades en contenedores Docker y en un entorno localhost. Ambas APIs fueron consumidas bajo condiciones controladas de complejidad y volumen de datos, asegurando una evaluación justa.
Resultados: el experimento mostró que el tiempo medio de respuesta en el entorno Docker fue significativamente inferior en comparación con el entorno localhost. Además, la API GraphQL superó a la API REST.

Además, se publicó en Zenodo un artefacto de investigación que incluía todos los materiales del estudio para apoyar la replicabilidad del experimento.
Conclusiones: la arquitectura desplegada en Docker es más eficiente para la ejecución de microservicios, particularmente cuando se utiliza GraphQL.

Palabras clave: Microservicios; GraphQL; REST; Experimento Computacional; Docker.

INTRODUCTION

Software development companies present several challenges, among which the integration of services and holistic management of their architectures make scalability and portability key factors of change.[1] However, code management and collaboration activities in the software development process are complex without neglecting the accelerated change of technology, the use of artificial intelligence, and the provision of cloud services that are constantly evolving.[2] During software development, when making any modification in the code or, in turn, in the life cycle phases, a compilation and deployment of the application is required, which produces a long and unfavorable iteration cycle, at least when the methodologies are not well carried out.[1] In the software development environment, there are problems such as low execution efficiency, unbalanced distribution of resources, and software reuse failures caused by different development environments, without leaving aside problems such as the high level of consumption of time and resources in the implementation and deployment of applications that cause excessive expenses, failures in the operation of services, and low performance of applications.[2]

To mitigate some of the above problems, the use of microservices solves certain drawbacks, where a single application is developed with a set of small services that can be compiled, deployed, and operated independently according to the needs of each organization. In the same sense, using containers facilitates the development, deployment, and execution of applications using the same environment.[3] Containers allow the packaging of an application together with all its dependencies and configurations, ensuring that it works consistently in any environment.[3] In this sense, we understand that researching the efficiency of Docker, GraphQL and REST to discover what conditions are most efficient for implementing microservices architectures, solving compatibility, scalability and performance issues. For the reasons stated above, the following research question is defined:

RQ1: how does the choice of deployment environment (Docker or Localhost) and the type of microservices architecture (GraphQL or REST) impact the software product's performance efficiency?

Our proposed solution to the research question is a computational experiment. This experiment will compare the performance efficiency of GraphQL and REST microservices architectures using APIs deployed in a Docker container (virtualized environment) and a localhost environment. The 'response time' metric, based on the ISO/IEC 25023 standard,[4] will be used to measure performance efficiency. This research is not just an academic exercise. It contributes to the fulfillment of Objective 9: "Industry, innovation and infrastructure. Technological advances are also essential for finding permanent solutions to economic challenges..." of the Sustainable Development Goals developed by the UN and UNESCO.[5] It's a step towards finding permanent solutions to economic challenges through technological innovation.

The rest of the document presents the following sections: Background, explaining software architectures, containers, experimentation in software engineering, and software product quality. Method, where the phases of the experimental process are explained, especially the design. Development, where it explains the operationalization and execution of the experiment. Results, where the main findings of the computational experiment are detailed. Threats to validity, it explains the types of threats found in the experiment. Discussion, the results of RQ1 are discussed. Conclusions and future work specify the future lines of research.

Background

Software Architecture

Santos et al.[6] determine that by not carefully analyzing the quality of th e architecture before the construction of a system, problems can occur, and the later these are detected, the greater the impact. It is considered that 50 % of the delayed projects and 35 % of those exceeding the production cost need help with the software architecture. The main purpose of the architecture is to support the software life cycle in order to facilitate development, implementation, and maintenance.[7] Among the main benefits are: i) increases software quality, ii) improves project delivery times, iii) reduces development costs.

Microservice Oriented Architecture

Yaryina et al.[8] see microservices as an approach to developing a single application with small services, each running in its process and communicating with lightweight mechanisms, often an API (Application Programming

Interface) of HTTP resources. One of the characteristics of these services is that they can be written in different programming languages and use different data storage technologies. Today, it is a well-established approach to modularity and agility, as each microservice becomes an independent development unit.[9] Microservices can be tested and deployed individually because each has its own lifecycle; they can also be combined with each other because of the advantage of language and technology independence.[10] Among the benefits are: i) service management, ii) independence, iii) modularity, iv) data management, v) fault tolerance.

REST architecture

The REST (Representational State Transfer) architecture is based on Uniform Resource Identifiers (URI), which provide a unique resource identifier, and on the Hypertext Transfer Protocol (HTTP), which defines the type of operation to be performed on the resource.[11] It is an architectural style based on the client-server paradigm that enables scalability, availability, and performance to be defined in distributed systems. REST-based APIs are exposed through endpoints, and each endpoint returns the information defined according to the operation.[12]

GraphQL architecture

GraphQL is a query language for implementing web services. It was developed internally at Facebook as an alternative to REST-based applications.[13] In GraphQL, the data required by the services are defined dynamically, while with REST, the server returns a JSON document with all the fields, even if the client only requires one.[12] Some terms used by GraphQL are data types, queries, and mutations.[14]

