Comparison of Rest vs GraphQL: Performance, Authentication and Scalability

Khushnuma Sadaf, Sidra Aslam Khan, Deepanshi Chourasia, Neelanjana Rai and Maria Jamal
Department of Electronics and Communication Engineering, Indira Gandhi Delhi Technical University for Women, James Church, New Church Rd, Opp. St, Kashmere Gate, South Delhi, Delhi - 110006, New Delhi, India

Keywords: REST API(RA), GraphQL API(GA), Query Optimization (QO), Over-Fetching, Under-Fetching (of, UF), OAuth 2.0, JSON Web Tokens (JWT).

Abstract: As the need for smooth data transfer between back and front-end applications grows within web and mobile platforms, choosing the proper API architecture has become a matter of critical importance for developers. APIs have evolved from SOAP services to REST design, and more recently, GraphQL solutions. These models come with their own strengths and weaknesses regarding performance, scalability, and security. The Researchers study a comparative analysis of REST and GraphQL highlighting key emphasis metrics like data-fetching efficiency, bandwidth consumption, query complexity, response time, and security concerns. We also consider authentication mechanisms, common security vulnerabilities, and the effect of rate limiting and caching strategies on API performance. This research attempts to provide practical guides for selecting the most suitable API paradigm, depending on particular requirements of the use-case in the form of data-driven practical analysis and a set of real-world use-case studies. Conclusions: REST is a solid foundation that is built with consideration of the specs of caching, while GraphQL allows for a more flexible querying capability, and can eliminate redundant clauses to reduce bandwidth. The research behind this paper provides useful directions for both developers and architects when it comes to deciding how APIs are implemented.

1 INTRODUCTION

The APIs are regarded as the fundamental components for web and mobile apps since it allows the system clients and servers to communicate easily in today's digital environment. One of the key factors influencing the performance, scalability and security of the entire application is an API efficiency. As the need for data-heavy and real-time applications has increased, API design has adapted to the challenges around data retrieval, response-time and security.

One of the first communication mechanism used in the early days of web services is a strict XML-based messaging protocol called SOAP (Simple Object Access Protocol), providing a reliable way to transmit data, but with a lot of overhead and complexity. Due to those limitations, Representational State Transfer (REST) has emerged as a simpler and more flexible alternative. REST's lightweight, stateless architecture described the industry standard, handling resources through HTTP methods such as GET, POST, PUT and DELETE. REST-style APIs are often criticized for their concept of over-fetching and under-fetching data as such they require multiple requests to gather related information which is not only hurtful to performance in data-heavy applications, but it also couples' clients with the API endpoints resulting in the need for multiple sets of endpoints on the server.

Well to fight with such in-efficiencies, GraphQL was introduced in 2015 and a query based flexible way of API designing came into existence. REST depends on endpoints that return static payloads, while in Graph QL the clients define what fields of data they need and only the requested data is returned, reducing used bandwidth and improving the speed of data handing. While GraphQL provides the ability to fetch dynamic data, it comes with its complexities in terms of having a greater query processing overhead (for parsing, validation, and execution), caching complexities.

These architectural differences can often force the developer to choose, so they often have to decide which of the two API models would fit their application best. We provide a deep dive into REST

Sadaf, K., Khan, S. A., Chourasia, D., Rai, N. and Jamal, M.
Comparison of Rest vs Graphql: Performance, Authentication and Scalability.
DOI: 10.5220/0013930100004919

In *Proceedings of the 1st International Conference on Research and Development in Information, Communication, and Computing Technologies (ICRDICCT'25 2025) - Volume 5*, pages 369-377
ISBN: 978-989-758-777-1

and GraphQL and analyse their performance based on data-fetching efficiency, response size, query complexity, security mechanisms, rate limiting, caching and scalability. We are looking for example applications and performance metrics so we can provide tangible knowledge that will help software developers decide the optimal API design for their application.

The rest of the paper is organized as follows: in Section 2, we introduce the history of API. In Section 3 we examine architectures, and highlight key technologies. Section 4 compares REST and GraphQL on their performance metrics axis e.g how well they retrieve data, bandwidth metrics and response times. In section 5, we discuss authentication strategies and common security mistakes. 1.6 Rate limiting, caching, and scaling considerations the results and discussion from implementation is presented in Section 7. 5. Similar is Section 8 with a summary of findings regarding differences in models and recommendations for selecting the one that is most suitable for your application.

