

Design Trade-offs in Public Web APIs: An Empirical Comparison of REST and GraphQL under Multi-Entity Access Patterns

Valter Granato Neto
Federal University of Lavras (UFLA)
Department of Computer Science (DCC)
Lavras-MG, Brazil
valtergranaton@gmail.com

Maurício Souza
Federal University of Lavras (UFLA)
Department of Computer Science (DCC)
Lavras-MG, Brazil
mauricio.ronny@ufla.br

Johnatan Oliveira
Federal University of Lavras (UFLA)
Department of Computer Science (DCC)
Lavras-MG, Brazil
johnatan.oliveira@ufla.br

Neumar Malheiros
Federal University of Lavras (UFLA)
Institute of Sciences, Technology and Innovation
(ICTIN)
São Sebastião do Paraíso-MG, Brazil
neumar@ufla.br

ABSTRACT

Web APIs are essential to modern Web systems, but providers of public APIs cannot always anticipate how external clients will retrieve and combine related resources. API design choices can therefore affect response time, client-side composition effort, maintainability, and system evolution. This paper empirically compares three Web API design strategies for multi-entity data retrieval: resource-oriented REST, aggregated REST, and GraphQL. We implemented all alternatives using the same technology stack and database schema and evaluated them under 12 controlled scenarios varying the number of related entities and retrieved records. Results show that resource-oriented REST becomes inefficient when clients require multiple sequential requests to compose related data, while GraphQL reduces response time by retrieving nested data in a single request. However, when access patterns are known in advance, aggregated REST achieved lower median response times than GraphQL, with API-G being approximately 1.06 to 1.17 times slower across multi-entity scenarios. These findings reveal a key trade-off: aggregated REST favors efficiency for stable access patterns, whereas GraphQL provides greater flexibility for heterogeneous and evolving client needs. The study provides empirical evidence to support Web API design decisions balancing efficiency, flexibility, and evolvability.

KEYWORDS

Web APIs, API Design, Multi-Entity Data Retrieval, REST, GraphQL, Software Architecture Evolution

1 INTRODUCTION

Web APIs have become a fundamental infrastructure for modern Web systems, enabling communication among Web

applications, mobile clients, microservices, dashboards, third-party services, and distributed software ecosystems [8]. In Web and multimedia systems, such as content platforms, analytics dashboards, mobile applications, and third-party integrations, APIs frequently expose interconnected resources whose retrieval patterns vary across clients and usage contexts.

Consider a public Web API consumed by a mobile application, an analytics dashboard, and external partners. While one client may need only a list of products, another may need products, categories, reviews, and recommendations in a single view. If the API exposes only isolated resources, clients may issue hundreds or thousands of sequential requests to compose a single screen. This scenario illustrates why API design is not merely an implementation decision, but an architectural decision that may affect performance, scalability, client-side complexity, maintainability, and the long-term evolution of Web-based systems.

Public Web APIs are particularly challenging because providers often cannot fully anticipate how external clients will consume and combine available resources. Different clients may require different combinations of related entities, response structures, and data granularities. As a result, data retrieval becomes a central concern in the design and evolution of Web APIs, especially when access patterns are heterogeneous or change over time.

REST (*Representational State Transfer*) is one of the most widely adopted architectural styles for building Web APIs [9]. Proposed by Fielding [3], REST follows a resource-oriented model based on principles such as stateless communication, resource identification through URIs, and the use of standard HTTP methods. However, when clients need data from multiple related resources, resource-oriented REST APIs may require several sequential requests to compose the final response. This request chaining can increase latency, transfer composition logic to the client, and make API evolution more difficult when new access patterns emerge.

GraphQL was introduced as an alternative approach for API design, allowing clients to specify the structure of the

In: Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia'2026). Lavras/MG, Brazil. Porto Alegre: Brazilian Computer Society, 2026.

© 2026 SBC – Brazilian Computing Society.
ISSN 2966-2753

data they need through a schema-based query language [4]. By enabling nested data retrieval through a single endpoint, GraphQL can reduce problems such as *over-fetching* and *under-fetching*, while offering a flexible query model for heterogeneous clients. However, this flexibility may also introduce additional server-side costs, including query parsing, schema validation, resolver execution, and query complexity management [2].

Previous studies have compared REST and GraphQL in terms of performance, usability, maintainability, and resource consumption [5, 10]. In general, these studies indicate that REST tends to perform well in simple and predictable operations, whereas GraphQL may be advantageous when clients need flexible or nested data retrieval. However, many evaluations focus on isolated resources or simple scenarios, and less attention has been given to multi-entity access patterns, which are common in public Web APIs.

