www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(5), 2026, 195-211. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2026.v17i5.1412

Performance Analysis of Web Microservices with Technological Convergence between REST and GraphQL API Architectures

Carlos Arturo Aguilar-Díaz[1], Nelson Rangel-Valdez[2], María Lucila Morales-Rodríguez[3], Guadalupe Castilla-Valdez[4], Claudia Guadalupe Gómez-Santillán[5], Ana Lidia Martínez-Salazar[6], Pedro Martín García-Vite[7]

[1,2,3,4,5,6,7] Tecnológico Nacional de México – Instituto Tecnológico de Ciudad Madero

carlos.ad@cdmadero.tecnm.mx[1], nelson.rv@cdmadero.tecnm.mx[2], lucila.mr@cdmadero.tecnm.mx[3], guadalupe.cv@cdmadero.tecnm.mx[4], claudia.gs@cdmadero.tecnm.mx[5], ana.ms@cdmadero.tecnm.mx[6], pedro.gv@cdmadero.tecnm.mx[7]

Abstract. Efficient data exchange between heterogeneous systems is a critical need in modern software development. REST API and GraphQL API are the dominant paradigms for web microservices implementation; however, the scientific community has not reached a consensus on which offers better performance. Both technologies offer complementary advantages that deserve to be studied, while REST has better response time and throughput, GraphQL improves on resource consumption. This paper presents a controlled and reproducible experiment that evaluates server response time when combining both technologies in different proportions within the development of a system. A set of 32 web microservices were designed by combining REST and GraphQL technologies on a basic generic user management case study, each composed by five CRUD operations and operating with server request volumes ranging from 1 to 10, accumulating up to 76,800 requests. Findings show that a web microservice design involving the management of 4 CRUD operations by REST and 1 by GraphQL achieves comparable or superior median response times in contrast with a design that implements all operations using REST. Particularly, when one of the read or create CRUD operations is implemented using GraphQL and the remaining with REST the response time is minimized. In contrast, implementing either the update or delete CRUD operations with GraphQL results in less favorable response time. These results provide empirical evidence of the requirements that a web microservice design must meet to strategically combine REST and GraphQL technologies without impacting server response time, contradicting the common sense that dictates using GraphQL considerably affects response times.

Keywords: web microservices, REST API, GraphQL API, software performance.

Article Info
Received: April 27, 2026
Accepted: Aug 2, 2026

1. Introduction

Modern computing systems are based on a combination of diverse platforms, programming languages and technologies. These systems communicate through Application Programming Interfaces (APIs), which are essentially mechanisms that allow different software components to interact based on a defined set of protocols and agreements (Amazon Web Services, n.d.). When it comes to web-based systems, implementing APIs via a microservices architecture - a collection of protocols and standards designed to help data exchange and interoperability among web systems - has proven to be the most effective approach (Brito & Valente, 2020).

Two widely adopted web API models are REST (REpresentational State Transfer) and GraphQL. For about the last two decades, REST has been the go-to standard for web microservices (hereinafter microservices) design, largely due to its inherent flexibility, resilience, and ability to scale (Khan et al., 2025; Sayago Heredia et al., 2019). However, REST does come with its share of known drawbacks, such as query complexity that can necessitate multiple HTTP requests, along with

issues of over-fetching, where a client receives more data than it actually needs, and under-fetching, where a single request doesn't provide enough data (Khan et al., 2025; Quiña-Mera et al., 2023).

As a popular alternative, GraphQL provides the solution to many problems with REST by offering a single endpoint, where clients have the power to define exactly the data that's needed to avoid over-fetching and under-fetching (Vadlamani et al., 2021). Its main drawbacks include a high learning curve and an ecosystem that is still maturing (Vadlamani et al., 2021).

The scientific problem is that existing comparisons of REST and GraphQL fail to find an absolute winner between the two technologies because multiple external contextual factors affect the performance observed in such comparisons (Khan et al., 2025; Quiña-Mera et al., 2023; Vadlamani et al., 2021; Lawi et al., 2021). This ambiguity makes it challenging for developers to know which technology is better suited for their specific needs.

This study is motivated by the complementary strengths and weaknesses of REST and GraphQL. Therefore, it posits that combining REST and GraphQL in a single system will lead to better performance than either system in isolation. Studies by Lawi et al. (2021) and Vadlamani et al. (2021) highlight that GraphQL performs better if the data needed by a client changes frequently and that REST is more effective if the same data needs to be fetched multiple times. Nevertheless, the integration between the two architectures has yet to be researched in any depth or modeled in any structured manner.

The overall goal of this research is to develop and empirically test a new microservices integration method that mixes REST and GraphQL technologies to improve the software performance depending on the particular case. More specifically, this research will determine if the existence of both architectures together within the same system can achieve comparable or improved server response times when compared to only using the REST architecture, as proven by performing a statistically relevant and reproducible experiment. Friedman test has been selected for validating the data gathered during the experiment.

The rest of this paper proceeds as follows. Section 2 offers an overview of prior art focusing on comparing REST and GraphQL performance as well as their integration into existing systems. Section 3 explains the chosen method of research, which will consist of presenting a case study, describing our model of coexistence, specifying the data gathering tools, outlining the statistical methods for analysis and the details of the experiment that will be carried out. Section 4 provides the test results obtained, which are validated by the Friedman test applied on the REST and GraphQL microservices configurations. Section 5 discusses how the results correspond to previous findings on REST and GraphQL, along with the limitations of our own research study. Finally, Section 6 concludes the study and points to possible avenues of future research.

2. Related Work

Sayago Heredia et al. (2020) provide a comparative analysis of SOAP, REST and GraphQL for the impact of microservice standards on execution performance of web applications. They conclude that REST performs worse, and SOAP performs worse in response time than GraphQL. But on mobile devices on systems with hardware limitations, REST and GraphQL perform better than SOAP.

Wittern et al. (2018) study the generation of GraphQL wrappers automatically from REST APIs. They show that many OpenAPI specifications contain an ambiguous or incomplete set of information, meaning that it is not possible to migrate to GraphQL entirely. This leads to the interpretation that both systems coexist in practice.

Brito et al. (2019) provide a REST to GraphQL migration study, describing that 29 REST calls were restructured into 24 GraphQL queries, reducing the response JSON size from an average of 93.5 fields to 5.5 fields (from 9.8 megabytes to 86 kilobytes). The reduction in number of calls, however, was modest.

Khan et al. (2025) also performed a comparative analysis of REST and GraphQL regarding performance, efficiency, user experience, and security; they found that for data retrieval, GraphQL reduced response time by 40% and payload size by 42% compared to REST. REST, however, showed better performance under high concurrent request conditions.