2 LITERATURE REVIEW

APIs are the true backbone of modern web development, enabling disparate systems to interoperate. API architectures have progressed from traditional SOAP web services to REST APIs and further enrich the architecture, adding GraphQL. However, with the increasing requirement for scale, flexibility, and security, developers have been looking for new API paradigms to improve performance and do better at exchanging data.

The SOA architecture used during the first few generations of web service communication was Simple Object Access Protocol (SOAP), which enabled structured information to be exchanged between web services through XML messaging in a well-defined but restrictive way. While SOAP APIs were incredibly powerful and useful, they came under fire for their more complex syntax and verboseness, and REST (Representational State Transfer) emerged as a simpler and more flexible, API standard. Given these pros, RESTful APIs became popular as it was stateless, easily scalable, and uses standard HTTP methods. The figure 1 shows the Architecture of REST.

But the advantages of REST come with some disadvantages as well: data over-fetching, under-fetching, and excessive network calls have led to an increased interest in GraphQL. GraphQL is more of a query language where the client can ask for exactly what it requires, thereby consuming a lot less bandwidth and improving API performance. Transitioning from fixed resource-based endpoints with REST to a flexible, Schema-first methodology present in GraphQL has represented a massive leap-forward in the field of API design

3 ARCHITECTURE

3.1 REST Architecture

REST (or Representational State Transfer) is an architectural style used for constructing APIs that communicate using the HTTP protocol. Although commonly mistaken for a protocol, REST is really an architectural style (or, more appropriately, a set of constraints) that specifies how resources should be represented and manipulated in a distributed system. RESTful APIs operate in a stateless client-server environment, facilitating interaction between the front-end and back-end by sending HTTP requests and receiving HTTP responses, emphasizing scalability, reliability, and simplicity and are more widely used for application development which is your case.

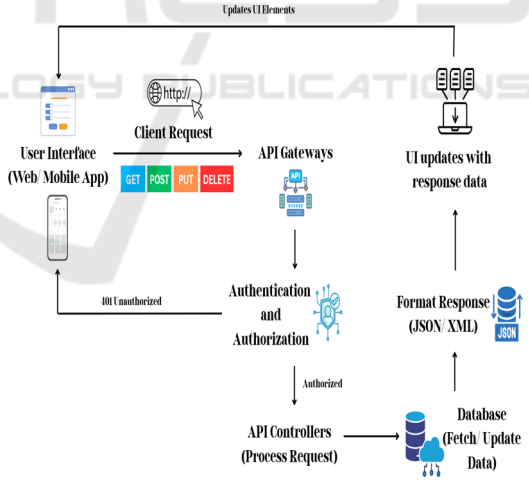

Figure 1: Architecture of REST.

Key Principles of REST Architecture

- REST APIs are stateless, which means that every request from the client must contain all the data required to fulfil that request, as the server does not retain information about previous requests of the client. This increases scalability as client API calls can be served in parallel and do not require persistence of session.

- REST architecture is resource-based, which means data entities are changed to resources, like users, products & orders, where each resource has a unique URI (uniform resource identifier). These resources are actuated using common HTTP methods e.g. GET, POST, PUT, DELETE.
- REST APIs follow a standard interface, allowing a streamlined communication between clients and servers. Designed to a specification which enables inter-operation across a wide set of platform and programming language- a request-response is as structured as possible.
- REST APIs also use caching mechanisms, which are used to increase performance by storing data. HTTP headers like Cache-Control and ETag allow the caching of requests on multiple levels, preventing unnecessary calls to your database and speeding up response times.
- REST APIs authenticate and authorize access using Oauth 2.0, JWT (JSON Web Tokens), and API key-based methods. They are used to limit access to sensitive resources and to facilitate secure interactions between clients and servers when verifying user identity.
- REST APIs are good for microservices-based architecture, where different microservices expose their individual REST APIs. This modular architecture allows services to be scaled up individually and deployed in cloud environments with ease.

Rate limiting and throttlingExample in REST APIsThe single user (client) abuse is restricted in recommended REST APIs using rate limiting and throttling. They enhance security, keep systems stable, and ensure maximum performance in high traffic situations by limiting requests on a per user or IP address basis.