Moreover, existing comparisons often frame the discussion as REST versus GraphQL, although REST itself can be designed in different ways. A REST API may expose one endpoint per resource, requiring clients to compose related data through multiple requests, or it may provide aggregated endpoints tailored to specific access patterns. This distinction is important because observed performance differences may result not only from the API technology adopted, but also from where data composition takes place: on the client side, in predefined server-side endpoints, or in client-defined server-side queries.

This paper investigates how different Web API design strategies affect client-perceived response time under multi-entity scenarios. We compare three implementations that expose the same underlying data model: (i) a resource-oriented REST API, in which each entity is retrieved through a separate endpoint; (ii) an aggregated REST API, in which related data are returned through a single endpoint tailored to a known access pattern; and (iii) a GraphQL API, in which clients retrieve nested data through a single query. By comparing these alternatives under the same experimental setting, we analyze how request chaining, server-side aggregation, and client-defined nested queries affect data retrieval performance.

The results indicate that resource-oriented REST becomes inefficient when clients perform multiple sequential requests to compose related data. In these scenarios, GraphQL reduces response time by retrieving nested data through a single request. However, when the required access pattern is known in advance and implemented as an aggregated REST endpoint, aggregated REST achieved the lowest median response times in most of the evaluated multi-entity scenarios.

The main contributions of this paper are as follows: (i) an empirical comparison of three Web API design strategies for multi-entity data retrieval: resource-oriented REST, aggregated REST, and GraphQL; (ii) an analysis of how request chaining, response aggregation, and data volume affect client-perceived response time in Web API scenarios; (iii) empirical evidence on performance trade-offs and a design-oriented analysis of flexibility and maintainability implications in public Web API design; and (iv) practical implications to support

architectural decisions in the development and evolution of Web-based systems.

Although API design affects several quality attributes, this study empirically focuses on client-perceived response time under controlled multi-entity retrieval scenarios. Other attributes, such as maintainability and evolvability, are discussed analytically based on the distribution of data composition responsibilities.

This paper is structured as follows. Section 2 presents the study design. Section 3 reports and discusses the empirical results. Section 4 reviews related work. Section 5 discusses threats to validity. Finally, Section 6 presents the conclusions and future work.

2 STUDY DESIGN

This section presents the design of our empirical study. We describe the study goal, research questions, API implementations, evaluation procedure, statistical analysis, dataset, and experimental assumptions. The study was designed to compare Web API design strategies under controlled conditions, focusing on client-perceived response time in multi-entity data retrieval scenarios.

2.1 Goal and Research Questions

The goal of this study is to evaluate how different Web API design strategies affect client-perceived response time when retrieving data with varying levels of entity complexity. We focus on multi-entity access patterns, which are common in public Web APIs consumed by heterogeneous clients, such as Web applications, mobile applications, dashboards, and third-party services.

Rather than treating the comparison only as REST versus GraphQL, this study considers three API design strategies: resource-oriented REST, aggregated REST, and GraphQL. This distinction is important because REST APIs can be designed in different ways, and their performance may depend not only on the architectural style itself, but also on how data retrieval endpoints are structured and where data composition takes place. Therefore, the study analyzes the distribution of data composition responsibilities among clients, predefined server-side endpoints, and client-defined server-side queries.

The study addresses four research questions: **RQ1**: How do resource-oriented REST, aggregated REST, and GraphQL compare in single-entity retrieval? **RQ2**: How do these strategies behave when retrieving multiple related entities? **RQ3**: How does data volume affect their response-time growth? and **RQ4**: What design trade-offs emerge for public Web APIs? RQ1–RQ3 empirically examine performance across retrieval complexity and data volumes, whereas RQ4 interprets the observed behavior in terms of performance, flexibility, client-side composition effort, and maintainability.

2.2 API Implementations

We implemented three APIs that expose the same underlying data model through different design strategies. All implementations were developed using Java 21, Spring Boot 3.3.5, JPA, and PostgreSQL 16.1. The APIs were deployed using Docker and shared the same database schema, data volume, persistence strategy, and execution environment. This setup was adopted to reduce implementation variability and isolate the effect of API design decisions on response time.

The first API, named **REST-R1**, follows a resource-oriented REST design, in which each entity is exposed through a separate endpoint. When clients need related data, they must perform multiple HTTP requests and compose the final response on the client side. For example, after retrieving a list of **Pokemon** records, the client must issue additional requests to retrieve the related **Ability**, **Item**, or **PokemonForm** records required by a given scenario. REST-R1 is not intended to represent an optimized REST solution, but a resource-oriented design commonly found in public APIs that expose resources independently.

The second API, named **REST-R2**, follows an aggregated REST design. In this implementation, related data are returned through a single endpoint tailored to the evaluated access pattern. REST-R2 was included to avoid reducing the comparison to a binary REST-versus-GraphQL comparison, representing the case in which REST is designed with prior knowledge of the required data composition. Aggregated REST is related to API aggregation strategies, such as Backends for Frontends, in which server-side endpoints are tailored to specific client needs [6].