Brito and Valente (2020) carry out a controlled experiment with 22 students to evaluate the effort of implementing GraphQL instead of REST queries. They conclude that GraphQL needs less effort, especially if the REST endpoint involves several parameters.

Vadlamani et al. (2021) examines if REST can be replaced with GraphQL based on quantitative as well as qualitative criteria. They conclude that both paradigms have specific benefits and limitations, and neither can replace the other one soon, thus motivating the research into integration methods.

Lawi et al. (2021) propose an evaluation methodology of REST and GraphQL services on a live management information system. REST outperforms GraphQL on response time (50.50% more efficient) and throughput (37.16% more efficient), while GraphQL outperforms REST on resource usage (37.26% less on CPU and 39.74% less on memory usage). As these empirical results show that REST and GraphQL have complementing strengths, the present work aims to investigate these differences in performance by implementing a system in REST, GraphQL, and a combination of REST and GraphQL.

Ala-Laurinaho et al. (2022) compare REST and GraphQL interfaces on OPC UA, showing that GraphQL outperforms REST in reading and writing multiple values together while REST outperforms GraphQL when reading and writing single values. Quiña-Mera et al. (2023) conducted a systematic review of GraphQL, identifying the following areas for further study: GraphQL API service consumption, especially when it comes to integration with REST.

Kurniawati et al. (2025) developed a web system with a RESTful architecture and concluded that REST is suitable for high-load conditions and that GraphQL is a good alternative for applications with access to complex or real-time data. They also identified that a possible hybrid REST-GraphQL architecture could improve efficiency. This last point aligns with the REST and GraphQL integration methodology proposed in this paper.

Unlike the previously reviewed works, which evaluate the two technologies separately, the present study evaluates the performance of both technologies combined within the same system. It seeks to answer the research question of whether or not pure REST is always the best architecture to build software when the minimum response time is required. The results obtained in the conducted experiment to prove this research question resulted in the development of a new research perspective that none of the previous works have investigated.

3. Methodology

This section describes the methodology designed to evaluate the performance implications of integrating REST and GraphQL APIs within a single microservice. This study aims to address the question of feasibility for combining these architectural approaches by investigating a single micro case study: a client-server web application that supports basic CRUD operations for managing users. Given that most client-server interactions involve these fundamental interactions, the study selects an application that implements a basic Create, Read, Update and Delete interface for user records in a relational database as a representative case. Using this architecture, a method to combine REST and GraphQL APIs is proposed. The methodology involves combining multiple microservices: microservices operating entirely in REST, entirely in GraphQL and microservices combining the two technologies in varying proportions. These microservices differ in terms of which CRUD operations are delegated to which architectural approach. Finally, for validation purposes, each microservice is exposed to concurrent request sets and replicated numerous times.

3.1. Case Study

A web application designed for user management was selected as a case study to design and test the feasibility of a REST/GraphQL integration approach. This application functions as a client-server web system allowing multiple simultaneous client connections. The supported user management actions are as follows: create a new user, update an existing user's information, query the details of a specific user, query all users' data, and delete a user from the relational database. Even though the application has few functionalities, the system has the critical components found in client-server systems, whereby users trigger various actions through a GUI. The client-server system, through its APIs, can interact with a relational database according to the requests generated by the users interacting with the GUI. This can be regarded as a simplified representation of many client-server systems common throughout the software industry.

The micro case study is concerned with the implementation of Create, Read, Update, Delete operations on user data. CRUD was chosen because the core interaction model between any application and a database relies on performing operations such as these. It is nearly impossible to find a system that deals with any kind of data in software applications without a combination of Create, Read, Update, and Delete capabilities (Elghazal et al., 2025). Because these are such basic and generic building blocks for most information management systems, they were used as the foundation of the study and experimentation, so that the results would be generalizable and not tied to any specific context. Two APIs were implemented on the same system: one following a REST architecture and one following GraphQL, both supporting five user actions: add a user, retrieve all users, retrieve a specific user, update a user's data, and delete a user.

The APIs were hosted on a local web server running Apache 2.4.48, implemented in PHP 8.3.11, using the Eloquent ORM from Laravel Framework 11.0, webonyx/graphql-php v15.14.0, and backed by MySQL 10.4.21-MariaDB. The relational database contains a *users* table with the fields: *id* (integer), *fname* (string), *lname* (string), *email* (string), and *ingreso* (integer).

3.2. Proposed Architecture

The conventional design of a web application within a client-server system consists of implementing microservices using a single architecture: either REST or GraphQL. In software development practice, developers routinely face the decision of which technology to adopt, which necessarily means forgoing the advantages offered by the other.

Prior research does not identify any fundamental constraint that would preclude the simultaneous integration of both technologies within a single microservice implementation. Combining REST and GraphQL within a single implementation may therefore leverage the strengths of each technology to improve overall performance. This study proposes a case study described in the previous section, involving a minimum set of typical CRUD (Create, Read, Update, Delete) operations (e.g., five CRUD operations, as in the present study). These operations could either be handled entirely by REST or GraphQL (i.e., RESTful operations and a single GraphQL API to interact with these five operations) or they could be split across both, where some are executed by REST and some by GraphQL. The following section describes how the proposed model differs from a common architectural decision for client-server web applications.

3.2.1. Proposed Architectural Design

A client-server web application's architectural design is typically implemented using microservices. Developers of microservices are free to use either REST or GraphQL exclusively for the services. Figure 1 provides a traditional example with two independently implemented microservices, one implemented with REST and another with GraphQL, both accessing the same database.

The proposed architectural model can support the two architectural approaches of REST and GraphQL within a single microservice. From the client (e.g., frontend) side, requests can be sent in either architecture depending on the experimental configuration. The REST API and the GraphQL API access the exact same relational database. Hence, differences in response times should be caused exclusively by API technology and not by differences in the underlying database. The architectural model can be visualized as shown in Figure 2. The REST architecture offers multiple endpoints depending on the HTTP method used, such as GET, POST, PUT, and DELETE, whereas the GraphQL API operates with a single endpoint using the POST method. Operations are differentiated by the query type or mutation type sent in the POST body.