Docker

Docker is an open-source platform that follows a client-server architecture and aims to automate distributed application deployment based on containers. A container is in charge of packaging everything indispensable for the system to work; therefore, there is no need to worry about software versioning or computer resources since they are inside a container.[15] Docker is a lightweight virtualization solution that improves the execution of a container and decreases memory consumption by sharing the same kernel between the host and a virtual container.[16]

Experimentation in Software Engineering

Experimentation in software engineering makes it possible to determine the cause of certain results. Experimentation is not very simple because a design must be established that solves the statistical power that generalizes the results through hypothesis testing methods.[17] One of the advantages of experimentation is the control of subjects, objects, and instrumentation.[18] Controlled and computational experiments,[19] require following an experimental process to determine the scope, planning, operationalization, analysis, interpretation, presentation of results, and packaging of research artifacts.[20]

Characteristics and measures of software quality

The ISO/IEC 25000 family of standards provides quality models for computer systems and software products. From the characteristics and sub-characteristics defined in this family, it is possible to measure, validate, and evaluate software product quality.[21] In this study, we will employ the "Performance Efficiency" characteristic stated in the software product quality model of ISO/IEC 25010.[22] We will use the "Average Response Time" metric to measure this efficiency, according to ISO/IEC 25023.[4]

METHOD

Experimental Setting

Goal Definition

We used the (GQM) approach,[23] to define the objective of the experiment, considering the following:

- Analyze microservices architectures in the software deployment process.
- In order to compare the performance efficiency, validating with the response time metric.
- Concerning software quality.
- From a researcher's point of view.
- In the context of a computational laboratory.

Factors and Treatments

The research factor in the experiment is software architecture, focusing specifically on the deployment of APIs as microservices; the treatments were:

- Deployment of microservices in a virtualized environment using Docker.
- Deployment of microservices in a localhost environment.

https://doi.org/10.56294/dm2025199

In each environment a GraphQL API and a REST API were deployed.

Variables

"Microservices-oriented software architecture" was defined as an independent variable, represented by the GraphQL and REST APIs, deployed in the Docker and localhost environments. On the other hand, "Software product quality" was established as a dependent variable, measured in terms of the characteristic "Performance efficiency".

Next, it describes how the metric "Average response time", referred to in the characteristic "Performance Efficiency" of the ISO/IEC 25010 standard,[22] is operationalized.

Average response time: It refers to the time a request takes to complete, i.e., the average time it takes to complete an asynchronous process.[4] The measurement function when applying was:

$$X = \sum_{i=1}^{n} (B_i - A_i) \; / \; n$$

Where:
Ai = time to start job i.
Bi = time to complete job i.
n = number of measurements.

Hypothesis

The following hypotheses were defined and derived from research question RQ_1

- H_0: (Null hypothesis): there is no significant difference in software product quality when deploying GraphQL and/or REST microservices in a virtualized Docker environment or a localhost environment. The software quality of microservices is similar in both environments, regardless of the type of API used (GraphQL or REST).
- H_1: (Alternative Hypothesis 1): there is a significant difference in software product quality when deploying GraphQL and/or REST microservices in a virtualized environment using Docker. It is posited that the software quality of GraphQL and/or REST microservices deployed in Docker is superior to that obtained when deployed in a localhost environment.
- H_2: (Alternative Hypothesis 2): there is a significant difference in software product quality when deploying GraphQL and/or REST microservices in a localhost environment. It is posited that the software quality of GraphQL and/or REST microservices deployed on localhost is superior to that obtained when deployed in a virtualized Docker environment.

Design

A computational laboratory was designed with the necessary conditions to compare the performance efficiency of microservices implemented with GraphQL and REST APIs deployed in Docker and localhost environments. Four use cases (CU) were defined to simulate controlled workloads, with equivalence in the number of records and data manipulation complexity for each API. Table 1 shows the experiment design.

Table 1. Experiment design

Use Case	REST (Docker - localhost) number of records	GraphQL (Docker - localhost) number of records
CU1	1,100,1000,10000	1,100,1000,10000
CU2	1,100,1000,10000,100000	1,100,1000,10000,100000
CU3	1,100,1000,10000,100000	1,100,1000,10000,100000
CU4	1,100,1000,10000,100000	1,100,1000,10000,100000

CU1 focuses on data insertion, while CU2, CU3, and CU4 correspond to data queries with increasing complexity, ranging from the relationship with one table to the interaction with three tables in a relational database. This approach allows for the evaluation of the behavior of architectures and deployment environments under different workload levels. Each CU will be run five times for each number of records; these repetitions guarantee the reduction of variability and biases, providing more accurate and reliable results.

https://doi.org/10.56294/dm2025199

Experimental Tasks

The experiment contemplates the implementation of two functionally identical APIs for GraphQL and REST, where 16 Experimental Tasks (ET) that will execute the four use cases in Docker and localhost environments were designed, as detailed in table 2.