REST-R2 should not be interpreted as canonical resource-oriented REST, but as a pragmatic RESTful aggregation strategy commonly adopted in Web systems to reduce client-side composition. Its inclusion allows us to separate the cost of HTTP request chaining from the cost of GraphQL processing.

The third API, named **API-G**, uses GraphQL. In this implementation, clients retrieve nested data through a single query submitted to a GraphQL endpoint. This strategy allows clients to specify the structure of the requested data, offering a flexible query model when different clients require different combinations of entities.

Together, these implementations allow us to compare where data composition takes place: on the client side in REST-R1, in a predefined server-side endpoint in REST-R2, and in a client-defined server-side query in API-G. For comparability, the GraphQL queries were designed to retrieve the same set of fields returned by the corresponding REST responses in each scenario. Therefore, the comparison focuses on the effect of data composition strategy rather than on differences in the amount of information requested.

No caching mechanism, database tuning, operating system optimization, or API-specific optimization was applied. This decision was made to compare the API design strategies under equivalent and controlled conditions. The goal was not to identify the best possible optimized implementation of each technology, but to analyze how different API design

strategies behave under the same baseline implementation assumptions.

2.3 Evaluation Procedure

We defined 12 scenarios by combining four levels of entity complexity with three dataset sizes. The entity complexity levels range from a single entity to four related entities, while the dataset sizes include 100, 500, and 1000 **Pokemon** records. The entity combinations were: (i) **Pokemon**; (ii) **Pokemon** + **Ability**; (iii) **Pokemon** + **Ability** + **Item**; and (iv) **Pokemon** + **Ability** + **Item** + **PokemonForm**.

Requests were executed locally using a Python 3.12.3 script. The script sent HTTP requests to each API and recorded the total response time from the client perspective, including request transmission, server-side processing, database access, serialization, and response transfer. Running the experiments locally allowed us to reduce external network variability and focus on the impact of API design decisions under controlled conditions. Therefore, the results should be interpreted as controlled measurements of design-strategy behavior rather than as production-level benchmarks under distributed or high-concurrency conditions.

Each scenario was executed 100 times with a 3-second interval between consecutive executions. We used the median response time as the main descriptive statistic because response time measurements are often affected by outliers and non-normal distributions. The comparison among the APIs considered absolute response times, execution time ratios, and normalized median values to analyze growth trends across entity complexity and data volume.

2.4 Statistical Analysis

We adopted a non-parametric statistical analysis procedure to inspect the response time distributions. First, the Shapiro-Wilk test was applied to assess normality. Since the distributions were not normally distributed, the Mann-Whitney U test was used as a complementary check when comparing API implementations.

The statistical tests were used to support the interpretation of differences among the evaluated strategies. However, the discussion emphasizes median response times, execution time ratios, and growth trends, since these measures provide a direct interpretation of practical performance differences in the context of Web API design. Thus, the analysis combines statistical evidence with effect-oriented interpretation of the observed response-time behavior.

2.5 Dataset and Schema

The dataset is based on PokeAPI¹, an open Web API for the Pokemon game series. We used this domain because it represents a public API context in which a central entity is connected to multiple related entities. Although the domain is specific, this structure is common in Web systems such as e-commerce platforms, social networks, content management systems, media platforms, and dashboards.

¹<https://pokeapi.co/>

The relational model was implemented in PostgreSQL and includes the following entities: **Pokemon**, **Ability**, **Item**, and **PokemonForm**. The **Pokemon** entity represents the central resource, while the other entities represent related information that may be required by different clients.

Each experimental scenario used one of the following entity combinations: (i) **Pokemon**; (ii) **Pokemon + Ability**; (iii) **Pokemon + Ability + Item**; and (iv) **Pokemon + Ability + Item + PokemonForm**. Each combination was evaluated with 100, 500, and 1000 **Pokemon** records, resulting in 12 scenarios. Table 1 shows the total amount of data retrieved in each configuration.

The evaluated scenarios increase both in the number of records and in the amount of related data retrieved. Retrieving only **Pokemon** records produces relatively small responses, ranging from 12.81 KB to 127.73 KB. However, when related entities are included, the response size grows substantially, reaching 1.1 MB in the largest configuration. This variation allows us to evaluate not only how each API design behaves with more records, but also how it responds to increasingly complex multi-entity retrieval scenarios. Therefore, the dataset supports the analysis of both data volume and data composition effects on API response time.

Table 1: Amount of data retrieved in each test scenario.