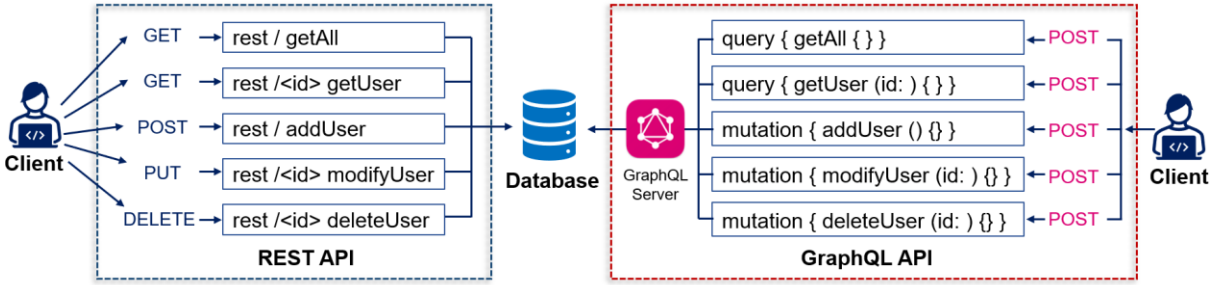

Figure 1. Classic system for implementing separate microservices in REST or GraphQL.

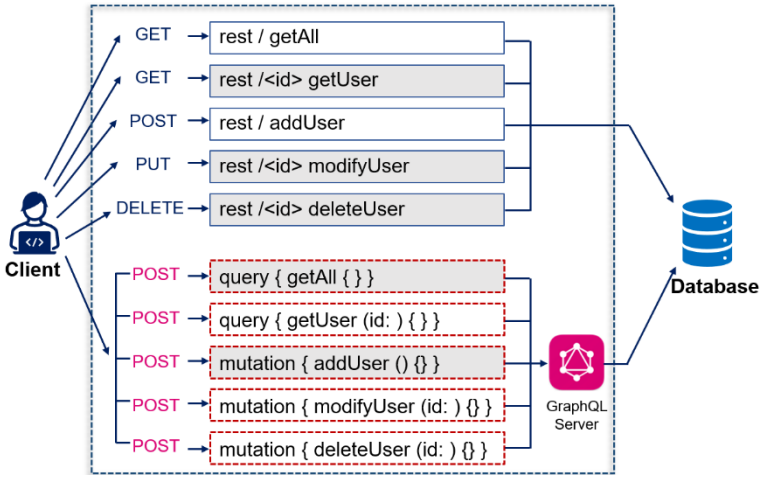

Figure 2. Proposed model for implementing microservices with REST and GraphQL integration.

Figure 1 shows the common approach in which all microservice operations are exclusively supported by REST or GraphQL. Figure 2 shows the architecture proposed to handle both REST and GraphQL requests within a single microservice. Although all operations are specified for each approach in Figure 2, the combination was only used when needed. The design REST and GraphQL within a single microservice in our case study is described below according to each design approach.

3.2.2. REST Design

For each user action mentioned in the case study, a distinct REST API endpoint was provided. The */rest/* prefix with extensions for each method (GET, POST, PUT, DELETE) defines all the API routes (Table 1). Except for retrieving all users (GET /rest/) and creating users (POST /rest/), all other methods require the user's *id* to specify which user the operation targets: retrieving one user (GET /rest/{id}), updating a user (PUT /rest/{id}), and deleting a user (DELETE /rest/{id}). All operations return a JSON string, except for retrieving all users, where the result is a JSON array of user objects. Each user object has the following attributes: *id, fname, lname, email,* and *ingreso* (date of registration).

Table 1. Operations implemented in the REST API.

Action	Endpoint	HTTP Method	Response
getAll (retrieve-all)	/rest/	GET	JSON (list of objects containing request result)
getOne (retrieve-one)	/rest/?id=<id>	GET	JSON (object containing requested record)
addUser (create)	/rest/	POST	JSON (object containing new record)
modifyUser (update)	/rest/?id=<id>	PUT	JSON (object containing updated record)
deleteUser (delete)	/rest/?id=<id>	DELETE	JSON (object containing deleted record)

3.2.3. GraphQL Design

The GraphQL users API has only one endpoint, */graphql/*. All requests use the HTTP POST method. For read requests (retrieve all users, retrieve one user), the query type is used. For write requests (create, update, and delete), the mutation type is used. All methods in this case study (Table 2) perform a CRUD operation on a user. The response is the same as with the REST implementation: a JSON array of users for retrieving all users and a single JSON string for all other operations. The user object definition is the same: *id, fname, lname, email,* and *ingreso* (date of registration).

Table 2. Operations implemented in the GraphQL API.

Action	Endpoint	GraphQL Type	HTTP Method
getAll (retrieve-all)	/graphql/	query { users }	POST
getOne (retrieve-one)	/graphql/	query { user(id) }	POST
addUser (create)	/graphql/	mutation { addUser }	POST
modifyUser (update)	/graphql/	mutation { modifyUser }	POST
deleteUser (delete)	/graphql/	mutation { deleteUser }	POST

3.3. Metrics

To assess the behavior of a server when a microservice exposes a mix of REST and GraphQL APIs, its performance was measured based on the server-side response time. This metric provides a technology-agnostic and direct measure of the efficiency of each API architecture without considering external factors such as network latency or client performance.

The primary metric is the server-side response time for each action, RTa, measured in milliseconds (ms) as the difference between the request reception timestamp, $ts_a^{request}$, and the response dispatch timestamp, $ts_a^{response}$, recorded at the server. Equation (1) defines the response time RT for action a; where $a \in CRUD$, and CRUD = {getAll, getOne, addUser, modifyUser, deleteUser}

$$RT_a = ts_a^{response} - ts_a^{request} \quad (1)$$

3.4. Statistical Analysis

From the primary response time measurement, additional metrics are derived to characterize the overall performance of the microservice as it processes the five concurrent CRUD requests. The average response time for a microservice is computed as shown in Equation (2).

Equation (2) defines the average response time *ART* for a microservice with configuration ***c***, calculated in milliseconds as the sum of the individual response times across each CRUD action ***a***, divided by the total number of actions ***n*** in the microservice. Here, ***c*** is the configuration of one of the 32 microservices described later in this work.

$$ART_c = \frac{1}{n} \sum_{a=1}^{n} \{ RT_a \} \tag{2}$$

Since the distribution of response times for web systems often suffers from extreme outlying measurements, the arithmetic average (mean) is usually insufficient for determining central tendency. Thus, this analysis supplemented by reporting the median response time. Equation (3) MRT is the response time that falls exactly in the middle when response time measurements for ***c*** are sorted according to time into a set of ***n*** records. This approach isn't as sensitive to outliers compared to the average response time (ART).

$$MRT_c = \begin{cases} RT_{[n/2]} & \text{if } n \text{ is odd} \\ \dfrac{RT_{n/2} + RT_{(n/2)+1}}{2} & \text{if } n \text{ is even} \end{cases} \tag{3}$$

In this paper, ART and MRT metrics were employed to quantify and compare server-side performance of the configured microservices.