3.2 GraphQL Architecture

GraphQL: A query language for your API and a server-side runtime that accepts queries and returns fast results based on the specific data type definition. While GraphQL was created by Facebook in 2012 and publicly released in 2015, it's a comparatively a newer technology that provides a flexible, efficient, and scalable alternative to REST APIs. Unlike REST APIs, where clients have to pull data from hard-coded endpoints, GraphQL grants clients the power to create their own queries in order to receive exactly what they need in a structured response.the figure 2 shows Architecture of GraphQL.

Key Principles of GraphQL Architecture

- Whereas REST uses multiple endpoints to access different resources, GraphQL works through a single endpoint. This design allows clients to retrieve all the necessary data in a single request and avoids making multiple requests to different endpoints.

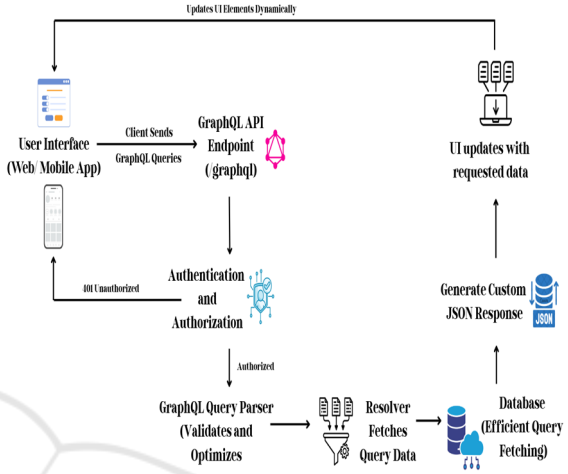

Figure 2: Architecture of GraphQL.

- Declarative data fetching With GraphQL, clients have the capability to describe which data they need in a query. Using this method, allows for no over-fetching, where it retrieves unnecessary information, and no under-fetching, where multiple requests must be performed in order to obtain related data.
- A strongly typed schema describes the data that can be fetched through a type system with your GraphQL server, including types, queries, and mutations. This schema-driven approach enables better data validation, consistency, and self-documentation, making it easier to understand and use the API.
- One of the cores idea behind GraphQL is pulling multiple nested or relational data in a single query and reducing the network latencies. GraphQL collects related resources into a single execution and allows for query batching, unlike make separate API calls to get those relationships.
- GraphQL also has more structured error handling than REST, as error messages are included in API responses. Unlike REST APIs that use HTTP status codes to identify errors, GraphQL provides detailed error objects, including the error itself, its location, and

possible solutions, enhancing our debugging capabilities.

- GraphQL has built-in protections against abuse, with features like query complexity control to limit depth and analyze costs. Such mechanisms can limit very deep or expensive queries, thus helping to prevent resource over-consumption, and virtual denial-of-service attacks.
- GraphQL's integration with microservices and federated architectures is natural, with each microservice able to define and expose its own, separate schema and then being combined into a unified API in so-called "API composition." This improves the scalability of distributed applications while also simplifying integration of heterogeneous data sources.

4 PERFORMANCE

Performance evaluation is an important part of any API development to ensure scalability, responsiveness, and resource utilization. In this section, the focus will be on key performance metrics such as efficiency, bandwidth use, and response time to analyze how effective REST and GraphQL are for communicating workloads. We hope to learn more about the trade-offs between the two methods for real-world use by measuring the previous variables.

4.1 Efficiency

Data efficiency is the measure of the amount of data fetched (in bytes or number of records) in a certain amount of time. So, the formula for efficiency can be like this:

$$Efficiency = Amount\ of\ Data\ Fetched \div Response\ Time \quad (1)$$

The efficiency hence can vary with change in the amount of data fetched and the time taken to fetch that data. The two methods REST and GraphQL both have different efficiency due to their difference in their way of working. While REST uses the approach of multiple endpoints to fetch the data, the GraphQL method uses the query approach to access the data.

So firstly, focusing on response time, REST has a relatively faster response time than that of GraphQL due to the complexities in the query of GraphQL, which can take higher time to process. REST fetches data from the database when requested. GraphQL also fetches data when requested, but before querying the database, it first resolves the query to determine what specific fields are needed, fetching only the required data. This processing of query in the middle of request and response cycle leads to increase in the time of response for GraphQL APIs.

