GraphQL vs. REST: A Performance and Cost Investigation for Serverless Applications

Runjie Jin
University of Washington
Tacoma, Washington, USA
rjjin@uw.edu

Robert Cordingly
University of Washington
Tacoma, Washington, USA
rcording@uw.edu

Dongfang Zhao
University of Washington
Tacoma, Washington, USA
dzhao@uw.edu

Wes Lloyd
University of Washington
Tacoma, Washington, USA
wlloyd@uw.edu

Abstract

Serverless computing simplifies application deployment by removing the need for infrastructure management, with RESTful APIs being the common interface. However, REST can lead to inefficiencies such as data over-fetching and under-fetching, which impact performance and cost. This paper investigates GraphQL as an alternative to REST for serverless functions using a serverless image processing pipeline. We evaluate roundtrip time (RTT), scalability, and cost, while also examining managed (AWS AppSync) and unmanaged (Apollo Server) GraphQL hosting solutions. Our results show that GraphQL generally outperforms REST with respect to pipeline RTT, especially when there is high network latency, offering a potentially better fit for optimizing data transfer in serverless applications.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **General and reference** → **Performance**.

Keywords: Function-as-a-Service, Serverless Computing, Performance, GraphQL

ACM Reference Format:
Runjie Jin, Robert Cordingly, Dongfang Zhao, and Wes Lloyd. 2024. GraphQL vs. REST: A Performance and Cost Investigation for Serverless Applications. In *10th International Workshop on Serverless Computing (WOSC '24), December 2–6, 2024, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3702634.3702956

WOSC '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-1336-1/24/12
https://doi.org/10.1145/3702634.3702956

1 Introduction

Serverless computing has changed how applications are deployed and managed by abstracting the underlying infrastructure. With serverless computing, developers no longer manage servers and scaling since Function-as-a-Service (FaaS) platforms, such as AWS Lambda, automate these tasks. This enables developers to focus primarily on application logic, while the cloud provider adapts server resources on demand. Serverless applications are typically event driven and billed only for the resources consumed during execution, making them cost efficient and highly scalable. Applications provide interfaces to serverless functions using RESTful APIs consisting of standard HTTP methods including GET, PUT, POST, and DELETE to communicate between clients and servers. Representational State Transfer (REST) is widely recognized for its simplicity and ability to abstract web communications, making it a go-to solution for web services. Cloud providers like AWS and Google Cloud offer extensive support for REST APIs with seamless integration to their serverless platforms.

While REST's widespread adoption has made it the default interface for serverless functions, it may not be ideal for all serverless use cases. REST interfaces can lead to inefficient data exchange, such as overfetching (i.e., retrieving unnecessary data) and underfetching (i.e., not retrieving enough data) [12]. These inefficiencies can increase execution time and resource consumption, important considerations in a serverless environment where performance and resource utilization translate to cost. This paper investigates implications of using GraphQL interfaces in contrast to traditional REST interfaces for a serverless image processing pipeline, as a serverless application use case. We investigate GraphQL performance in contrast to REST, as a means to optimize data transfer, latency, and round-trip-time (RTT).

GraphQL is a query language for creating APIs that leverages an execution engine to evaluate queries. It was initially developed by Facebook in 2012 and open-sourced in 2015 [13]. GraphQL offers a dynamic and efficient alternative to REST, enabling clients to request exactly the data they need. Unlike REST, where data is returned in predefined formats,

GraphQL supports fine-grained control over data returned from servers. This makes GraphQL particularly well suited to serverless environments, where minimizing data transfer and session overhead is critical to optimize performance and reduce cost. By combining data from multiple sources into a single request, GraphQL can reduce the number of client-to-server round trips in serverless data processing pipelines, leading to more efficient resource utilization and faster response times [14].

Despite widespread adoption of serverless computing and GraphQL by companies like GitHub [7], and Netflix, only limited research has been conducted to investigate GraphQL performance, scalability, and cost-efficiency for serverless applications. Prior research includes case studies [23, 25], testing frameworks [15, 21], and extensions to GraphQL's query language [19, 20], but the utility of GraphQL as an interface for serverless functions in contrast to REST has not been rigorously evaluated. This paper presents an in-depth comparison of GraphQL and REST interfaces for serverless functions by evaluating their performance and cost for a serverless image processing pipeline. Insights from this research can help practitioners and researchers better understand the trade-offs of GraphQL vs. REST for providing an API interface for serverless applications.