3.5. Experimental Design

The prevailing technique in web app development uses microservices that expose only either REST or GraphQL. Lawi et al. (2021) claim that REST's response times are better than GraphQL under the most common circumstances. Our approach designs an experiment to test this claim under a novel architecture: Does having REST and GraphQL coexist side-by-side in the same application yield a similar or improved response time? Therefore, two hypotheses are proposed:

Null hypothesis (H_0): A hybrid API (REST and GraphQL) won't have a statistically significant difference in performance in a microservice compared to an exclusive REST implementation.

Alternative hypothesis (H_1): A hybrid API (REST and GraphQL) will have a statistically significant difference in performance in a microservice compared to an exclusive REST implementation, optimizing for response time given the transactional nature of the request.

To test these hypotheses, the web application case study previously proposed is used, in which the client (the frontend) sends five concurrent random requests on behalf of distinct users. The backend (server) receives the requests and processes each according to the following CRUD actions: (1) retrieve all users, or *getAll*; (2) retrieve a specific user, or *getOne*; (3) add a user, or *addUser*; (4) update a user's data, or *modifyUser*; and (5) delete a user, or *deleteUser*.

For each CRUD action within the microservice, its server-side response time is recorded. For simplicity, the set of performance metrics is restricted to only server-side response time. Performance metrics such as throughput, CPU load, and memory usage could be explored in future work.

Implementing the microservices requires defining the level of integration between REST and GraphQL technologies, organized into combination classes that specify the proportion of operations managed by each technology within each request set. Table 3 presents the six combination classes defined for both technologies. For each class, the number of CRUD operations assigned to REST and to GraphQL is specified. Given that each of the five CRUD operations can be independently assigned to either REST or GraphQL, a total of 32 (2^5) distinct configurations are possible, meaning that 32 combined microservices were implemented—one per configuration.

Table 3. Distribution of operations between REST and GraphQL.

Combination Class	Operations with REST	Operations with GraphQL	Distinct Configurations
Class a	0	5	1
Class b	1	4	5
Class c	2	3	10
Class d	3	2	10
Class e	4	1	5
Class f	5	0	1

For each configuration, it is necessary to specify which actions are implemented in REST and which in GraphQL. Table 4 shows the 32 configurations that were implemented as microservices. Note that the first digit indicates the number of operations implemented in REST, and the second digit indicates the number implemented in GraphQL, while the alphabetic letter identifies a sequential index within each class.

Table 4. Configurations of the 32 microservices.

Num	Microservice	Retrieve-all	Retrieve-one	Create	Update	Delete
1	Config05	GraphQL	GraphQL	GraphQL	GraphQL	GraphQL
2	Config14a	REST	GraphQL	GraphQL	GraphQL	GraphQL
3	Config14b	GraphQL	REST	GraphQL	GraphQL	GraphQL
4	Config14c	GraphQL	GraphQL	REST	GraphQL	GraphQL
5	Config14d	GraphQL	GraphQL	GraphQL	REST	GraphQL
6	Config14e	GraphQL	GraphQL	GraphQL	GraphQL	REST
7	Config23a	REST	REST	GraphQL	GraphQL	GraphQL
8	Config23b	REST	GraphQL	REST	GraphQL	GraphQL
9	Config23c	REST	GraphQL	GraphQL	REST	GraphQL
10	Config23d	REST	GraphQL	GraphQL	GraphQL	REST
11	Config23e	GraphQL	REST	REST	GraphQL	GraphQL
12	Config23f	GraphQL	REST	GraphQL	REST	GraphQL
13	Config23g	GraphQL	REST	GraphQL	GraphQL	REST

Num	Microservice	Retrieve-all	Retrieve-one	Create	Update	Delete
14	Config23h	GraphQL	GraphQL	REST	REST	GraphQL
15	Config23i	GraphQL	GraphQL	REST	GraphQL	REST
16	Config23j	GraphQL	GraphQL	GraphQL	REST	REST
17	Config32a	REST	REST	REST	GraphQL	GraphQL
18	Config32b	REST	REST	GraphQL	REST	GraphQL
19	Config32c	REST	REST	GraphQL	GraphQL	REST
20	Config32d	REST	GraphQL	REST	REST	GraphQL
21	Config32e	REST	GraphQL	REST	GraphQL	REST
22	Config32f	REST	GraphQL	GraphQL	REST	REST
23	Config32g	GraphQL	REST	REST	REST	GraphQL
24	Config32h	GraphQL	REST	REST	GraphQL	REST
25	Config32i	GraphQL	REST	GraphQL	REST	REST
26	Config32j	GraphQL	GraphQL	REST	REST	REST
27	Config41a	GraphQL	REST	REST	REST	REST
28	Config41b	REST	REST	REST	*GraphQL*	REST
29	Config41c	REST	REST	*GraphQL*	REST	REST
30	Config41d	REST	*GraphQL*	REST	REST	REST
31	Config41e	*GraphQL*	REST	REST	REST	REST
32	Config50	REST	REST	REST	REST	REST

As described in the experimental model, the client (frontend) submits a set of five concurrent CRUD requests to the server (backend) in randomized order to prevent predictable request patterns that could produce artificially uniform response times. Figure 3 illustrates an example microservice design corresponding to Class d, in which the operations for retrieving all users (*getAll*), retrieving a specific user (*getOne*), and adding a user (*addUser*) are handled by REST, while updating a user's data (*modifyUser*) and deleting a user (*deleteUser*) are handled by GraphQL. This architecture corresponds to only one of the 32 different microservices that are to be analyzed.

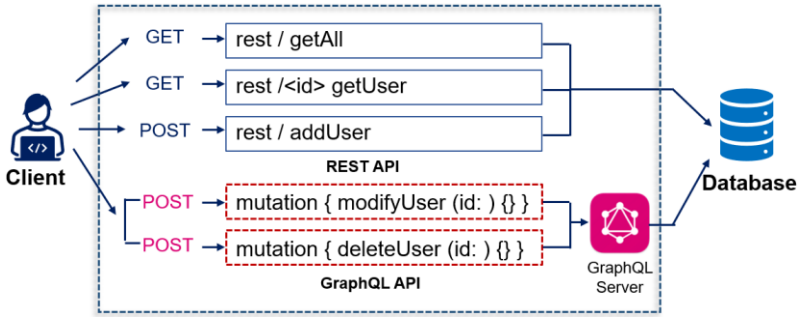

Figure 3. An example of the 'Class d' microservice that implements 3 REST-based CRUD operations along with 2 GraphQL-based CRUD operations.