Secondly, since REST fetches all the data from the database and GraphQL fetch only what is queried, the amount of data fetched is also different. Hence REST fetches way more data than GraphQL.

This relatively faster and more data fetching nature of REST results in REST having a higher efficiency that the GraphQL API methods.

4.2 Bandwidth

Bandwidth refers to the maximum rate at which data can be transferred over a network connection in a given amount of time.It measures the capacity of a channel for data transferring. The higher the bandwidth the higher the capacity of the channel to transfer large information at a particular instance of time.

$$Bandwidth\ (bps)\ \ Total\ Data\ Transferred\ (bits) \div Time\ (seconds) \quad (2)$$

REST relies on endpoints to request the data, which can also return unnecessary data and sometimes it may require multiple endpoints to collect different data fields which along with them bring unrequired data as well. This multiple endpoint requesting and collection of unnecessary data results in REST having a requirement of higher bandwidth then GraphQL.

GraphQL on the other hand has a structured query response requesting and fetching only the data fields which are necessary or required. This makes the Bandwidth of GraphQL much more optimized than that of REST, as unlike REST it doesn't over-fetch or under-fetch the data.

4.3 Query Complexity and Response Time

Query complexity refers to the computational cost of processing a query, influenced by factors like depth, the number of requested fields and objects, and the use of filters or aggregations. Higher complexity increases the server's processing load, making queries slower and more resource-intensive.

Response time is the total time taken from sending a request to receiving a response. It depends on query complexity, database efficiency, network latency, and server performance. Optimizing queries and reducing

unnecessary data retrieval can help improve response time.

$$Response\ Time = Time\ of\ Data\ Receiving - Time\ of\ API\ Request\ Sent \quad (3)$$

In simple use cases, REST APIs generally have lower response times than GraphQL because the execution of REST is straightforward with established endpoints which return static data. This minimizes processing overhead, as the server has no need to parse complex queries, nor dynamically deduce response structures. Additional benefits of RESTful architecture include caching on multiple levels (browser, CDN, and server-side), further reducing the amount of response time for repeatedly fetched data.

On the other hand, GraphQL lets clients request only the data that they require, mitigating issues with over-fetching. This flexibility comes with an additional computational cost, however. This also means the server has to dynamically process and resolve every field requested by a query which can slow down response time — particularly for deeply nested queries or when you're resolving relationships across multiple database tables. Moreover, GraphQL queries are often complex demands for multiple resolvers, which make them take longer to process compared to the single endpoint responses from Rest.

5 AUTHENTICATIONS

Authentication is the primary security mechanism of confirming the identity of users and applications accessing resources. REST and GraphQL also support common authentication methods like API keys, Basic Authentication, OAuth 2.0, and JSON Web Tokens (JWT), but their implementation is different.

Authentication is normally applied per endpoint in RESTful APIs. API Keys and Basic Authentication are easily implemented but with no advanced security. Moreover, these approaches explicitly expose a secret that needs to be sent in every request, which can be detrimental if this secret is not encrypted. The more secure alternatives are OAuth 2.0, which lets third-party applications access resources securely, and JWT which provides stateless authentication by putting userclaims (information about the user) directly in the token.

On the other hand, GraphQL authentication works at the resolver level instead of per endpoint because GraphQL is exposed through one entry-point. This enables developers to manage access more broadly, all the way down to the field level of a non-Basic ");}, But with that only increasing complexity as we would now have to enforce authentication logic within each resolver function.

All in all, REST combines simple authentication approaches with lots of support, whereas GraphQL allows more fine-grained control but can be difficult to integrate securely.

5.1 Common Security Risks

APIs come with many security vulnerabilities, but the nature of these vulnerabilities is not the same between REST and GraphQL because of their architectural differences.

- Injection Attacks: If user inputs are not sanitized correctly, REST and GraphQL applications are susceptible to SQL injection, NoSQL injection, and command injection. As REST has multiple endpoints, all of them could be a surface of attack. On the other hand, the dynamic query approach of GraphQL necessitates extensive input validation in resolvers to prevent malicious queries.
- Broken Authentication: REST APIs can suffer from misconfigured authentication on several endpoints, while with GraphQL the single endpoint presents a security risk if not adequately secured. Furthermore, GraphQL's introspection feature can reveal sensitive schema details if it is left enabled in production.
- Excessive Data Exposure: REST can return more data than what is needed (e.g. all fields), whereas GraphQL by definition can query only what is needed - assuming their access controls are well enforced.
- Denial-of-Service (DoS) Attacks: REST can be targeted through brute-force endpoint requests while GraphQL can be targeted using deeply nested or recursive queries, both of which can consume a lot of server resources. To protect against abuse, GraphQL APIs require defenses such as limiting query depth and analyzing complexity.