1.1 Research Questions

We will investigate the following research questions:
RQ-1: (*GraphQL API performance*) How well does GraphQL perform in the serverless environment with respect to RTT and cost? What are the performance implications in contrast to providing the same functionality using REST APIs to backend serverless functions?
RQ-2 : (*GraphQL managed vs. unmanaged*) What are the performance and cost differences for hosting GraphQL APIs using an unmanaged self-hosted GraphQL server vs. a managed GraphQL service supported by a commercial cloud provider?

1.2 Contributions

1. Comparison of GraphQL vs. REST for Serverless Functions: We provide a detailed comparison of GraphQL and REST APIs for serverless functions, investigating scalability and performance metrics including RTT and cost.
2. Evaluation of Managed vs. Unmanaged GraphQL Solutions: We investigate the performance and cost of managed (e.g., AWS AppSync) vs. unmanaged self-hosted GraphQL servers to guide developers in choosing between these solutions for serverless applications.

2 Background and Related Work

2.1 GraphQL for APIs

GraphQL is a query language backed by an execution engine to enable clients to request precisely the data they need using a defined schema. A schema specifies data types, structures, and relationships between entities, allowing efficient retrieval from multiple sources like databases, serverless functions, and external APIs using a single query. This approach minimizes over-fetching, improving flexibility to optimize resource usage important in serverless environments.

Central to GraphQL's operation are resolvers, which map query fields to data sources to perform the necessary operations to fetch or compute the requested information. Each field in a query is resolved independently, allowing GraphQL to concurrently pull data from various sources in a single request. This parallel execution is particularly beneficial in applications with complex or nested queries, as it reduces the number of API calls.

GraphQL supports mutations for data modification and subscriptions for real-time updates, making it ideal for dynamic, interactive applications. These features offer flexibility in API design, allowing developers to build responsive serverless applications capable of handling real-time data flows, such as live dashboards or collaborative tools.

2.2 GraphQL Case Studies and Surveys

Prior research has performed qualitative comparisons of GraphQL and REST in use cases and surveys, highlighting advantages and the best adoption scenarios for each approach ([16, 23–25]). While prior efforts offer insights into the strengths and weaknesses of GraphQL and REST, these investigations lack quantitative analysis.

Vadlamani et al. [24] interviewed GitHub employees asking them questions to compare REST vs. GraphQL. They found GraphQL and REST each have their best adoption scenarios. Mohammed et al. [23], Vázquez-Ingelmo et al. [25], and Bryant and Mike [16] utilized GraphQL in individual projects including an API to access medical records, an Observatory of Employment and Employability (OEEU) data API, and an archival metadata API for the European Holocaust Research Infrastructure (EHRI). Mohammed et al. [23] presented a new API developed from scratch with GraphQL, while Vázquez-Ingelmo et al. [25] and Bryant and Mike [16] described API conversions from REST to GraphQL, demonstrating performance improvements.

2.3 GraphQL Performance Analysis

Several research efforts have analyzed the performance of GraphQL APIs ([15, 18, 21, 22]). These efforts describe valuable methodologies and testing frameworks for quantitative analysis. However, these efforts concentrated solely on benchmarking GraphQL performance, and did not contrast performance with equivalent REST APIs or investigate providing interfaces for serverless functions.

Cheng et al. [18] proposed a GraphQL server benchmark, Linköping GraphQL Benchmark (LinGBM), to identify key technical challenges (e.g. choke points) when building efficient GraphQL servers. Their benchmark is more comprehensive than previous GraphQL benchmarks, and uses the choke-point design method for more precise results. Belhadi et al. [15] introduced a new testing framework based on

the open-source EvoMaster tool which executes white and black-box tests and automatically generate JUnit and Jest test cases. Karlsson et al. [21] presented a new method to automatically generate GraphQL queries to test GraphQL APIs and evaluate test coverage. Mavroudeas et al. [22] utilized machine learning to predict the price cost of GraphQL queries, showing that their approach could outperform traditional static analysis.