The variable *volume* was introduced to increase the size of the CRUD request set; its values were set to 1x, 5x, and 10x to multiply the microservice's workload. Each of the 32 microservices was subjected to these loads to generate sufficient data for statistical analysis of server response time. Each microservice received the same number of requests to maintain a level workload. To generate reliable statistical values in this type of system, Montgomery and Runger (2014) state that 30 *samples* (observations) are enough to decrease the uncertainty of the sample mean and variance, so that the latter can be taken as the reliable estimates of the population values.

A custom-made web application developed using HTML, CSS, JavaScript, and PHP was used for the configuration of experimental variables and the dispatching of requests to the server. All results were stored in the database for subsequent statistical analysis.

The experimental design specifies that the 32 microservices must process the set of five CRUD requests, applying three volume levels and generating 30 samples for each microservice. Table 5 shows the test organization, in which the combination of microservices, operations, volume, and samples determines the total number of requests generated. Volume defines the multiplier applied to the number of CRUD operations, while samples correspond to the repetitions needed to reach the minimum number of observations indicated by Montgomery and Runger (2014).

Table 5. Total requests in the experimental design.

Microservices	Operations	Volume	Samples	Requests
32	5	1	30	4,800
32	5	5	30	24,000
32	5	10	30	48,000

As shown in Table 5, when the volume is 1, each of the 32 microservices processes the 5 CRUD operations, repeating the activity 30 times to generate the minimum number of samples for each microservice, resulting in a total of 4,800 requests (32 × 5 × 1 × 30). When the volume is 5, each of the 32 microservices processes 25 (5 × 5) CRUD operations, repeating the activity 30 times, resulting in a total of 24,000 requests (32 × 5 × 5 × 30). Finally, when the volume is 10, each of the 32 microservices processes 50 (5 × 10) CRUD operations, repeating the activity 30 times, resulting in a total of 48,000 requests (32 × 5 × 10 × 30). As a result, a total of 76,800 requests were generated, with the response time recorded for each one.

4. Results

The average response time (ART) of the five CRUD operations was obtained for each of the 32 microservices. Table 6 lists the 32 microservices, ordered from lowest to highest by average response time; the table also indicates whether the CRUD operation was performed using REST or GraphQL.

Table 6 shows that the five microservices in Class e (four REST and one GraphQL CRUD operation) had ARTs lower than that of Config50 (pure REST, ART = 379 ms). Classes a, b, c, and d, which have at least two operations handled by GraphQL, underperformed compared to pure REST. Of all the microservices, Config05 (pure GraphQL) recorded the highest ART (652 ms), indicating the worst performance in the experiment, in line with results found by Lawi et al. (2021).

Table 6. Average response time (ART) per microservice, sorted in ascending order.

Microservice	Retrieve-all	Retrieve-one	Create	Update	Delete	ART (ms)
Config41c	REST	REST	GraphQL	REST	REST	350
Config41e	REST	REST	REST	REST	GraphQL	353
Config41a	GraphQL	REST	REST	REST	REST	358
Config41b	REST	GraphQL	REST	REST	REST	358
Config41d	REST	REST	REST	GraphQL	REST	367
Config50	**REST**	**REST**	**REST**	**REST**	**REST**	**379**
Config32b	REST	REST	GraphQL	REST	GraphQL	401
Config32f	REST	GraphQL	GraphQL	REST	REST	407
Config32h	GraphQL	REST	REST	GraphQL	REST	407
Config32c	REST	REST	GraphQL	GraphQL	REST	409
Config32e	REST	GraphQL	REST	GraphQL	REST	410
Config32i	GraphQL	REST	GraphQL	REST	REST	420
Config32j	GraphQL	GraphQL	REST	REST	REST	423
Config32a	REST	REST	REST	GraphQL	GraphQL	424
Config32d	REST	GraphQL	REST	REST	GraphQL	426
Config32g	GraphQL	REST	REST	REST	GraphQL	438
Config23d	REST	GraphQL	GraphQL	GraphQL	REST	444
Config23f	GraphQL	REST	GraphQL	REST	GraphQL	462
Config23j	GraphQL	GraphQL	GraphQL	REST	REST	472
Config23c	REST	GraphQL	GraphQL	REST	GraphQL	473
Config23g	GraphQL	REST	GraphQL	GraphQL	REST	477
Config23b	REST	GraphQL	REST	GraphQL	GraphQL	484
Config23a	REST	REST	GraphQL	GraphQL	GraphQL	491
Config23i	GraphQL	GraphQL	REST	GraphQL	REST	494
Config14e	GraphQL	GraphQL	GraphQL	GraphQL	REST	503
Config23e	GraphQL	REST	REST	GraphQL	GraphQL	507
Config14d	GraphQL	GraphQL	GraphQL	REST	GraphQL	513
Config23h	GraphQL	GraphQL	REST	REST	GraphQL	516
Config14b	GraphQL	REST	GraphQL	GraphQL	GraphQL	530
Config14c	GraphQL	GraphQL	REST	GraphQL	GraphQL	558
Config14a	REST	GraphQL	GraphQL	GraphQL	GraphQL	568
Config05	**GraphQL**	**GraphQL**	**GraphQL**	**GraphQL**	**GraphQL**	**652**

Based on the results shown in Table 6, a more in-depth statistical analysis is necessary for the five microservices in Class e (four REST and one GraphQL CRUD operation), the Config50 microservice (pure REST), and the Config05 microservice (pure GraphQL).

The first analysis focuses on obtaining the difference in *ART* between REST and GraphQL, as well as between REST and each of the Class e microservices, as shown in Table 7. The Config41c microservice achieves the greatest gain by using GraphQL for the addUser request: its ART of 350.27 ms is 29.17 ms (7.7%) lower than the 379.44 ms ART of pure REST. Other good candidates were the Config41e microservice (delete in GraphQL, 26.20 ms gain) and the Config41a microservice (retrieve-all in GraphQL, 21.70 ms gain). The last of the Class e microservices, Config41d (using GraphQL for the update request), gained the least, at only 12.90 ms (3.4%). This suggests that update requests benefit the least from being handled by GraphQL.

Table 7. Average server-side response time for each microservice in Class e versus pure REST and pure GraphQL.