Table 2. Distribution of experimental tasks

Use Case	GraphQL API Docker	GraphQL API localhost	REST API Docker	REST API localhost
CU1	ET01	ET02	ET03	ET04
CU2	ET05	ET06	ET07	ET08
CU3	ET09	ET10	ET11	ET12
CU4	ET13	ET14	ET15	ET16

The experimental lab architecture was designed so the same computer could host both the localhost and virtualized Docker environments. First, the necessary tools (PostgreSQL, NGINX) were installed and configured to deploy the GraphQL and REST APIs in the localhost environment. Subsequently, in this environment, Docker was configured in a manner equivalent to localhost. Finally, a client application was deployed on localhost to consume the GraphQL and REST APIs deployed in both environments. Figure 1 shows the detailed architecture of the experimental lab.

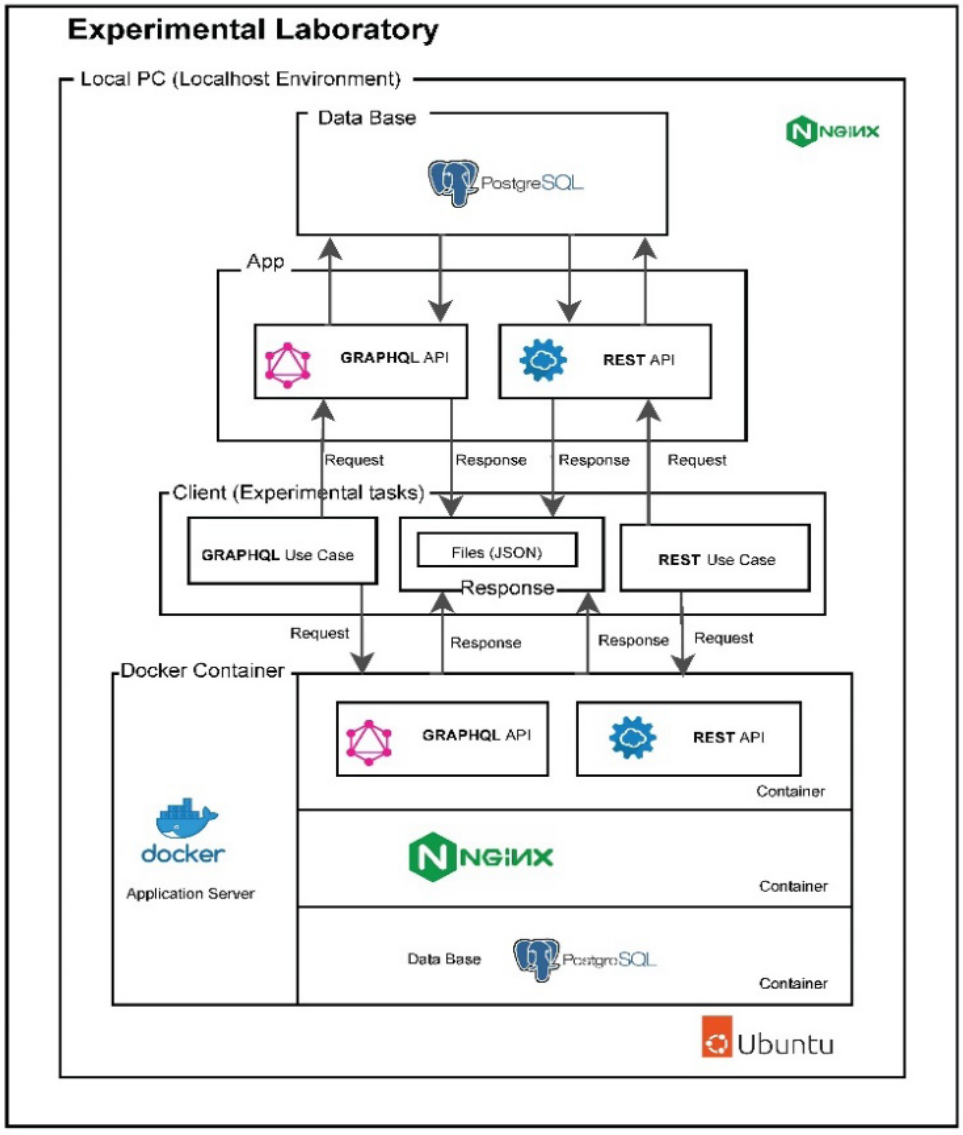


Figure 1. Experimental laboratory architecture

https://doi.org/10.56294/dm2025199

DEVELOPMENT

Instrumentation

This section specifies the infrastructure, technologies, and libraries used in the computational laboratory:

Computer description

- Operating System: Linux Ubuntu 22.04 64-bit.
- Processor: AMD Ryzen 7 4700U with Radeon Graphics 2.00 GHz.
- Memory: RAM 16.0 GB.

Development Environment

- IDE integrated development environment: Visual Studio Code v1.67.2.
- Programming Language: Typescript v4.7.4.
- Javascript runtime environment: NodeJS v16.17.
- npm library for the GRAPHQL API: @nestjs/graphql v10.1.1, @nestjs/apollo v10.1.0, @nestjs/core v9.0.0, @nestjs/server-express v3.10.2.
- Database Structure Mapping: @nestjs/typeorm v9.0.1, typeorm v0.3.9.
- Database driver (PostgreSQL): pg v8.8.0.
- Client application for REST API consumption: Postman v9.18.3.
- Client application for GRAPHQL API consumption: GraphQL Playground.