Entities	Pokemon	+Ability	+Item	+Form
100 Pokemon	12.81 KB	36.01 KB	159.54 KB	176.11 KB
500 Pokemon	63.45 KB	172.65 KB	651.59 KB	745.61 KB
1000 Pokemon	127.73 KB	342.05 KB	925.33 KB	1.1 MB

2.6 Experimental Assumptions

The comparison should be interpreted as an evaluation of API design strategies rather than a definitive ranking of REST and GraphQL technologies. REST-R1, REST-R2, and API-G represent different ways of distributing data composition responsibilities between clients and servers. In REST-R1, composition is delegated to the client through multiple requests. In REST-R2, composition is performed by a predefined aggregated endpoint. In API-G, composition is expressed by the client through a GraphQL query and resolved on the server side.

The experiments do not aim to represent fully optimized production APIs, but baseline implementations executed under equivalent conditions. Backend optimization strategies, such as join fetching, batching, data loaders, database indexing, and caching, could affect absolute response times. However, these optimizations are orthogonal to the main goal of this study, which is to compare how different data composition strategies affect client-perceived response time under the same experimental assumptions.

Accordingly, the results should not be generalized as universal claims that one API technology is always superior to another. Instead, they should be interpreted within the scope of the evaluated scenarios: a controlled local environment,

sequential requests, a relational data model, three data volumes, four levels of entity complexity, and non-optimized implementations sharing the same technology stack.

3 RESULTS

This section presents and discusses the results of the empirical study. The analysis is organized according to the research questions defined in Section 2. We first examine single-entity retrieval scenarios, then analyze multi-entity access patterns, and finally discuss how data volume and request composition affect response-time growth in each Web API design strategy. The Mann-Whitney U test was used as a complementary check and indicated statistically significant differences in the main pairwise comparisons discussed in this section ($p < 0.05$). However, the interpretation emphasizes median response times, execution-time ratios, and observed growth trends, since these measures provide a direct practical view of the differences among the evaluated strategies.

3.1 RQ1 – Performance in Single-Entity Scenarios

RQ1 – How do resource-oriented REST, aggregated REST, and GraphQL compare in single-entity data retrieval scenarios?

RQ1 investigates how resource-oriented REST, aggregated REST, and GraphQL compare when the client retrieves data from a single entity. In this scenario, the requested resource is **Pokemon**, and all API implementations require only one HTTP request.

Table 2 shows the median response times for single-entity retrieval, considering three dataset sizes. REST-R1 achieved median response times of 36 ms, 43 ms, and 55 ms for 100, 500, and 1000 **Pokemon** records, respectively. REST-R2 achieved similar values, with 35 ms, 43 ms, and 54 ms. API-G achieved 40 ms, 48 ms, and 63 ms for the same dataset sizes.

Table 2: Median response time (ms) for single-entity retrieval.

API	100 records	500 records	1000 records
REST-R1	36	43	55
REST-R2	35	43	54
API-G	40	48	63

The ratios indicate that API-G was, on average, 1.12 times slower than REST-R1 and 1.14 times slower than REST-R2. Conversely, REST-R1 required approximately 89% of the execution time observed for API-G. These differences are relatively small in absolute terms, but they are consistent across the three dataset sizes.

These results indicate that GraphQL introduces additional overhead in simple and predictable data retrieval scenarios under the evaluated implementation assumptions. Although API-G also uses a single HTTP request, it requires additional processing steps, such as query parsing, schema validation, and resolver execution. REST-R1 and REST-R2, in contrast,

map the request to predefined endpoints and return a fixed response structure.

Therefore, for single-entity retrieval, the REST-based strategies achieved the lowest median response times among the evaluated alternatives. This result is consistent with previous studies that reported lower response times for REST in simple retrieval operations [5]. From an architectural perspective, the result suggests that GraphQL's flexible query model may not compensate for its processing overhead when the access pattern is stable and already well represented by a REST endpoint.

3.2 RQ2 – Performance in Multi-Entity Scenarios

RQ2 – How do these Web API design strategies behave when clients retrieve multiple related entities?

RQ2 analyzes how the three Web API design strategies behave when clients retrieve multiple related entities. In these scenarios, the client requests combinations involving **Pokemon**, **Ability**, **Item**, and **PokemonForm**. The main difference among the implementations is how the final response is composed. REST-R1 requires multiple HTTP requests and performs data composition on the client side, REST-R2 returns the related data through a single aggregated endpoint, and API-G retrieves nested data through a single GraphQL query.

Before analyzing response times, it is important to clarify how the number of HTTP requests differs among the evaluated strategies. This distinction is central to our study because the compared strategies differ not only in the API technology adopted, but also in where data composition takes place. REST-R1 performs client-side composition through multiple requests, whereas REST-R2 and API-G retrieve the required data through a single API-level request.

Table 3 uses N to represent the number of **Pokemon** records retrieved in each scenario. In REST-R1, the client first sends one request to retrieve the list of **Pokemon** records. Then, for each **Pokemon**, it sends additional requests to retrieve the related entities required by the scenario. Therefore, the number of HTTP requests in REST-R1 grows with both the number of records and the number of related entities. For example, in the four-entity scenario, the client sends one initial request plus three additional requests per **Pokemon**, resulting in $(1 + 3N)$ HTTP requests. In contrast, REST-R2 and API-G require only one API-level request in all scenarios, because data composition is performed on the server side, either through a predefined aggregated endpoint or through a GraphQL query.