Comparison	REST Average (ms)	Config. Average (ms)	Difference (ms)	Interpretation
REST vs Config41a	379.43	357.73	+21.70	Config41a faster
REST vs Config41b	379.43	358.27	+21.17	Config41b faster
REST vs Config41c	379.43	350.27	+29.17	Config41c faster
REST vs Config41d	379.43	366.53	+12.90	Config41d faster
REST vs Config41e	379.43	353.23	+26.20	Config41e faster
REST vs Pure GraphQL	379.43	652.20	−272.77	Pure GraphQL inferior

Table 7 shows that all Class e hybrids were faster than pure REST, which in turn was faster than pure GraphQL. Pure GraphQL registered an ART of 652.20 ms, 272.77 ms (71.9%) higher than that of pure REST microservice.

To verify the consistency of response times among the microservices under analysis, a more rigorous analysis was performed using the Friedman test, with a significance level of 0.05. The Friedman test compares k related groups by assigning ranks to each dataset and its resulting statistic follows a chi-square distribution with $k - 1$ degrees of freedom. In this experiment, $k = 7$ groups were analyzed (Config50, Config41a, Config41b, Config41c, Config41d, Config41e, Config05); thus, the resulting statistic is distributed as χ^2 with 6 degrees of freedom. The null hypothesis (H_0) states that the means of the compared groups are all equal, whereas the alternative hypothesis (H_1) states that at least one pair of groups differs. Table 8 shows the results obtained by applying the Friedman test.

Table 8. Friedman test results ($\alpha = 0.05$).

χ^2 Statistic	p-value	Result
17.91081	0.00000	H_0 rejected

Table 8 shows that the χ^2 statistic of 17.911 is greater than the critical value of a chi-square distribution with 6 degrees of freedom for $\alpha = 0.05$ (χ^2critical = 12.592). The p-value of 0.00000 effectively rules out random variation as the cause of differences of this size under H_0. Hence, H_0 is rejected, meaning that at least one microservice under test performed differently from the other microservices in this experiment.

In addition to testing for overall differences, the Friedman test also produces a ranking that reflects each microservice's relative efficiency in handling requests across the 30 samples of the experiment. The result is a ranking where a lower value indicates better performance. The ranking produced by the Friedman test is shown in Table 9.

Table 9. Configuration rankings produced by the Friedman test.

Rank	Configuration / Algorithm	Observation
2.83333	**Config41c (GraphQL ADD – create operation)**	**Best overall performance**
3.46667	Config50 (pure REST)	Reference
3.58333	Config41a (GraphQL ALL – retrieve-all)	
3.63333	Config41b (GraphQL ONE – retrieve-one)	
3.65000	Config41e (GraphQL DEL – delete)	
3.90000	Config41d (GraphQL UPD – update)	Worst within Class e
6.93333	**Pure GraphQL**	**Worst overall performance**

Table 9 shows that the Config41c microservice achieved the lowest rank (2.833), making it the highest-performing microservice among the set analyzed, better than the Config50 microservice (pure REST, rank 3.467) and the rest of the Class e operations (ranks between 3.583 and 3.900). In contrast, the Config41d microservice, with the lowest performance in Class e, achieved the highest rank (3.900), confirming that replacing the update operation with GraphQL provides the smallest performance benefit. Pure GraphQL finished last with a rank of 6.933, further emphasizing that it performed poorly under the experimental conditions of this study.

Due to the significant difference observed in the Friedman test for the complete set of measurements, a Bonferroni-Dunn post-hoc test was performed to determine the source of the discrepancies. This test involves comparing one group with the other groups in pairs, rejecting H_0 when $p < \alpha/(K-1)$, using the group with the lowest ART and Friedman rank as the reference (control) group. Table 10 summarizes the results of comparing REST with the others microservices using the Config41c microservice as control (lowest Friedman rank and ART of 350.27 ms) against the remaining microservices.

Table 10. Bonferroni-Dunn post-hoc test with Config41c as the control method ($\alpha = 0.05$).

Comparison	Statistic	Adjusted p-value	Result
REST vs Pure GraphQL	**6.21519**	**0.00000**	**H_0 rejected (significant difference)**
REST vs Config41c (GraphQL ADD)	1.13547	1.00000	H_0 not rejected (no significant difference)
REST vs Config41d (GraphQL UPD)	0.77690	1.00000	H_0 not rejected (no significant difference)
REST vs Config41e (GraphQL DEL)	0.32869	1.00000	H_0 not rejected (no significant difference)
REST vs Config41b (GraphQL ONE)	0.29881	1.00000	H_0 not rejected (no significant difference)
REST vs Config41a (GraphQL ALL)	0.20917	1.00000	H_0 not rejected (no significant difference)

Table 10 shows that the only comparison in which the null hypothesis (H_0) can be reject is when REST vs. Pure GraphQL is compared (value of statistic is 6.215 and the adjusted p-value is 0.000). The difference of 272.77 ms between REST and pure GraphQL is significant. In the remaining comparisons, namely REST vs. Config41a (statistic is 0.209), REST vs. Config41b (statistic is 0.299), REST vs. Config41c (statistic is 1.135), REST vs. Config41d (statistic is 0.777) and REST vs. Config41e (statistic is 0.329), the adjusted p-value is 1.000 for every case, so H_0 is not rejected in every comparison. Small values of statistic together with adjusted p-values of 1.000 leave no statistical space to claim that there's a noticeable difference in the behavior of the microservices in Class e and REST based on the Bonferroni-Dunn post-hoc test ($p < \alpha/(K - 1) = 0.05/6 = 0.0083$ for the test result).

5. Discussion

The experimental setup aims to falsify the null hypothesis that there is no statistically significant response-time difference between a REST-GraphQL hybrid architecture and a solely REST architecture. The statistical findings are not a simple accept/reject of the null hypothesis. Globally, the Friedman test rejects H_0 ($\chi^2 = 17.911$, $p = 0.000$), which indicates that the setting under observation is not statistically uniform. Post-hoc testing using Bonferroni-Dunn reveals the source of the non-uniformity is the gap between purely REST and purely GraphQL, not any gap between hybrid microservices in Class e and pure REST. For all five Class e setups, the test fails to reject H_0 (adjusted $p = 1.000$ in all cases). In other words, running a microservice with four REST operations alongside one GraphQL operation has no statistical impact on server-side response time when contrasted with only using REST.

This observation undermines common fears that introducing any GraphQL to a REST service is guaranteed to degrade performance. As supported by the data, the integration of GraphQL into a few operations (primarily addUser and getAll) has no detectable, statistical cost compared to an entirely REST system. Every one of the Class e sample averages is lower than pure REST, with savings ranging from 12.90 ms to 29.17 ms. None of the differences reach the Bonferroni-Dunn significance value, but it is interesting to note that the hybrid Class e designs lead to times comparable to those of pure REST.