5.2 Rate Limiting and Caching

To avoid this storm and have a better performance in the API by preventing Abuse, rate limiting, and caching is a necessary process.

In REST, rate limiting is trivial because the requests flow to a specific endpoint. Commonly used by API gateways and proxy tools, traffic monitors use IP, API keys, or user accounts for abuse detection and

set up different rate limits based on fixed-window or token-bucket algorithms.

For GraphQL this can be a little less straightforward. Because all requests pass through a single endpoint, standard request-based limits don't work. Alternatively, GraphQL relies on a query cost analysis, assigning a "weight" to each query according to the complexity. By limiting the depth of queries the server will process, query depth limiting protects the server from excessive nesting of queries.

REST is generally simpler to cache — which makes it well-suited for the storage of frequently-accessed data — since you can easily use built-in HTTP caching mechanisms such as Cache-Control as well as the ETag and Last-Modified headers. GraphQL's flexible query will make you more difficult to cache, the server response will be different depending on the fields requested. Instead GraphQL caching works on a client-side (like Apollo Client), persisted queries and response normalization.

Well-known and simpler solutions for authentication, caching and rate limiting is already providing by REST. Correctly implemented, GraphQL can be more flexible and performant, albeit makes it more challenging to optimize for security and performance.

6 SCALABILITY CONSIDERATIONS

Scalability is a critical factor in modern API design, ensuring that a system can handle increasing workloads without significant performance degradation. This section explores key aspects of scalability, including managing high traffic loads, optimizing database queries, and ensuring flexibility in API evolution. By analysing these factors, we aim to assess how REST and GraphQL adapt to growing demands and evolving application requirements.

6.1 Handling High Traffic Loads

In an era where digital applications are scaling at an unprecedented rate, APIs are expected to manage high traffic volumes without a decline in performance. REST APIs are stateless, which means it is inherently load-balanced and caches requests, making them suitable to handle a surge of requests from users. But with more distributed services depending on REST APIs, the design of API gateway solutions and rate-limiting solutions need more improvement to speed performance in large-scale applications.

In stark contrast, GraphQL allows you to consolidate multiple requests into a single query, flowing less network overhead and performing effectively for high-demand applications. Nonetheless, this benefit has a processing cost, as complex queries can put a high load on servers. future work can revisit considerations of how to manage consumed resources through either cost analysis of the overall graph and therefore on query complexity automatically, sentence rate limiting, similar to what is done in rest (aka Oas) regarding route limits, and other measures to ensure that the graph remains scalable under heavy traffic. One hypothesis for an improvement in this case, could be adding caching at the level of query in GraphQL, significantly reducing not only the execution time but also eliminating redundant calculations

6.2 Database Performance and Query Optimization

The interaction of the APIs with the database is one of the most important aspects of performance. Traditionally REST APIs depend on structured SQL databases, and the queries pass through predefined endpoints that return fixed datasets. However, maintaining data integrity in such scenarios leads to redundant queries being fired, which is a bottleneck for the response time especially for complex data relationships. In RESTful architectures, future investigations into AI is also needed, including automated query rewriting and dynamic indexing with an emphasis on evaluating the quality and use of query logs in AI-enabled indexing systems.

GraphQL also has the model of executing a query on-demand with respect to NoSQL databases. On the other hand, too deeply nested queries in the case of GraphQL can lead to expensive database joins to occur for a long query execution time. Predictive caching, which refers to loading frequently accessed data cognizant of user behaviour, can also be explored in future studies to reduce latency experienced by query users. Batched queries would fall into this category, as would deferred execution strategies that optimize database access without requiring all data to be fetched and derived by the server just to satisfy a GraphQL–style resolver.