Table 4 reports the median response times for all multi-entity scenarios, while Figure 1 illustrates the behavior in the largest evaluated dataset. The results reveal a substantial performance degradation in REST-R1 as the number of related entities and records increases. For example, when retrieving four related entities for 1000 **Pokemon** records, REST-R1 reached 63,480 ms, whereas REST-R2 and API-G required 1,773 ms and 1,886 ms, respectively.

Across the multi-entity scenarios, API-G was approximately 21 to 34 times faster than REST-R1, with an average

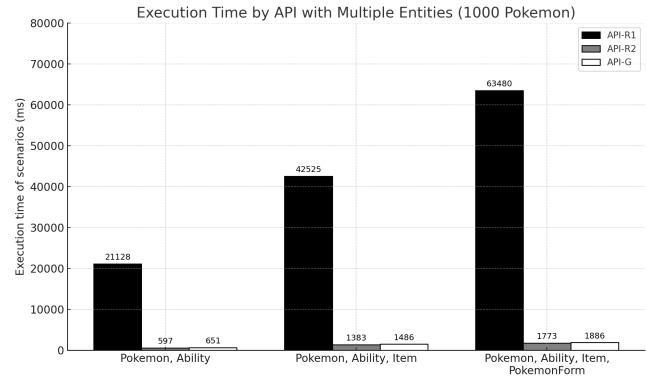


Figure 1: Execution time with 1000 Pokemon and multiple entities.

speedup of about 28 times. In contrast, API-G was only about 1.11 times slower than REST-R2 on average. These ratios indicate that, in the evaluated scenarios, the dominant observed performance factor was the cost of repeated HTTP interactions when data composition was delegated to the client.

These findings show that the main distinction is not simply between REST and GraphQL, but between data composition strategies. Resource-oriented REST transfers composition responsibility to the client, aggregated REST avoids request chaining through predefined endpoints, and GraphQL supports client-defined nested queries through a single endpoint. Thus, GraphQL is particularly advantageous when compared with resource-oriented REST APIs that require multiple sequential requests, while aggregated REST remains the most efficient evaluated strategy when access patterns are known in advance.

3.3 RQ3 – Impact of Request Volume and Data Growth

RQ3 – How does data volume affect the response-time growth of each API design strategy?

RQ3 investigates how data volume affects the response-time growth of each API design strategy. Figure 2 illustrates the normalized growth of execution time in the largest evaluated dataset, with 1000 **Pokemon** records. The same trend was observed for the smaller datasets.

The results show that all API designs are affected by data growth, but the magnitude of this effect differs substantially. REST-R1 presents the steepest growth because the number of HTTP requests increases as the client retrieves related entities. In contrast, REST-R2 and API-G keep the number of API-level requests constant, since both strategies retrieve the required data through a single request.

This behavior can be explained by the interaction between API-level requests, transferred data, and database access. REST-R2 and API-G keep the number of API-level requests constant, whereas REST-R1 increases the number of requests

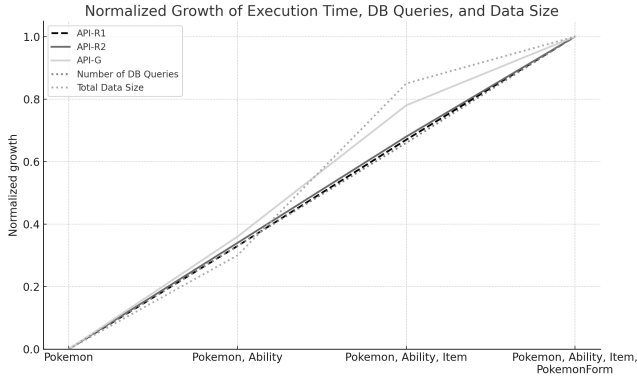
Table 3: Number of HTTP requests required by each API design strategy.

Scenario	REST-R1	REST-R2	API-G
Pokemon	(1)	(1)	(1)
Pokemon + Ability	(1 + N)	(1)	(1)
Pokemon + Ability + Item	(1 + 2N)	(1)	(1)
Pokemon + Ability + Item + PokemonForm	(1 + 3N)	(1)	(1)

Note: (N) denotes the number of **Pokemon** records retrieved. The initial request retrieves the **Pokemon** list, while each related entity requires one additional request per **Pokemon** in REST-R1.

Table 4: Median response time (ms) for different APIs and number of entities.