Data collection and analysis

- Microsoft Excel 365.
- IBM SPSS v27.0.

Table 3 shows the structure of the experiment's data collection.

Table 3. Data collection structure

Variable	Description
Sample	Sample number (auto numeric)
Architecture	GraphQL or REST
Environment	Docker or localhost
Use Case	Executed use case
Repetition	Number of the repetition (between 1 to 5)
No. Records	Number of records queried or inserted
Time	Response time of the use case execution, captured in milliseconds

Experiment Execution

This experiment was carried out in September 2022.

- Preparation of the experimental laboratory. To ensure the functionality of the experimental lab, a functional test was executed for each use case in the APIs (GraphQL and REST) of the environments (Docker and localhost), according to the distribution detailed in table 3.
- Execution of the experiment. After testing the functionality of the experimental tasks, the experiment was run iteratively for each architecture in each environment. Each client application run executed five iterations of the use case for each number of records in the GraphQL and REST architectures. For CU1 (data insertion), a distribution of 1, 100, 1000, and 10,000 records was used, while in CU2-CU4 (data queries), a distribution of 1, 100, 1000, 10,000, and 100,000 records was used, see table 1. In total, 380 samples were obtained in each run (iteration). Figure 2 illustrates the detailed process for each iteration.
- Data Collection. At the end of the four iterations of the client application, according to the experimental design (table 1), the distribution of experimental tasks (table 3), and the execution process (figure 2), a total of 1520 samples were collected. These data were stored in a Microsoft Excel 365 file for analysis and tabulation. Examples of response times (in milliseconds) captured from the Visual Studio Code console because of running the client application are presented in figure 3. The data were copied and pasted into the Microsoft Excel 365 file in table 4.

https://doi.org/10.56294/dm2025199

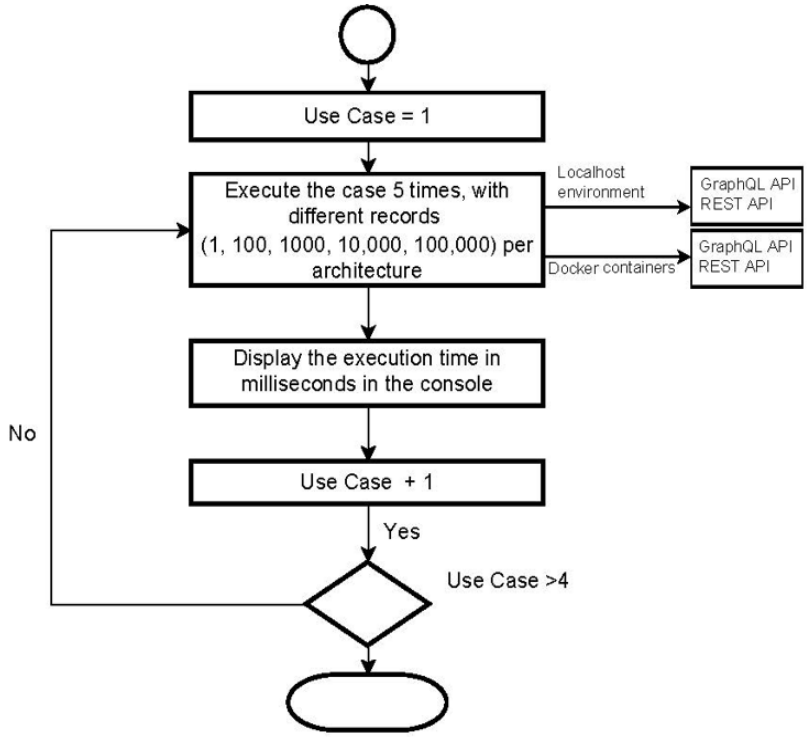


Figure 2. Experimental process in the execution of the client application

```
microservicesf | GRAPHQL Runtime CU-02: 1.335s
nginx          | 192.168.1.103 - - [02/Oct/2022:18:17:19 +0000] "POST /graphql HTTP/1.1" 200 15843095 "http://192.168.1.103/graphql"
microservicesf | GRAPHQL Runtime CU-02: 1.208s
nginx          | 192.168.1.103 - - [02/Oct/2022:18:17:27 +0000] "POST /graphql HTTP/1.1" 200 15843095 "http://192.168.1.103/graphql"
microservicesf | GRAPHQL Runtime CU-02: 1.346s
nginx          | 192.168.1.103 - - [02/Oct/2022:18:17:36 +0000] "POST /graphql HTTP/1.1" 200 15843095 "http://192.168.1.103/graphql"
microservicesf | GRAPHQL Runtime CU-02: 1.212s
nginx          | 192.168.1.103 - - [02/Oct/2022:18:17:45 +0000] "POST /graphql HTTP/1.1" 200 15843095 "http://192.168.1.103/graphql"
microservicesf | GRAPHQL Runtime CU-02: 1.161s
nginx          | 192.168.1.103 - - [02/Oct/2022:18:17:56 +0000] "POST /graphql HTTP/1.1" 200 15843095 "http://192.168.1.103/graphql"
```