2.4 GraphQL Platforms

AWS AppSync is a fully managed GraphQL service offered by AWS (Amazon Web Services) [3]. AppSync facilitates the development of scalable and flexible applications by enabling data synchronization across multiple data sources. AppSync is serverless and does not require users to provision or manage virtual machines with fixed vCPU or memory allocations. Using GraphQL's data query abilities, AppSync allows developers to create interactive applications with optimized data retrieval. It also integrates well with other AWS services, providing a secure and scalable environment for complex data manipulation tasks.

Apollo Server is an open source GraphQL server designed to simplify the process of building, deploying, and maintaining GraphQL APIs [2]. It supports integration with various data sources, including relational databases and REST APIs, enabling developers to build efficient APIs. Apollo server supports custom type definitions, resolvers, and directives, enabling tailored creation of GraphQL schemas to meet specific application needs. It is compatible with any GraphQL client, most notably the Apollo client. Apollo Server is regarded for its extensibility and customizability, offering features enabling developers to adapt it to a variety of use cases.

In addition to Apollo Server, there are other open source GraphQL server implementations, including GraphQL.js (the reference implementation) [8], express-graphql (deprecated) [6], and Hasura [9]. These servers emphasize different goals. For example, Hasura [9] reduces developer time and effort by automatically generating GraphQL APIs code, allowing developers to skip writing boilerplate code required to create an API. GraphQL.js is a basic implementation that requires a deeper understanding of the GraphQL specification, making it more challenging for beginners compared to other higher-level frameworks. PostGraphile [11] is an extension to the PostgreSQL database that supports only limited usage scenarios. In this paper, we focus on performance evaluation of Apollo Server as an unmanaged server in contrast to AppSync, a managed GraphQL service.

Table 1. Test Client Configurations

Client	Type	Region	Description
Local	Desktop	Local	32 GB RAM, Intel i5-13600K 20 cores, 350 Mbps Band
EC2	c7i.8xlarge	us-west2	64 GB RAM, Intel Xeon 8488C 32vcpu, 12.5 Gbps Band
GCP	c3.standard-8	us-west2	32 GB RAM, Intel Xeon 8481C 8vcpu, 32 Gbps Band
Lambda	–	us-e2	–

3 Methods

3.1 Image Processing Pipeline Use Case

To better understand performance differences between REST and GraphQL interfaces for serverless functions, we implemented an image processing pipeline. Our pipeline consists of seven serverless functions, where each performs a specific task: **rotate, flip, crop, brighten, contrast, grayscale, and resize**. To exercise the pipeline, we sent a 4.8 MB JPG image in the request payload which is just under AppSync's 5MB payload data limit. For our functions, intermediate data is passed between stages, enabling the ordering of filters to vary on demand. In our GraphQL implementation, for each request, GraphQL resolves the requested image processing filters and invokes them to eliminate multiple client-server round-trips. For our REST implementation, the client orchestrates the control flow by calling the corresponding serverless functions to apply the requested filters using multiple client-server round-trips.

This use case is well-suited for evaluating performance of GraphQL and REST interfaces for serverless functions. An image processing workflow involves computationally intensive independent functions that require data exchange between stages. Featuring intensive I/O and computation, the pipeline is an ideal use case to compare the performance of the GraphQL and REST interfaces. The pipeline's extensibility allows addition of new image processing functions, as well as the reordering of processing stages, enabling comparison of multiple image processing workflow sequences over image datasets.

3.2 REST and GraphQL Servers

Image processing functions were hosted using the serverless AWS Lambda FaaS platform [4]. ARM64 Graviton Lambda functions without simultaneous multi-threading (SMT) were used to reduce function performance variance [17]. Function REST interfaces were implemented using the AWS API Gateway, a managed service designed to implement scalable REST APIs [1].

To provide GraphQL interfaces for serverless functions, we investigated AWS AppSync and Apollo Server as described in Section 2.4 to investigate **(RQ-2)**. AppSync allowed us to measure the performance of serverless, fully managed GraphQL APIs, with built-in scaling and integration features. We also implemented GraphQL APIs on Apollo Server hosted using Amazon EC2 instances as an unmanaged GraphQL server that requires manual server setup and management. To host Apollo, we leveraged a c7i.8xlarge EC2 instance with 32 vCPUs and 64 GB memory @ 3.2 GHz with an Intel Xeon(R) Platinum 8488C processor. For AppSync and Apollo

Server, GraphQL resolvers written in Python invoked AWS Lambda functions using the AWS SDK (Boto3) [5].