Entities	API	100 Pokemon	500 Pokemon	1000 Pokemon
Pokemon + Ability	REST-R1	2210	10595	21128
Pokemon + Ability	REST-R2	95	310	597
Pokemon + Ability	API-G	104	347	651
Pokemon + Ability + Item	REST-R1	4299	21285	42525
Pokemon + Ability + Item	REST-R2	172	725	1383
Pokemon + Ability + Item	API-G	202	802	1486
Pokemon + Ability + Item + Form	REST-R1	6392	31766	63480
Pokemon + Ability + Item + Form	REST-R2	212	921	1773
Pokemon + Ability + Item + Form	API-G	245	1004	1886

**Figure 2: Execution time growth with 1000 Pokemon.**

as more related entities are retrieved. As a result, even if each individual endpoint is simple, the cumulative cost of multiple HTTP calls leads to substantially higher response times.

Backend data access may also affect response-time growth. In particular, ORM-based implementations may suffer from $N+1$ query behavior when related entities are resolved individually. Since we did not instrument the persistence layer to count SQL statements, this effect should be interpreted as a possible explanatory factor rather than as a directly measured result. Therefore, reducing HTTP requests is necessary but not sufficient: single-request API designs should still be combined with backend optimizations such as join fetching, batching, data loaders, caching, and database indexing.

3.4 RQ4 – Architectural Trade-offs for Public Web APIs

RQ4 – What architectural trade-offs emerge from these strategies in the design of public Web APIs?

The results suggest that there is no universally superior API design strategy. Resource-oriented REST provides a simple and standardized design, but may transfer data composition responsibility to clients, increasing latency and client-side orchestration when related data are required. Aggregated REST avoids request chaining and achieved the best response-time performance in most evaluated scenarios, but requires the API provider to anticipate access patterns and maintain predefined endpoints. GraphQL offers a flexible query model that supports heterogeneous clients through nested queries, but introduces server-side processing overhead and requires concerns such as query validation, resolver optimization, and query complexity control.

Therefore, Web API design should consider not only response time, but also the expected variability of client needs, the cost of client-side composition, and the maintenance effort associated with predefined aggregated endpoints. The implications regarding maintainability, flexibility, and API evolution are design-oriented analytical insights derived from the architectural characteristics of each strategy, rather than direct empirical measurements of these attributes.

4 RELATED WORK

Several studies have investigated the differences between REST and GraphQL regarding response time, resource consumption, data transfer, and developer experience. Most of these works compare REST and GraphQL as broad API

alternatives. In contrast, our study distinguishes Web API design strategies according to where data composition takes place: on the client side, in predefined server-side endpoints, or in client-defined server-side queries.

Marchuk et al. [5] compared different data access approaches, including direct database access, ORM, REST, and GraphQL. Their results showed that GraphQL presented higher execution time than REST in simple data retrieval scenarios. Similarly, Wahono et al. [12] evaluated database access methods in containerized applications and observed that GraphQL showed lower performance than REST and direct access alternatives. These studies provide evidence that GraphQL may introduce additional overhead in simple and predictable operations, which is consistent with our findings for single-entity retrieval scenarios. However, their evaluations focus mainly on simple data access scenarios and do not analyze how different REST endpoint structures affect multi-entity data retrieval.

Vadlamani et al. [10] presented a broader comparison between REST and GraphQL by combining quantitative experiments and qualitative feedback from developers. Their results showed that the relative performance of REST and GraphQL depends on the operation being executed: REST performed better in some cases, while GraphQL was more efficient in others. The authors also reported that developers considered REST easier to learn and maintain, whereas GraphQL was perceived as promising due to its flexibility and efficient data usage. This evidence reinforces that the comparison between REST and GraphQL should not be reduced to raw performance alone, but should also consider flexibility, maintainability, developer effort, and API evolution.

Other studies have analyzed REST and GraphQL in distributed and microservice-based environments. Vohra and Manuaba [11] compared REST and GraphQL in a microservice architecture using a gateway-based implementation. Their results indicated that REST achieved faster average response times during read operations, especially under higher concurrency. Niswar et al. [7] extended this type of comparison by including gRPC, REST, and GraphQL in microservice communication scenarios. Their results showed that GraphQL presented higher latency and CPU usage under multiple service hops. These studies highlight that communication patterns, architectural context, runtime conditions, and service composition can significantly affect API performance.

Ala-Laurinaho et al. [1] investigated REST and GraphQL as interfaces for industrial OPC UA communication. Their results showed that REST was faster for simple single-value operations, while GraphQL performed better when retrieving bulk data involving multiple values. This result is particularly relevant to our study because it suggests that GraphQL may become advantageous when clients need to retrieve structured or aggregated data rather than isolated resources.