Some of the present findings align with the work of Lawi et al. (2021), who found that REST outperforms GraphQL in terms of response time (50.50%) and throughput (37.16%) under normal load. Our system produced in an *ART* of 652.20 ms for pure GraphQL versus 379.43 ms for pure REST, a 71.9% increase. The direction is similar, although the difference is more pronounced because of deviations in the experimental setup, including the testing platform, size of the data, and number of concurrent requests-factors that suggest comparisons of API architectures do not generalize well from one setup to another.

What differentiates our research is the focus on the hybrid REST and GraphQL integration. Studies found through the literature review (Sayago Heredia et al., 2020; Vadlamani et al., 2021; Brito et al., 2019; Ala-Laurinaho et al., 2022) do not specifically look into microservices which have both architectures run at the same time. Vadlamani et al. (2021) pointed out that REST and GraphQL cannot practically replace one another anytime soon, a gap that tries to fill with hybrid models. Results in this study provide a partial answer by showing that coexistence between the two styles does not come at any performance penalty.

Ala-Laurinaho et al. (2022) observed that GraphQL is more efficient for multiple simultaneous reads/writes, while REST is faster for individual values. This may offer an architectural basis for why Config41c microservice (GraphQL create) and Config41a microservice (GraphQL retrieve-all) produce the highest *ART* reductions within Class e, where retrieve-all transfers a full collection of records and create writes a completely new entry, both involving an exchange of many field values over a single request trip.

In contrast, the Config41d microservice (GraphQL update) provides the smallest *ART* gain within Class e: 12.90 ms (3.4%). When a record needs to be updated, the operation requires a specific record ID for the user record that should be changed, meaning there are not many values to transmit to update the record's existing fields. The complexity of GraphQL processing−schema parsing, mutation processing, and serialization−could then nullify the advantages otherwise gained by specifying only required fields. This result is in line with what was found by Lawi et al. (2021) regarding the greater per-request computations demands of GraphQL.

It can be concluded that architects creating microservices in client-server systems can realistically consider a hybrid approach: route create and read operations through GraphQL and all other operations (update, delete) through REST to reduce the overall server response time while keeping the performance difference between this hybrid approach and pure REST within the range of a negligible margin. Based on the statistical equivalence observed between Class e hybrids and pure REST, combined with the benefits of GraphQL (no over-fetching/under-fetching, full control over specified fields; Vadlamani et al., 2021), this hybrid architecture can be considered a strong option in performance-critical microservice systems.

In this set of microservices, the Config41c microservice had the highest practical relevance, showing an *ART* of 350.27 ms and the best Friedman rank in this set (2.833). Assigning the create operation to GraphQL, a common high-volume event for client applications needing to add user records, works well for the application, as it also entails writing a well-defined set of fields−a clear fit for GraphQL's granular precision.

The data present a more tempered case for update and delete operations. Config41d and Config41e microservices yield the smallest *ART* reductions within Class e, with the Config41d microservice showing a reduction of merely 12.90 ms. The additional overhead of GraphQL mutations when updating or deleting existing records partially counteracts the performance gains achievable with granular field selection.

When processing requests in a local environment, network latency is virtually nonexistent. This can be seen as a threat to the external validity of results, since in a production server, the interaction between the frontend and backend can travel across networks with varying latency. This could alter the distribution of REST and GraphQL loads, impacting response times and making them different from those obtained in a local environment. Another limitation to consider is that a local environment offers a dedicated CPU, high-speed local disk, and unconstrained memory. This contrasts with shared servers on local networks or cloud infrastructure, where performance differences can be masked, potentially making the advantages of hybrid microservices indistinguishable. These limitations will be addressed in the next section.

6. Conclusions

This work contributes to microservices architecture and web API performance research by proposing, describing, and experimentally validating a hybrid REST-GraphQL architectural model that incorporates both technologies within a single microservice, assigning each of the five CRUD operations (create, retrieve all, retrieve one, update, and delete) of the user data model to either REST or GraphQL. This resulted in 32 distinct microservice configurations, of which those combining four REST operations with one GraphQL operation showed the best performance. Existing literature has investigated REST (Brito et al., 2019), GraphQL (Lawi et al., 2021), or REST with added GraphQL query capabilities (Vadlamani et al., 2021), but does not provide guidance or experimental evidence for a hybrid approach within the same microservice.

A controlled and reproducible experiment was designed with a total volume of 76,800 requests distributed equally among all microservices. The experiment contained 32 different microservices, each evaluated during 30 samples with varying traffic volume levels.

Although the hybrid Class e approach (4 REST and 1 GraphQL) shows the lowest average response times (12.90–29.17 ms faster than pure REST across the five comparisons), it still results in indistinguishable response times relative to the pure REST microservice under the Bonferroni-Dunn correction test (adjusted p-value = 1.000 for all five comparisons).

It was found that operation type has a significant impact on how much performance improvement the technology substitution offers, suggesting that create (addUser) and retrieve-all (getAll) operations benefited the most from using GraphQL in a mostly REST microservice, while update (modifyUser) showed the smallest improvement.

Retrieve-all (getAll) and create (addUser) involve a substantial exchange of multiple data fields in a single round trip, where GraphQL's granular precision in restricting and specifying only the required fields results in a considerable reduction in payload size. This gain in transmission efficiency outweighs the processing costs on the server, minimizing the final response time. The update (modifyUser) and delete (deleteUser) operations, by contrast, are executed using mutation types in GraphQL, requiring a high computational load due to schema parsing, internal mutation processing, and subsequent serialization. Modifying or deleting a record, however, does not involve transmitting large volumes of data to the server. With few fields to optimize, the computational overhead of mutation largely offsets any advantage offered by field selection, slowing the response compared to a direct REST approach.

REST and GraphQL operations interact with the same relational database, resulting in identical execution times for SQL queries at the storage engine. The observed variations in the performance of these operations are due more to the computational and processing requirements of each architecture (REST and GraphQL) than to the database interaction itself.

The H_0 test did not detect any statistically significant performance difference between a hybrid REST-GraphQL architecture and a pure REST architecture. However, the overall result requires more nuanced interpretation. The Friedman test failed to rejected H_0 ($\chi^2 = 17.911$, $p = 0.000$, 6 degrees of freedom), indicating that the observed variations across all test configurations are unlikely to be due to chance. Nevertheless, the Bonferroni-Dunn post-hoc test shows that these variations are specifically

attributable to the difference between pure REST and pure GraphQL. For each of the five Class e microservices (4 REST and 1 GraphQL) compared with pure REST, the test did not reject H_0 (adjusted p-value = 1.000 for all five comparisons).