3.3 Clients

To evaluate the performance of REST and GraphQL interfaces for serverless functions, we tested a mix of local and cloud-based VMs to investigate multiple scenarios. These clients are described in Table 1.

1. Local: Testing was conducted using a local desktop computer. This test case enables benchmarking REST and GraphQL performance from offices or homes where the cloud is accessed using a shared internet connection with potentially high network latency and low bandwidth.

2. EC2: A c7i.8xlarge instance in us-west-2 was used to test REST and GraphQL performance when the client and backend shared a common cloud network.

3. GCP: We leveraged a Google Cloud Platform (GCP) VM in us-west-2 to test REST and GraphQL cross-cloud interface latency. In this configuration, the client and backend use cloud networks from different cloud providers.

4. AWS Lambda: For scalability testing of Apollo and AppSync **(RQ-2)**, we orchestrated concurrent calls to an AWS Lambda client function which invoked GraphQL backends to test performance under significant load. The Lambda client function invoked an Apollo or AppSync backend deployed in the same region, us-east-2.

4 Results

4.1 GraphQL vs. REST Performance Comparisons

To investigate **RQ-1**, we analyzed RTT for API calls executed from three different clients: a local machine, a us-west-2 Google Cloud VM, and a us-west-2 AWS EC2 instance. These clients represent three common scenarios: a local client, a client on another cloud provider, and a client on the same cloud as the serverless backend. We tested a GraphQL API that accessed serverless functions using Boto3 on an unmanaged Apollo server, the same API on Apollo which accessed serverless functions via the Amazon API Gateway, and a REST API hosted with the Amazon API Gateway. These APIs enable a performance comparison between GraphQL and REST, with and without the Amazon API Gateway.

Our findings revealed that Apollo Server leveraging the Amazon API Gateway provided the best performance of the interfaces tested; supporting the lowest RTTs. Although intuitively, one would assume that use of the API Gateway would increase latency due to the additional layer between the client and server, our results indicate that the API Gateway provides an optimized endpoint with faster response time versus invoking serverless functions directly in GraphQL resolvers using AWS SDK (Boto3). This performance improvement can likely be attributed to API Gateway optimizations in routing and caching, which reduce latency.

When comparing the performance of GraphQL to REST, GraphQL consistently outperformed REST in terms of RTT,

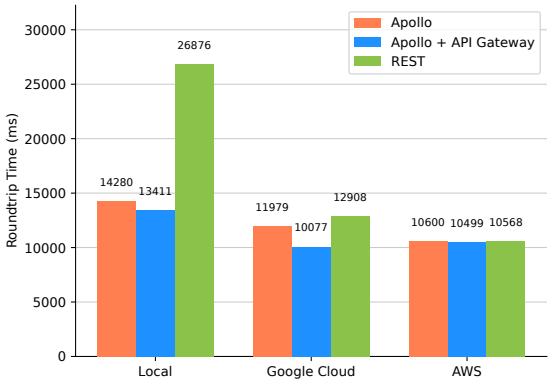

Figure 1. RTTs (s) of different clients in GraphQL and REST settings, clients: local desktop, AWS EC2, GCP VM (us-west) especially in environments with lower bandwidth and higher latency. The performance gap between GraphQL and REST was most noticeable when requests were made from a local machine, when requests were sent over a higher-latency, lower-bandwidth network. In such cases, GraphQL's ability to aggregate function calls and eliminate unnecessary roundtrips reduces the volume of data transferred, to help lower RTT. For cloud-based clients (i.e., Google Cloud VM and AWS EC2), the performance difference between GraphQL and REST interfaces is less. This is likely due to lower network latency because the client traffic travels over a cloud network vs. the internet.

Our results suggest that the decision to adopt GraphQL vs. REST interfaces for serverless functions should take client network conditions into consideration. In higher latency scenarios, when clients access the cloud via the internet, or for edge/IoT devices accessing the cloud from remote networks, GraphQL appears to provide a distinct advantage by combining function calls to eliminate extra roundtrips. In contrast, as bandwidth and latency improves, particularly in cloud-native setups, REST becomes more competitive, and the advantages of GraphQL, though still present, are less impactful. Selecting between these two architectures requires careful consideration of the specific deployment context and the expected network conditions.