Overall, existing studies indicate that REST tends to perform well in simple, stable, and isolated operations, whereas GraphQL may be beneficial when clients require flexible, nested, or aggregated data retrieval. However, most previous evaluations frame the comparison as REST versus

GraphQL without explicitly distinguishing resource-oriented REST from aggregated REST. This distinction is important because REST APIs may expose one endpoint per resource, requiring clients to compose related data through multiple requests, or may provide aggregated endpoints tailored to specific access patterns. Therefore, observed performance differences may be caused not only by the API technology adopted, but also by the placement of data composition responsibilities.

From the perspective of this study, a remaining gap in prior work is that performance differences are often attributed to REST or GraphQL as technologies, without explicitly isolating the role of endpoint structure and data composition placement. This paper addresses this gap by evaluating three Web API design strategies under the same experimental setting: resource-oriented REST, aggregated REST, and GraphQL. Instead of treating REST and GraphQL as monolithic alternatives, we analyze how client-side request chaining, predefined server-side aggregation, and client-defined nested queries affect response time in multi-entity retrieval scenarios. Thus, our contribution is not only a performance comparison between REST and GraphQL, but an empirical analysis of how the placement of data composition responsibilities affects public Web API design.

5 THREATS TO VALIDITY

This section discusses the main threats to the validity of our study, following Wohlin et al. [13].

Internal Validity. Response-time measurements may have been influenced by implementation-specific effects, such as framework overhead, serialization, container execution, garbage collection, JVM behavior, and operating system scheduling. To mitigate this risk, all APIs were implemented using the same programming language, framework, database system, schema, data volume, persistence strategy, deployment mechanism, and execution environment.

Response equivalence is another concern. Since GraphQL allows clients to select fields, differences in the amount of retrieved data could favor or penalize one strategy. To ensure comparability, GraphQL queries retrieved the same fields returned by the corresponding REST responses. Consequently, this study does not explore GraphQL's potential to reduce payload size through selective field retrieval.

Database access patterns may also affect the results. We did not apply optimizations such as data loaders, join fetching, caching, additional database indexing, or query batching. Thus, the results represent baseline non-optimized implementations rather than fully optimized production APIs, consistent with our goal of comparing API design strategies under equivalent implementation assumptions.

The REST variants also require careful interpretation. REST-R1 represents a resource-oriented strategy in which each entity is exposed through a separate endpoint and composition is delegated to the client. REST-R2 represents an aggregated REST strategy with a predefined endpoint for a known access pattern. Evaluating both variants allows us

to distinguish the cost of HTTP request chaining from the processing overhead associated with GraphQL.

External Validity. The dataset was inspired by PokeAPI and may not represent all public Web API domains. However, its structure of a central entity connected to multiple related entities is also found in e-commerce platforms, social networks, content management systems, media platforms, dashboards, and third-party integrations. The evaluated data volumes were limited to 100, 500, and 1000 *Pokemon* records, and the experiments were executed sequentially and locally. Thus, the results do not capture very large datasets, concurrent access, distributed deployments, production network variability, or high-load conditions and should be interpreted as controlled evidence about response-time behavior rather than production-level performance claims.

Construct Validity. Client-perceived response time was used as the main metric because it directly reflects the latency experienced by API consumers. The measurement includes request transmission, server-side processing, database access, serialization, and response transfer within the controlled environment.

However, API design involves other quality attributes that were not directly measured, including CPU usage, memory consumption, payload size, throughput, bandwidth usage, maintainability, evolvability, and developer effort. Therefore, conclusions regarding maintainability, flexibility, and API evolution should be interpreted as design-oriented insights rather than direct empirical measurements. Similarly, because concurrent workloads and throughput under load were not evaluated, the study analyzes response-time growth across data volumes and entity complexity rather than providing a complete scalability evaluation.

Conclusion Validity. To reduce random variation, each scenario was executed 100 times with a fixed interval between executions, and the median was used as the main descriptive statistic. Since the Shapiro-Wilk test indicated non-normal distributions, the Mann-Whitney U test was used as a complementary analysis. Execution-time ratios were also considered to support the practical interpretation of differences among the evaluated strategies.

Despite these precautions, the findings should not be interpreted as universal claims that one API technology or architectural style is always superior. The conclusions are limited to the experimental context of this study: three API design strategies, one relational data model, controlled local execution, sequential requests, three data volumes, four levels of entity complexity, and non-optimized implementations sharing the same technology stack. Future work should extend the evaluation with optimized implementations, concurrent workloads, larger datasets, distributed deployments, and additional performance metrics.

6 CONCLUSION

This paper investigated how different Web API design strategies affect client-perceived response time under multi-entity access patterns. Instead of treating the comparison only as

REST versus GraphQL, we evaluated three designs: REST-R1, a resource-oriented REST API; REST-R2, an aggregated REST API; and API-G, a GraphQL-based API for nested data retrieval.