6.3 Flexibility in API Evolution

To support Lightbend Akka HTTP v10.1.0, we introduce two major additions: Revisions to the API

and an up-gradation to the latest version of Akka HTTP. REST APIs generally use versioning strategies (v1/v2, etc. as part of the URL or headers) to allow changes while preserving backward compatibility. Although this strategy ensures stability, it can result in progressive fragmentation and maintenance problems when the Added several versions of the API exist. Future studies might include automated version migration tools, and self-adapting API frameworks that assure minimum disruption in updates.

The fast development of API is due to GraphQL and it may introduce new fields to the schema and will not affect the existing queries. It allows backwards compatibility; i.e., any changes introduced to the API can be done without breaking existing consumer applications, reducing the need for versioning. This is very challenging though to manage schema complexity as the APIs eventually grow. Future work can explore automated schema refactoring approaches that incrementally removes deprecated fields, and AI-based schema evolution assessment that aims to foresee performance bottlenecks before they occur.

7 RESULTS AND DISCUSSION

7.1 GraphQL Method

The performance evaluation of the GraphQL API was conducted by analyzing key metrics: Bandwidth, Efficiency, and Complexity Time. The results are visualized in Figure 3, showcasing variations over five test instances.

The Bandwidth (indicated by the blue line) seems to be unchanged under all test instances, while the values is high (around 689 units). This means that, no matter how complex your query may become, GraphQL always transfers the same amount of data.

The Efficiency metric (shown as purple) does have a slight amount of variation but most of the time is stable and stays between 60 and 80. The line shows a gradual increase between the 2nd and 3rd cases which implies that under some conditions, we are able to retrieve data more efficiently than the previous example. But the small differences suggest that efficiency can depend on the structure of the query as well as one-time server-side optimisations.

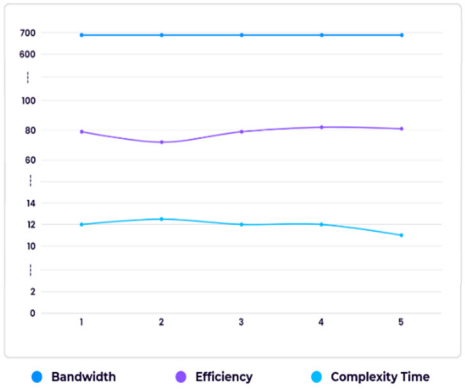

Figure 3: Performance metrices of GraphQL.

The Complexity Time (cyan line) oscillates over classic 10 and 14 of unit, that measures that supervises execution time in the query is slightly variable, but gets manage in the unit acceptable. The slight fluctuations indicate that a GraphQL query has stable complexity and does not have a compounded performance degradation at the tested cases.

In general, the study shows that GraphQL APIs are a fundamentally efficient way for querying and resolving data, due to their remarkably consistent bandwidth usage as well as reasonable bandwidth consumption and overall GraphQL execution times. The findings indicate that query optimization techniques and caching strategies could provide additional performance improvements.

7.2 REST Method

The performance evaluation of the REST API was conducted by analyzing key metrics: Bandwidth, Efficiency, and Complexity Time. The results are presented in Figure 4, illustrating trends across five test instances.

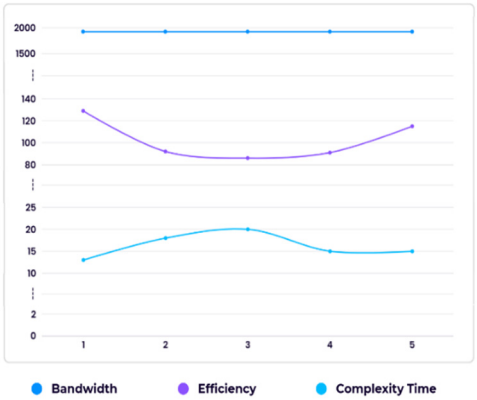

Figure 4: Performance metrics of rest.

Bandwidth (blue line) remains fairly high at ~1927 units for all test instances. This indicates that REST APIs are bandwidth heavy compared to GraphQL, possibly due to over-fetching of data.

The Efficiency metric (purple line) follows a U-trend, it starts at a high value, gets a little down at the second instance, drops very low at the third instance, and then raises up towards the fifth instance. This means REST APIs become inefficient based on how the resource request-response structure works and can suffer in performance while dealing with complex queries.