4.2 GraphQL vs. REST Performance Scalability

To investigate scalability of API performance, we leveraged an 8 vCPU Google Cloud VM as a client and tested using 3, 10, 50, and 70 worker threads that performed 10 sequential calls each. Figure 2 illustrates the RTT distributions as the number of threads increases. Each row of graphs depicts the performance distribution with a given number of worker threads. The first row is 3 threads with 30 total runs, the second row is 10 threads with 100 total runs, the third row is 50 threads with 500 total runs, and the fourth row is 70 threads with 700 total runs. Columns depict the server type: Apollo, Apollo+API Gateway, and REST. The statistical data are displayed in each graph, with μ representing the mean, σ for standard deviation, and CV for coefficient of variation.

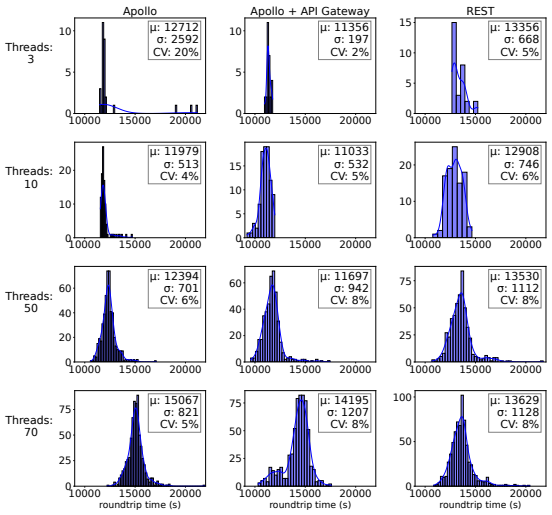

Figure 2. Distributions of RTTs with different number of threads, client: Google Cloud VM in us-west-2

The total average RTT are 13,038ms for Apollo, 12,070ms for Apollo and API Gateway, and 13,355ms for REST.

With low concurrency, the Apollo Boto3 configuration exhibited 20% variance caused by cold-start latency observed in a few runs. As concurrency increased, each distribution became more normally distributed with variance of 5 to 8%, a value likely caused by function runtime variance on AWS Lambda. With higher concurrency (70 threads), Apollo using the API Gateway exhibited a bimodal distribution. Inspection of the data revealed that with 70 concurrent clients, about 15% of the requests were processed by Apollo at the end of the batch of 700 requests when total server load dropped enabling faster RTT. With lower concurrency, the REST API and Apollo distributions appear more log normally distributed with a long right tail, indicating the presence of performance outliers. REST and Apollo performance trended towards normality as the thread and sample count increased.

4.3 Performance Comparisons for Unmanaged Apollo vs. Managed AppSync

To investigate **(RQ-2)**, an AWS Lambda function was used to generate concurrent calls to prevent bottlenecks which occur when using a single VM as multi-threaded client. Figure 3 compares mean RTT in seconds of Apollo and AppSync as the number of concurrent requests was scaled from 1 to 75.

With lower concurrency, both Apollo and AppSync provide similar response times, though Apollo was approximately 6% faster. RTT on the left side of the graph is influenced by server cold starts which appear to dissipate as concurrency is scaled up. As the thread count increases, a clear difference emerges between the two different servers. When scaled beyond 53 concurrent requests, AppSync provided lower RTTs compared to Apollo Server, which reached

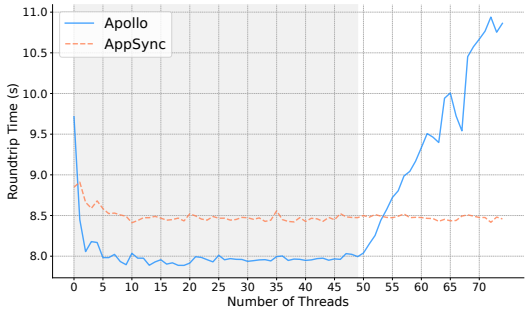

Figure 3. Apollo vs. AppSync when increasing thread counts, client: AWS lambda functions