The results indicate that the REST-based strategies achieved lower median response times than GraphQL in single-entity retrieval scenarios, where all alternatives required only one HTTP request. In multi-entity scenarios, however, REST-R1 presented the highest response times because clients had to perform multiple sequential requests to compose related data, whereas GraphQL reduced this cost by retrieving nested data through a single request.

REST-R2 achieved the lowest median response times in most evaluated multi-entity scenarios when the required access pattern was implemented as an aggregated endpoint. This result highlights an important design trade-off: aggregated REST can be highly efficient when data retrieval needs are known in advance, but it may require additional endpoint design and maintenance effort as new client requirements emerge. GraphQL, on the other hand, offers a flexible query model for heterogeneous and evolving clients, but may introduce additional server-side processing overhead.

Overall, our findings indicate that there is no universally superior API design strategy. API response-time behavior depends not only on the architectural style adopted, but also on where and how data composition is performed. Therefore, API designers should consider the expected variability of client needs, the cost of client-side composition, and the maintenance effort associated with predefined aggregated endpoints. The implications regarding maintainability and evolvability should be interpreted as design-oriented analytical insights, rather than as direct empirical measurements of these attributes.

As future work, we plan to evaluate optimized implementations using techniques such as data loaders, join fetching, query batching, caching, and database indexing. We also intend to include additional metrics, such as CPU usage, memory consumption, throughput, payload size, and bandwidth usage, as well as concurrent workloads, distributed deployments, and GraphQL selective field retrieval.

7 ETHICAL CONSIDERATIONS

This study did not involve human participants or personal, sensitive, or identifiable data. The experiments consisted exclusively of automated performance measurements using a dataset based on the publicly available PokeAPI. Therefore, CEP/CONEP review and a CAAE number were not applicable under Brazilian Resolution CNS No. 510/2016.

ACKNOWLEDGMENTS

The authors acknowledge the use of generative AI tools limited to language revision and clarity enhancement. The authors retained full editorial control and assume complete responsibility for all findings, results, and conclusions. This research was funded by FAPEMIG under grant number APQ-00763-24.

REFERENCES

- [1] Riku Ala-Laurinaho, Joel Mattila, Juuso Autiosalo, Jani Hietala, Heikki Laaki, and Kari Tammi. 2022. Comparison of REST and GraphQL Interfaces for OPC UA. *Computers* 11, 5 (2022). doi:10.3390/computers11050065
- [2] Alan Cha, Erik Wittern, Guillaume Baudart, James C. Davis, Louis Mandel, and Jim A. Laredo. 2020. A principled approach to GraphQL query cost analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) (*ESEC/FSE 2020*). Association for Computing Machinery, New York, NY, USA, 257–268. doi:10.1145/3368089.3409670
- [3] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine, CA.
- [4] GraphQL Foundation. 2025. Introduction to GraphQL. <https://graphql.org/learn/>. Accessed: 2025-07-03.
- [5] Yaroslav Marchuk, Ivan Dyyak, and Ihor Makar. 2023. Performance Analysis of Database Access: Comparison of Direct Connection, ORM, REST API and GraphQL Approaches. In *2023 IEEE 13th International Conference on Electronics and Information Technologies (ELIT)*. 174–176. doi:10.1109/ELIT61488.2023.10310748
- [6] Sam Newman. 2015. Backends for Frontends. <https://samnewman.io/patterns/architectural/bff/>. Accessed: 2026-05-27.
- [7] Muhammad Niswar, Reza Arisandy Safruddin, Anugrayani Bustamin, and Iqra Aswad. 2024. Performance evaluation of microservices communication with REST, GraphQL, and gRPC. *International Journal of Electronics and Telecommunication* 70, 2 (2024), 429–436.
- [8] Red Hat. 2022. What is an API? <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>. Accessed: 2025-07-03.
- [9] Carlos Rodríguez, Marcos Báez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percanella. 2016. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *Web Engineering: 16th International Conference, ICWE 2016*. Springer, 21–39. doi:10.1007/978-3-319-38791-8_2
- [10] Sri Lakshmi Vadlamani, Benjamin Emdon, Joshua Arts, and Olga Baysal. 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)*. 10–17. doi:10.1109/SER-IP52554.2021.00009
- [11] Naman Vohra and Ida Bagus Kerthyayana Manuaba. 2022. Implementation of REST API vs GraphQL in Microservice Architecture. In *2022 International Conference on Information Management and Technology (ICIMTech)*. 45–50. doi:10.1109/ICIMTech55957.2022.9915098
- [12] Bari Hade Variant Wahono, Asfihani, and Umi Laili Yuhana. 2024. Comparative Analysis of Database Access Method Performance on Containerized Application. In *2024 IEEE International Conference on Artificial Intelligence and Mechatronics Systems (AIMS)*. 1–6. doi:10.1109/AIMS61812.2024.10513113
- [13] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2024. *Experimentation in Software Engineering* (2 ed.). Springer.