Figure 3. Results of the GraphQL client application run on Docker

Table 4. Collection of GraphQL execution results on Docker

Sample	Architecture	Environment	Use Case	Repetition	No. Records	Time (ms)
326	GraphQL	Docker	CU2	1	100000	1335
327	GraphQL	Docker	CU2	2	100000	1208
328	GraphQL	Docker	CU2	3	100000	1346
329	GraphQL	Docker	CU2	4	100000	1212
330	GraphQL	Docker	CU2	5	100000	1161

RESULTS

This section presents the analysis of the results obtained from executing the computational experiment proposed in this study. It specifically analyzes the "average response time" metric of GraphQL and REST architectures deployed in Docker and localhost environments and its impact on "performance efficiency," considered a key feature of software quality. A percentage analysis of the performance efficiency for each use case is then provided, comparing the average response time of both architectures in the different deployment environments.

Table 5 presents the percentage analysis for use case 1. The results clearly demonstrate that the GraphQL architecture improved by 29,64 % in performance efficiency in the Docker environment compared to running it on localhost. Similarly, REST achieved a 38,79 % improvement on Docker versus localhost. These compelling results allow us to confidently determine that, in the data insertion use case, Docker is significantly more efficient than the localhost environment.

https://doi.org/10.56294/dm2025199

Table 5. Percentage of performance efficiency - CU1

Environment/ Architecture	Average response time	Environment/ Architecture	Average response time	
Docker/ GraphQL	224,33	Docker/REST	222,75	0,70 % REST more efficient than GraphQL in Docker
Localhost/ GraphQL	318,83	Localhost/REST	363,91	12,39 % GraphQL more efficient than REST on localhost
	29,64 % Docker/ GraphQL efficiency		38,79 % Docker/ REST efficiency	

Table 6 presents the percentage analysis of use case 2, where it is observed that the GraphQL architecture is 38,79 % more efficient in the Docker environment compared to its execution on localhost. On the other hand, the REST architecture shows a 43,33 % higher efficiency in Docker than in localhost. These results, once again, allow us to confidently determine that, in the case of simple queries to a database table, the virtualized Docker environment is significantly more efficient than the localhost environment.

Table 6. Percentage of performance efficiency – CU2

Environment/ Architecture	Average response time	Environment/ Architecture	Average response time	
Docker/ GraphQL	289,60	Docker/REST	281,54	2,79 % GraphQL more efficient than REST in Docker
Localhost/ GraphQL	481,02	Localhost/REST	496,79	3,17 % GraphQL more efficient than REST on localhost
	39,79 % Docker/GraphQL efficiency		43,33 % Docker/REST efficiency	

Table 7 presents the percentage analysis of use case 3, where it is observed that GraphQL architecture is 20,44 % more efficient in the Docker environment compared to localhost. On the other hand, REST shows a 20,73 % higher efficiency in Docker compared to localhost. These results allow us to determine that, in the case of queries involving two tables of a database, the virtualized Docker environment is more efficient than the localhost environment.

Table 7. Percentage of performance efficiency – CU3

Environment/ Architecture	Average response time	Environment/ Architecture	Average response time	
Docker/ GraphQL	682,20	Docker/REST	711,95	4,18 % GraphQL more efficient than REST in Docker
Localhost/ GraphQL	857,44	Localhost/REST	898,11	4,53 % GraphQL more efficient than REST on localhost
	20,44 % Docker/GraphQL efficiency		20,73 % Docker/REST efficiency	

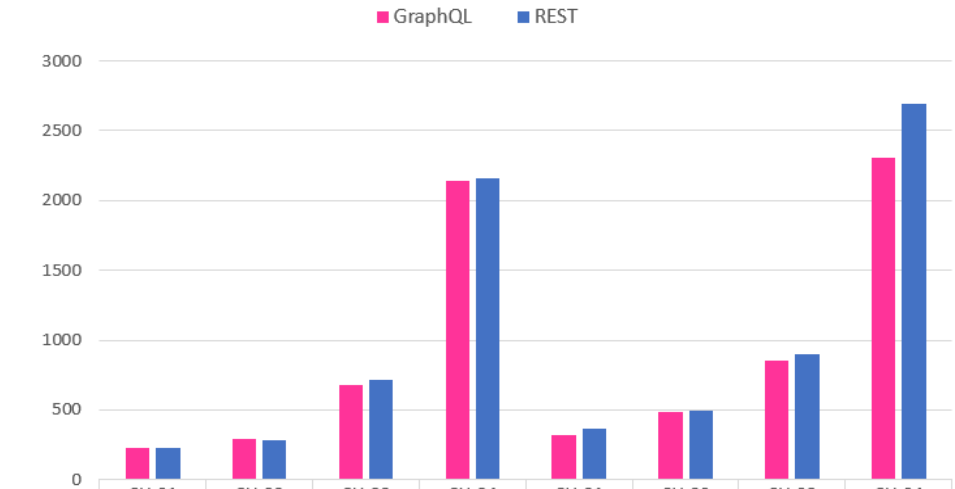
Table 8. Percentage of performance efficiency – CU4