a scaling bottleneck on a c7i.8xlarge EC2 instance. For up to 70 concurrent requests, AppSync's managed environment provided better scalability for parallel processing, which is crucial for applications requiring high concurrency. Apollo's mean RTT increases as the thread count rises, indicating greater sensitivity to concurrency, highlighting the limitation in Apollo's ability to handle high concurrency. To address scalability, Apollo Server can be deployed on a larger VM with more vCPUs, but there are limits to vertical scaling, with VMs with 192 vCPUs being the current upper limit on AWS. Apollo Federation can be leveraged to scale unmanaged Apollo servers beyond a single VM [10]. This requires additional effort to configure than a managed serverless AppSync for hosting GraphQL APIs.

4.4 Cost Comparison for Unmanaged Apollo vs. Managed AppSync

In this section, we compare hosting costs for GraphQL APIs associated with using AWS AppSync versus Apollo Server. The cost of data transfer and AWS Lambda execution are excluded from this analysis because they are essentially the same (i.e., AppSync charges the same price for data egress as EC2 for Apollo). Differences emerge when considering the cost models for hosting and request handling.

AppSync uses a pay-as-you-go pricing model, charging $4 per million queries and data modification operations. This linear and inexpensive price structure makes AppSync attractive for workloads with high volumes of requests and scenarios that involve rapid scaling. AppSync, as a managed service, eliminates user management and maintenance of infrastructure, which can further reduce operational costs.

We hosted **Apollo Server**, on an AWS c7i.8xlarge instance, which costs $1.428 per hour with on-demand pricing. To estimate the cost of providing a GraphQL interface, we evaluated cost with a concurrency of 53 threads, the point where Apollo RTT matches AppSync as shown in 3 at 8.5 seconds per request. At this concurrency level, we estimated the cost for Apollo to handle one million requests would be $63.62. **In contrast, the GraphQL interface cost using AppSync is just $4.** This comparison highlights the significant cost differential of hosting a GraphQL API (unmanaged

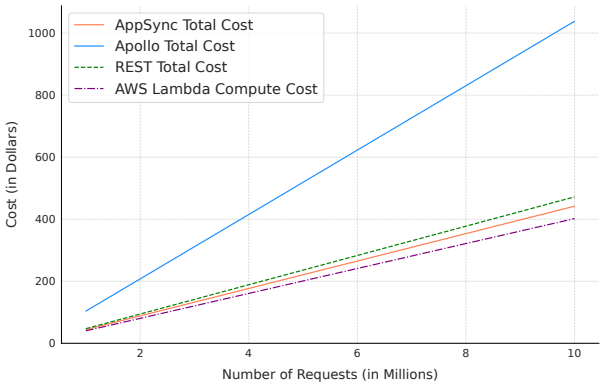

Figure 4. Apollo, AppSync and REST cost estimation comparison when increasing the request number under 53 threads, client: AWS lambda functions

vs. managed), and also the performance limitations when using Apollo Server.

As a baseline comparison, the **REST** API cost includes API Gateway's pricing. In us-east-2 (Ohio) the price is $1 for the first 300 million, and $0.9 later on. For our REST image processing pipeline, the price to invoke all 7 function calls in the pipeline is $7 per million.

Figure 4 illustrates the cost difference between AppSync and Apollo for increasing numbers of requests. The graph excludes AWS Lambda compute and data transfer charges since they will be essentially the same. For a small number of requests, both solutions are relatively inexpensive, but as the number of requests grows, Apollo's cost increases dramatically due to the higher infrastructure costs and performance degradation. **In contrast, AppSync's costs remain stable, scaling linearly at $4 per million requests, which is even less expensive than REST with the API Gateway.**

This analysis shows that AppSync offers a more cost-effective solution for handling large volumes of requests, particularly when the workload requires high scalability and low management overhead. In contrast, Apollo may be more suitable for use cases where the user requires greater control over the hosting environment, or where the request volume is low enough that the infrastructure cost is manageable.