Complexity Time (cyan line) differs a lot from instance to instance rising from the first instance to the third, then dipping slightly and finally converging. The peak indicates that some queries take longer to process, probably because they require more than one endpoint request because GraphQL gather in one query.

The results of the analysis show that REST APIs are the ones that consume more bandwidth and are less stable in terms of efficiency compared to GraphQL. Since REST essentially needs multiple calls to get associated data, higher complexity time is observed. Such data retrieval is better in use cases where optimized data fetching is needed which concludes that GraphQL can serve as a better alternative for such scenarios.

8 CONCLUSIONS

In this study, we thoroughly analysed the performance, scalability, and authentication mechanisms of the REST and GraphQL APIs based on essential metrics (such as bandwidth usage, throughput, complexity time, and security concerns). Our results show that GraphQL considerably increases data-fetching efficiency, minimizing over-fetching and under-fetching, which in turn reduces the amount of data sent over the wire and decreases the complexity of queries. On the other hand, the REST APIs show more bandwidth and flutter time complexities for the same reasons (i.e., multiple endpoint calls and see the duplicate Transfer of data).

Regarding data scalability, GraphQL has a main advantage over REST, because each REST endpoint answers only one part of the object, and the client has to call many different URLs to fetch data about that object. However, the additional per-query overhead in GraphQL could lead to an increased response time under high concurrency conditions. In contrast, REST APIs have less complexity to deal with, predictable load balancers and more simplified caching mechanisms.

In terms of authentication and security, both API architectures offer support for common mechanisms like OAuth 2.0, JWT, and role-based access control (RBAC). Yet, GraphQL also adds new such as deep query complexity attacks and unauthorized data exposure, so it's crucial to limit query depth and include access control logic. The statelessness of REST allows for more structured enforcement of security policies at each endpoint, allowing stricter authentication to be more easily enforced.

In general, the performance-oriented and dynamic nature of GraphQL makes it superior for use cases where we can reduce bandwidth consumption with higher fetching efficiency but if you need an environment with scalability and security the REST API approach seems to be the best solution. This will include testing these APIs on different network conditions, applying advanced caching mechanisms and analyzing security vulnerabilities.

REFERENCES

A. Johnson, B. Lee, and C. Martinez, "Bandwidth and Latency Considerations in RESTful and GraphQL APIs for Scalable Systems," in Proceedings of the International Conference on Cloud Computing and Big Data (ICCCBD), 2022, pp. 215-223.

A. Gupta, P. Singh, and R. Patel, "Comparative Analysis of Authentication Mechanisms in REST and GraphQL APIs," in Proceedings of the IEEE International Conference on Cybersecurity and Privacy (ICCP), 2022, pp. 350-360.

C. Lewis and D. Zhang, "Reducing Network Overhead in Web Applications: A Performance Analysis of GraphQL," in Proceedings of the International World Wide Web Conference (WWW), 2019, pp. 510-519.

D. K. Smith and E. Miller, "Optimizing API Performance: A Comparative Study of REST and GraphQL Under Varying Load Conditions," in Proceedings of the IEEE International Conference on Web Services (ICWS), 2019, pp. 78-85.

J. Anderson and P. White, "Scalability and Maintainability of API Architectures: REST vs. GraphQL," in Proceedings of the IEEE International Conference on Software Engineering and Knowledge Engineering (SEKE), 2022, pp. 415-423.

K. Wang, J. Luo, and M. Patel, "A Comparative Study on Data Fetching Efficiency in RESTful and GraphQL APIs," in Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2021, pp. 143-152.

L. R. Barros, G. Gousios, and A. Zaidman, "Comparing the Efficiency of GraphQL and REST APIs: An Empirical Study," in Proceedings of the IEEE International

Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 450-461.

P.M. Runeson, J. Malmqvist, and R. Torkar, "Performance Implications of REST vs. GraphQL: A Case Study in API Design," in Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE), 2020, pp. 330-342.

S. Bose, M. Roy, and L. Kim, "Load Balancing and Caching Strategies for Scalable GraphQL APIs," in Proceedings of the IEEE International Conference on Cloud Computing (CLOUD), 2023, pp. 98-107.

T. Nakamura and L. Chen, "A Study of Over-Fetching and Under-Fetching in REST and GraphQL APIs," in Proceedings of the ACM International Conference on Distributed Systems and Web Engineering (ICDSWE), 2021, pp. 180-192.