Environment/ Architecture	Average response time	Environment/ Architecture	Average response time	
Docker/ GraphQL	2140,51	Docker/ REST	2158,57	0,84 % GraphQL more efficient than REST in Docker
Localhost/ GraphQL	2302,71	Localhost/ REST	2697,58	14,64 % GraphQL more efficient than REST on localhost
	7,04 % Docker/GraphQL efficiency		19,98 % Docker/REST efficiency	

https://doi.org/10.56294/dm2025199

Table 8 shows the percentage analysis of use case 4, where it is observed that the GraphQL architecture is 7,04 % more efficient in the Docker environment compared to localhost. On the other hand, the REST architecture shows a 19,98 % improvement in efficiency in Docker versus localhost. These results allow us to determine that, in the case of queries involving three tables of a database, the virtualized Docker environment is more efficient than the localhost environment.

Figure 4 summarizes the average response time of GraphQL and REST APIs, grouped by Docker and localhost environments.



	Docker				Localhost			
	CU-01	CU-02	CU-03	CU-04	CU-01	CU-02	CU-03	CU-04
GraphQL	224,339	289,6036	682,208	2140,516	318,8305	481,0276	857,4404	2302,716
REST	222,753	281,5472	711,9596	2158,576	363,9136842	496,798	898,1112	2697,584

Figure 4. Average performance efficiency value between GraphQL and REST

Likewise, table 9 summarizes the performance efficiency percentages obtained in the execution of all use cases for the GraphQL and REST APIs, highlighting that the Docker environment was more efficient compared to the localhost environment.

Table 9. Percentage of Docker performance efficiency compared to localhost

Use case	Docker/GraphQL	Docker/REST
CU1	29,64 %	38,79 %
CU2	39,79 %	43,33 %
CU3	20,44 %	20,73 %
CU4	7,04 %	19,98 %
Average	24,23 %	30,71 %

Finally, table 10 summarizes the performance efficiency percentages of GraphQL compared to REST, obtained from running all use cases in the Docker and localhost environments. The results show that GraphQL is more efficient than REST in most cases.

Table 10. Percentage of GraphQL performance efficiency compared with REST

Use case	Efficiency GraphQL/REST on Docker	Efficiency GraphQL/REST on localhost
CU1	-0,70 %	12,39 %
CU2	-2,79 %	3,17 %
CU3	4,18 %	4,53 %
CU4	0,84 %	14,64 %
Average	0,38 %	8,68 %

https://doi.org/10.56294/dm2025199

Statistical analysis of the result

The overall statistical results show that the average efficiency value for GraphQL in the localhost environment was 1025,328 (mean) with a standard deviation of 2319,840 and a 95 % confidence interval [552,753-1497,904]. On the other hand, REST had a mean value of 1149,820, with a standard deviation of 2709,509 and a 95 % confidence interval [597,865-1701,775], these values being higher than those of GraphQL. Despite this, the mean response time of GraphQL was determined to be more efficient than REST in the localhost environment.

Similarly, the statistical analysis for the Docker architecture shows that the mean efficiency value for GraphQL was 866,262 with a standard deviation of 2108,299 and a 95 % confidence interval [436,780-1295,745]. On the other hand, REST presented a mean value of 876,39 with a standard deviation of 2132,421 and a 95 % confidence interval [441,994-1310,787]. These results determined that GraphQL response time is more efficient than REST in the Docker architecture.

Threats to validity

A series of situations were defined, factors that reinforce the weaknesses or limitations that could threaten the validity of the results of the experiment:

- Internal Validity: the experimental lab was developed by building REST APIs and GraphQL for real data from a banking module, using the SCRUM agile development framework. Iterative-incremental monitoring was performed, and technical support was provided for the requirements raised in the tasks and the design of the experiment. In addition, acceptance tests were performed for each implemented use case, which contributed to ensuring internal validity.
- External Validity: in the context of the experimental laboratory, an application of a financial system belonging to a private entity was used, which allowed obtaining a close approximation to the data used in the industrial practice of software engineering. The experimental tasks included operations of simple-medium complexity, covering common data query and insertion scenarios. While generalization of the results to more complex tasks is limited, the study's focus on basic data manipulation operations opens exciting possibilities for future research into behavior in more complex systems.
- Construct Validity: specific constructs were defined for the execution phase of the experiment to automate the measurement of performance efficiency in each use case and to evaluate the impact of the architectures in the deployment of GraphQL and REST services on software quality. These constructs were not just developed but developed in consensus among the experimenters, ensuring that everyone's expertise was considered and validated through acceptance testing.
- Validity of Conclusion: to mitigate possible threats to the conclusions, the study was supported with a statistical analysis of the results obtained in the experiment, ensuring that the findings are based on reliable data.

DISCUSSION

The adoption of microservices architectures and deployment in virtualized environments such as Docker has transformed how services are developed, deployed, and scaled in today's computing systems.[24] The reduction in response time observed in the Docker architecture versus the localhost environment can be attributed to several reasons. First, encapsulating services within containers provides isolation that eliminates many of the conflicts typical of traditional environments, such as configuration issues, incompatible library versions, or discrepancies between development and production environments. It is important to note that, according to ISO/IEC 25010 standards,[22] performance efficiency has become a critical factor in evaluating software systems, especially in distributed applications.[25]

In this study, it was possible to observe the reduction in response time in the Docker architecture versus the localhost environment; this can be attributed to several reasons. First, the encapsulation of services within containers provides isolation that eliminates many of the conflicts typical of traditional environments, such as configuration problems, incompatible versions of libraries, or discrepancies between development and production environments.