5 Conclusions

This paper presents our investigation of GraphQL as an alternative interface to REST for serverless functions leveraging an image processing pipeline as a use case. **(RQ-1):** For serverless function pipelines, GraphQL eliminates client-to-server roundtrips to reduce RTT. RTT improvement is greater when the client-to-server latency is higher, a common network characteristic in edge computing environments. **(RQ-2):** AppSync, a managed GraphQL server, provided better performance at scale with lower costs, while Apollo Server (an unmanaged server), provided better RTT when not over-provisioned, but with higher costs. Providing GraphQL interfaces for our serverless functions was less expensive than REST. This savings will occur whenever GraphQL eliminates round-trips of client-to-server functions in the pipeline.

References

[1] 2024. Amazon API Gateway. https://aws.amazon.com/api-gateway/.
[2] 2024. Apollo GraphQL. https://www.apollographql.com.
[3] 2024. AWS AppSync. https://aws.amazon.com/appsync.
[4] 2024. AWS Lambda. https://aws.amazon.com/lambda.
[5] 2024. AWS SDK for Python. https://aws.amazon.com/sdk-for-python/.
[6] 2024. expressgraphql. https://github.com/graphql/express-graphql.
[7] 2024. Github GraphQL API. https://docs.github.com/en/graphql.
[8] 2024. graphql-js. https://github.com/graphql/graphql-js.
[9] 2024. Hasura. https://hasura.io.
[10] 2024. Introduction to Apollo Federation. https://www.apollographql.com/docs/graphos/schema-design/federated-schemas/federation.
[11] 2024. PostGraphile. https://www.graphile.org/postgraphile.
[12] 2024. REST APIs' Exhaustion Signs. https://www.programmersinc.com/over-fetching-and-under-fetching-rest-apis-exhaustion-signs.
[13] 2024. What is GraphQL and why Facebook felt the need to build it? https://buddy.works/tutorials/what-is-graphql-and-why-facebook-felt-the-need-to-build-it.
[14] 2024. Wikipedia: GraphQL. https://en.wikipedia.org/wiki/GraphQL.
[15] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based automated testing for GraphQL APIs. In *Proc of the Genetic and Evolutionary Computation Conf Companion*. 778–781.
[16] Mike Bryant. 2017. GraphQL for archival metadata: An overview of the EHRI GraphQL API. In *2017 IEEE Int Conf on Big Data*. 2225–2230.
[17] Xinghan Chen, Ling-Hong Hung, Robert Cordingly, and Wes Lloyd. 2023. X86 vs. arm64: an investigation of factors influencing serverless performance. In *Proceedings of the 9th International Workshop on Serverless Computing*. 7–12.
[18] Sijin Cheng and Olaf Hartig. 2022. LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers (Extended Version). *arXiv preprint arXiv:2208.04784* (2022).
[19] Marcos V De F. Borges et al. 2022. MicroGraphQL: a unified communication approach for systems of systems using microservices and GraphQL. In *Proc of the 10th IEEE/ACM Int Workshop on Soft Eng for Systems-of-Systems and Soft Ecosystems*. 33–40.
[20] Olaf Hartig and Jan Hidders. 2019. Defining schemas for property graphs by using the GraphQL schema definition language. In *Proc of the 2nd Joint Int Workshop on Graph Data Mgmt Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–11.
[21] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2021. Automatic property-based testing of graphql apis. In *2021 IEEE/ACM Int Conf on Automation of Soft Test (AST)*. IEEE, 1–10.
[22] Georgios Mavroudeas, Guillaume Baudart, Alan Cha, Martin Hirzel, Jim A Laredo, Malik Magdon-Ismail, Louis Mandel, and Erik Wittern. 2021. Learning GraphQL query cost. In *2021 36th IEEE/ACM Int Conf on Automated Soft Eng (ASE)*. IEEE, 1146–1150.
[23] Sabah Mohammed, Jinan Fiaidhi, Darien Sawyer, and Mehdi Lamouchie. 2022. Developing a GraphQL SOAP Conversational Micro Frontends for the Problem Oriented Medical Record (QL4POMR). In *Proc of the 6th Int Conf on Medical and Health Informatics*. 52–60.
[24] 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 Int Workshop on Soft Eng Research and Industrial Practice (SER&IP)*. IEEE, 10–17.
[25] Andrea Vázquez-Ingelmo, Juan Cruz-Benito, and Francisco J Garcí a Peñalvo. 2017. Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL. In *Proc of the 5th Int Conf on Technological Ecosystems for Enhancing Multiculturality*. 1–8.