The above means that the hybrid Class e architecture (4 REST operations and 1 GraphQL operation) performed at a level statistically indistinguishable from pure REST according to the study's parameters. Therefore, H_0 holds when each hybrid microservice in Class e is compared with pure REST, but is rejected when pure REST is compared with pure GraphQL. This indicates that integrating specific GraphQL operations into a REST-based microservice does not compromise response time, as the experimental data show that comparable performance can be achieved.

Further directions for future work include running the experiment using volumes 5 and 10, meaning 24,000 and 48,000 requests, respectively, to determine whether the volume-1 behavior holds under greater contention. Higher server load could alter the distribution of response times, potentially challenging some of the assumptions underlying the current findings.

Another approach for measuring different aspects would be to expand metrics to CPU consumption, memory usage, throughput, etc., since Lawi et al. (2021) report reductions in CPU usage (37.26%) and memory usage (39.74%) with GraphQL, so it'd be interesting to see whether that extends to the Class e hybrids (4 REST operations and 1 GraphQL operation) described in this paper.

Future work could also replicate the experiment on a cloud-based provider or over a network connection with variability in latency to examine whether the findings hold in a production-like setting, as opposed to the local server setting used in this study, which does not account for network latency.

In addition, the case study could be extended to multiple entities per schema, deeply nested queries, and dynamic filter criteria to test whether the hybrid advantage observed holds true as complexity increases. This is because GraphQL is known to excel in exactly these types of scenarios (Vadlamani et al., 2021; Ala-Laurinaho et al., 2022).

Another direction for future research is to test the microservices in Classes b, c, and d, where two, three, or four operations are assigned to GraphQL, respectively. Designing workloads in which clients request fields at different frequencies may reveal cases where the reduction in over-fetching achieved by GraphQL outweighs the increased per-request overhead observed in this study.

Finally, the use of statistically more powerful tests in future experiments involving datasets of greater complexity or a larger proportion of GraphQL requests would provide additional evidence to confirm or refute the trends documented in this research.

Acknowledgments

This work was supported by funding for high-quality graduate studies under the Program for the Professional Development of Faculty Members (PRODEP), Type S247, under project number M00/1525/2023 and ID PTC 131626. The authors also acknowledge the support provided by the Tecnológico Nacional de México (TecNM), through the Instituto Tecnológico de Ciudad Madero, for the use of the Laboratorio Nacional de Tecnologías de la Información (LaNTI), as well as the support received through the TecNM projects with code 25083.26-P "Optimización inteligente con recuperación de información aumentada para mallas ferroviarias de la Red Nacional de Pasajeros" and project code 25139.26-P "Desarrollo de un agente virtual inteligente de apoyo a la toma de decisión con impacto en la navegación de vehículos eléctricos, fase 2".

References

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

Amazon Web Services. (n.d.). *Microservices*. Retrieved May 25, 2026, from https://aws.amazon.com/microservices/

Amazon Web Services. (n.d.). *What is an API (application programming interface)?* Retrieved May 25, 2026, from https://aws.amazon.com/what-is/api/

Bogutskii, V. (2025). Features of REST API development for high-load systems. *International Journal of Scientific Engineering and Science, 9*(3), 67–71. Journal PDF

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

Brito, G., & Valente, M. T. (2020). REST vs GraphQL: A controlled experiment. In *2020 IEEE International Conference on Software Architecture (ICSA)* (pp. 81–91). IEEE. https://doi.org/10.1109/ICSA47634.2020.00016

Devalla, S. (2024). Enterprise-scale evaluation of REST and GraphQL: Balancing performance, scalability, and resource utilization. *International Journal of Core Engineering & Management, 7*(12), 396–416. Official journal copy

Elghazal, M. S. M., Aneiba, A., & Shahra, E. Q. (2025). Performance evaluation of REST and GraphQL API models in microservices software development domain. In *Proceedings of the 21st International Conference on Web Information Systems and Technologies: Volume 1—WEBIST* (pp. 83–91). SciTePress. https://doi.org/10.5220/0013729900003985

China, C. R. (2024, March 29). *GraphQL vs. REST API: What's the difference?* IBM. IBM article

Jin, R., Cordingly, R., Zhao, D., & Lloyd, W. (2025). GraphQL vs. REST: Investigating performance and scalability for serverless data persistence. In *2025 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 155–161). IEEE. https://doi.org/10.1109/IC2E65552.2025.00031

Khan, I. A., Mishra, H., & Choubey, K. (2025). A comparative analysis of REST and GraphQL APIs: Performance, efficiency, and developer experience. *International Journal of Advanced Multidisciplinary Scientific Research, 8*(4), 29–39. https://doi.org/10.31426/ijamsr.2025.8.4.8212

Kurniawati, D., Kusjani, A., Buwono, R. C., & Ridhoni, M. A. (2025). Pengembangan sistem transformasi dan konversi data berbasis web menggunakan arsitektur RESTful API. *Bulletin of Computer Science Research, 5*(6), 1395–1402. https://doi.org/10.47065/bulletincsr.v5i6.818

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

Montgomery, D. C., & Runger, G. C. (2013). *Applied statistics and probability for engineers* (6th ed.). Wiley.

Quiña-Mera, A., Fernandez, P., García, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A systematic mapping study. *ACM Computing Surveys, 55*(10), Article 202. https://doi.org/10.1145/3561818

Sayago Heredia, J., Flores-García, E., & Solano, A. R. (2020). Comparative analysis between standards oriented to web services: SOAP, REST and GRAPHQL. In M. Botto-Tobar, M. Zambrano Vizuete, P. Torres-Carrión, S. Montes León, G. Pizarro Vásquez, & B. Durakovic (Eds.), *Applied technologies* (pp. 286–300). Springer. https://doi.org/10.1007/978-3-030-42517-3_22

Soni, A., & Ranga, V. (2019). API features individualizing of web services: REST and SOAP. *International Journal of Innovative Technology and Exploring Engineering, 8*(9S), 664–671. https://doi.org/10.35940/ijitee.I1107.0789S19

Vadlamani, S. L., Emdon, B., Arts, J., & Baysal, O. (2021). Can GraphQL replace REST? A study of their efficiency and viability. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)* (pp. 10–17). IEEE. https://doi.org/10.1109/SER-IP52554.2021.00009

Wittern, E., Cha, A., & Laredo, J. A. (2018). Generating GraphQL-wrappers for REST(-like) APIs. In T. Mikkonen, R. Klamma, & J. Hernández (Eds.), *Web engineering* (pp. 65–83). Springer. https://doi.org/10.1007/978-3-319-91662-0_5