Docker's advantage in microservices management also lies in its ability to scale horizontally more efficiently, allowing new containers to be created in a matter of seconds, which can be vital for applications that require high availability and fault tolerance. This scalability contrasts with the limitations of the localhost environment, where service replication can involve much slower and more demanding processes in terms of system resources. The experiment results underscore the efficiency of GraphQL in the Docker architecture. This aligns with previous research, highlighting its ability to reduce data overhead and optimize specific queries, in contrast to the broader and less controlled approach of REST. This efficiency in inter-service communication is particularly valuable in scenarios where there is a need to optimize latency and minimize bandwidth usage. Despite the clear benefits, it is crucial to keep in mind the challenges inherent in implementing containers in microservices architectures. The complexity of orchestrating and monitoring multiple containers can lead to operational overhead if the right tools, such as Kubernetes, are not implemented.[26] The above-discussed results support

the adoption of Docker architectures, especially in scenarios where response time efficiency and scalability are priorities. Furthermore, the use of GraphQL combined with Docker can support the development of applications that are adaptable to current performance demands.

CONCLUSIONS

The present study has shown that the virtualized Docker environment offers higher efficiency in deploying API services than the localhost environment. GraphQL and REST obtained better response times and higher overall efficiency in Docker, highlighting REST with an efficiency of 30,71 % and GraphQL with 24,23 %. Complementary, it was observed that the efficiency of GraphQL versus REST was remarkable in most cases, especially in localhost, where GraphQL showed an average efficiency of 8,68 %, compared to a modest 0,38 % in Docker. However, REST showed an advantage in some specific cases such as insertion and simple data queries in the Docker environment, demonstrating its potential for use when APIs that perform simple data manipulation actions are needed. The results also indicate that the average response time increases with the complexity of the queries, especially in those cases involving relationships between several tables. In summary, the virtualized Docker environment has proven to be more efficient for deploying API services, and its combination with the GraphQL architecture offers a promising approach for applications that require data handling efficiency and reduced response time. Therefore, according to the results obtained, H_1: (Alternative Hypothesis 1) is accepted: There is a significant difference in software product quality when deploying GraphQL and/or REST microservices in a virtualized environment using Docker. It is proposed that the software quality of GraphQL and/or REST microservices deployed in Docker is superior to that obtained when deployed in a localhost environment.

Future Work

Based on the results of this study, several future works can be considered, such as:

- Testing in High Complexity Scenarios: while this study focused on simple-medium complexity use cases, it would be valuable to expand the research to higher complexity tasks, such as queries involving multiple relationships between tables or more complex integrations with other services.
- Security Assessment and Fault Tolerance: fFuture studies could focus on assessing security and fault tolerance in Docker and GraphQL-based architectures to identify potential vulnerabilities and propose solutions to improve system robustness.
- Cache Optimization in GraphQL: since REST showed an advantage in some cases due to caching, future research could explore implementing caching mechanisms in GraphQL to improve its performance further and reduce response times.
- Exploration of Other Orchestration Platforms: although Docker has proven efficient, research could be extended by evaluating other container orchestration platforms, such as Kubernetes, to get a broader view of the impact on efficiency and scalability.

Research artefact

The set of items included in the research artifact to support the reproducibility of the present experiment is available in the Zenodo repository.[27]

BIBLIOGRAPHIC REFERENCES

1. Gouigoux J-P, Tamzalit D. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. 2017 IEEE Int. Conf. Softw. Archit. Work., 2017, p. 62–5. https://doi.org/10.1109/ICSAW.2017.35.

2. Qian L, Chen H, Yu J, Zhu G, Zhu J, Ren C, et al. Research on Micro Service Architecture of Power Information System Based on Docker Container. IOP Conf Ser Earth Environ Sci 2020;440:32147. https://doi.org/10.1088/1755-1315/440/3/032147.

3. Shifeng Z, Shanliang P. Application of Docker technology in micro-service [J]. Electron Technol Softw Eng 2019;4:164.

4. International Organization for Standardization. ISO/IEC 25023:2016(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality. 2016.

5. United Nations. 17 Goals to Transform Our World - Sustainable Development Goals 2024. https://www.un.org/sustainabledevelopment/.

6. Santos DS, Oliveira BRN, Kazman R, Nakagawa EY. Evaluation of Systems-of-Systems Software Architectures: State of the Art and Future Perspectives. ACM Comput Surv 2022;55. https://doi.org/10.1145/3519020.

7. Sobhy D, Bahsoon R, Minku L, Kazman R. Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review. ACM Trans Softw Eng Methodol 2021;30. https://doi.org/10.1145/3464305.

8. Yarygina T, Bagge AH. Overcoming Security Challenges in Microservice Architectures. 2018 IEEE Symp. Serv. Syst. Eng., 2018, p. 11–20. https://doi.org/10.1109/SOSE.2018.00011.

9. Zhang D, Si X, Qian B, Tan F, He P. Design and Research of Adaptive Filter Microservices Based on Cloud-Native Architecture. 2024 5th Int. Conf. Comput. Eng. Appl., 2024, p. 521–5. https://doi.org/10.1109/ICCEA62105.2024.10603929.

10. Ayas HM, Hebig R, Leitner P. The Roles, Responsibilities, and Skills of Engineers in the Era of Microservices-Based Architectures. 2024 IEEE/ACM 17th Int. Conf. Coop. Hum. Asp. Softw. Eng., 2024, p. 13–23.

11. Neumann A, Laranjeiro N, Bernardino J. An Analysis of Public REST Web Service APIs. IEEE Trans Serv Comput 2021;14:957–70. https://doi.org/10.1109/TSC.2018.2847344.

12. Brito G, Valente MT. REST vs GraphQL: A Controlled Experiment. 2020 IEEE Int. Conf. Softw. Archit., 2020, p. 81–91. https://doi.org/10.1109/ICSA47634.2020.00016.

13. Quiña-Mera A, Fernandez P, García J, Ruiz-Cortés A. GraphQL: A Systematic Mapping Study. ACM Comput Surv 2023;55:1–35.

14. Diyasa GSM, Budiwitjaksono GS, Ma'rufi HA, Sampurno IAW. Comparative Analysis of Rest and GraphQL Technology on Nodejs-Based Api Development. Nusant Sci Technol Proc 2021:43–52. https://doi.org/10.11594/nstp.2021.0908.

15. Davis A. Bootstrapping Microservices with Docker, Kubernetes, and Terraform: A project-based guide. 2021.

16. Zhao N, Lin M, Albahar H, Paul AK, Huan Z, Abraham S, et al. An End-to-end High-performance Deduplication Scheme for Docker Registries and Docker Container Storage Systems. ACM Trans Storage 2024;20. https://doi.org/10.1145/3643819.

17. Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A. Experimentation in Software Engineering. Springer Publishing Company, Incorporated; 2012.

18. Guevara-Vega C, Bernárdez B, Durán A, Quiña-Mera J, Cruz M, Ruiz-Cortés A. Empirical Strategies in Software Engineering Research: A Literature Survey. II Int. Conf. Inf. Syst. Softw. Technol. (ICI2ST 2021), Quito, Ecuador: IEEE Press; 2021.

19. Landeta-López P, Guevara-Vega C. Computational Experiments in Computer Science Research: A Literature Survey. IEEE Access 2024:1. https://doi.org/10.1109/ACCESS.2024.3458808.

20. Guevara-Vega C, Bernárdez B, Cruz M, Durán A, Ruiz-Cortés A, Solari M. Research artifacts for human-oriented experiments in software engineering: An ACM badges-driven structure proposal. J Syst Softw 2024;218:112187. https://doi.org/https://doi.org/10.1016/j.jss.2024.112187.

21. ISO - International Organization for Standardization. The ISO/IEC 25000 series of standards 2018.

22. International Organization for Standardization. ISO/IEC 25010:2023(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. 2023.

23. Basili VR. Software modeling and measurement: the Goal/Question/Metric paradigm. University of Maryland at College Park; 1992.

24. Merkel D. Docker: lightweight Linux containers for consistent development and deployment. Linux J 2014;2014.

25. Quiña-Mera A, García JM, Fernández P, Vega-Molina P, Ruiz-Cortés A. GraphQL or REST for Mobile Applications? Vol 1675 CCIS, Pages 16 - 30 2022;1675 CCIS:16–30. https://doi.org/10.1007/978-3-031-20319-0_2.

26. Carrión C. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. ACM Comput Surv 2022;55:1–37.

27. Quiña Mera JA, Guevara Vega C, Guitarra Z. Research Artefact of study: "Efficiency study of GraphQL and REST Microservices in Docker containers: A computational experiment". Zenodo; 2024. https://doi.org/10.5281/zenodo.13840814.

FINANCING

This work has been supported by the Universidad Técnica del Norte–Ecuador.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

AUTHORSHIP CONTRIBUTION

Conceptualization: Antonio Quiña-Mera, Cathy Guevara-Vega.
Data curation: Zamia Guitarra, Antonio Quiña-Mera.
Formal analysis: Zamia Guitarra, Antonio Quiña-Mera.
Acquisition of funds: Antonio Quiña-Mera.
Research: Antonio Quiña-Mera, Zamia Guitarra.
Methodology: Cathy Guevara-Vega, Zamia Guitarra, Antonio Quiña-Mera.
Project management: Antonio Quiña-Mera.
Resources: Zamia Guitarra.
Software: Zamia Guitarra.
Supervision: Antonio Quiña-Mera.
Validation: Zamia Guitarra, Antonio Quiña-Mera.
Display: Zamia Guitarra.
Drafting - original draft: Cathy Guevara-Vega.
Writing - proofreading and editing: Antonio Quiña-Mera, Cathy Guevara-Vega, Zamia Guitarra.