GraphQL vs. REST: Investigating Performance and Scalability for Serverless Data Persistence

Runjie Jin, Robert Cordingly, Dongfang Zhao, Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma, Washington USA
rjjin, rcording, dzhao, wlloyd@uw.edu

***Abstract*—Serverless computing simplifies application deployment by removing the need for infrastructure management, with REST APIs being the common interface. Data persistence is essential for serverless applications because serverless functions are stateless and short-lived. To retain information across function executions, relational databases are commonly used to persist data, to ensure continuity, scalability, and reliable management. However, REST can lead to inefficiencies such as data over-fetching and under-fetching, which impact performance. GraphQL is a data query and manipulation language that supports client queries that specify what data is to be retrieved or modified. GraphQL resolvers fetch and transform data as specified by client queries.**

This paper compares the performance and scalability of GraphQL APIs as a database interface middleware in contrast to REST APIs implemented using serverless functions. We compare a GraphQL API implemented using the managed GraphQL AWS AppSync service, an unmanaged GraphQL API hosted using Apollo Server, and a traditional REST API implemented via the Amazon API Gateway and AWS Lambda. We compare these alternatives using clients implemented with serverless AWS Lambda functions, and also a local machine, an Amazon virtual machine (VM), and a Google VM. Our data APIs accessed a managed Amazon Aurora PostgreSQL cluster populated with the U.S. Centers for Medicare & Medicaid Services (CMS) Open Payments dataset. Our GraphQL implementations show content-dependent performance compared to REST, with Apollo Server demonstrating 25-67% faster average round-trip times vs. REST for most operations, but worse scalability than REST with very high concurrent workloads. Our findings provide practical guidance for developers selecting serverless architectures for GraphQL and REST APIs based on specific application requirements, network conditions, and expected request volumes.

***Index Terms*—GraphQL, REST, serverless, database performance, API, AppSync, cloud computing**

I. Introduction

Serverless computing has transformed application deployment by abstracting away infrastructure management, allowing developers to focus primarily on application logic while cloud providers automatically manage server resources on demand. In this paradigm, applications typically provide interfaces to serverless functions using RESTful APIs with standard HTTP methods (GET, PUT, POST, DELETE) for client-server communication. While REST's widespread adoption has made it the default interface for serverless functions due to its simplicity and extensive cloud provider support, it may not be optimal for all use cases. REST's data exchange inefficiencies (overfetching/underfetching) increase execution time and resource consumption—critical considerations in serverless environments where performance directly impacts cost.

GraphQL, a query language for APIs with an execution engine originally developed by Facebook in 2012 and open-sourced in 2015, offers a more dynamic alternative [1]–[3]. Unlike with REST's predefined HTTP methods, GraphQL enables clients to request precisely the data they need with fine-grained control. This capability makes it particularly suitable for serverless environments where minimizing data transfer and session overhead is essential for optimizing performance and reducing costs. By consolidating data from multiple sources into a single request, GraphQL can decrease client-server round trips for web applications and data processing pipelines, potentially improving resource utilization and response times.

Database operations are a fundamental component of modern cloud applications, ranging from customer management systems to data processing platforms. Data-driven workloads, such as data processing pipelines, represent a considerable portion of serverless functions in production environments. Despite GraphQL's adoption by major technology companies like GitHub [4] and Netflix [5], there remains a notable gap in research comparing its performance characteristics against traditional REST interfaces specifically to support data persistence for serverless applications.

Our study addresses this gap by comparing three alternative database middleware implementations: a GraphQL API hosted using the managed AWS AppSync service with direct database integration, a GraphQL API hosted using an unmanaged Apollo Server deployed to an Amazon VM, and a REST API implemented using Node.js AWS Lambda functions via the Amazon API Gateway. We implemented nine specific data APIs consisting of a representative set of select and join database queries. We evaluated key performance metrics including round-trip time (RTT) and throughput using four Node.js-based clients, including: AWS Lambda serverless functions, and also a local machine, an Amazon Elastic Compute Cloud (EC2) VM, and a Google Cloud Platform (GCP) VM. We compared the performance of our data APIs using these clients to consider how the implementations of the underlying middleware influence the performance outcomes.

A. Research Questions

This paper investigates two core research questions:

1) **RQ-1:** *(API performance - relational data API)* For a data-intensive API against a relational database, how do different GraphQL and REST middleware implementations compare in end-to-end performance (RTT, throughput) under various load scenarios?
2) **RQ-2:** *(API Scalability)* How do these different alternative APIs scale under increasing concurrency and what are the implications for RTT stability and maximum achievable throughput?

B. Contributions

This study compares GraphQL vs. REST as alternative data API middleware implementations for relational database (RDB) access while making the following contributions:

- *Performance Evaluation:* Systematic comparison of managed (AWS AppSync) vs. unmanaged (Apollo Server) GraphQL and REST (API Gateway) data API performance using nine representative database API endpoints against an Amazon Relational Database Service (RDS) Aurora PostgreSQL relational database.
- *Scalability Evaluation:* Statistical analysis of query latency and throughput while scaling from 1 to 100 concurrent AWS Lambda function database query requests.

II. Related Work

Early empirical comparisons of GraphQL vs. REST interfaces for data persistence focused on traditional server environments. Vadlamani et al. [6] evaluated response time tradeoffs using a custom GitHub client, concluding that GraphQL and REST each retain unique advantages. They interviewed GitHub employees, finding that each paradigm has its best adoption scenarios. Other studies measured performance in niche contexts: Mohammed et al. [7] and Vázquez-Ingelmo et al. [8] utilized GraphQL for an API to access medical records and an observatory data API, respectively. Similarly, Hartina et al. [9] examined a university information system, Lee et al. [10] assessed mobile ESS data servers, and Lawi et al. [11] analyzed high-volume management systems. While insightful, these investigations predominantly focused on standalone or VM-based data API clients rather than modern serverless applications, leaving open the issue for how FaaS environments are impacted by data persistence middleware alternatives.

Several research efforts have analyzed the performance of GraphQL APIs [12]–[15]. These efforts describe valuable methodologies, but often concentrate solely on benchmarking GraphQL performance without contrasting it with equivalent REST APIs or investigating serverless function interfaces. Cheng and Hartig's LinGBM benchmark [12] offers a standardized test suite for evaluating GraphQL server implementations. Belhadi et al. [13] introduced a testing framework based on the open-source EvoMaster tool to automatically generate test cases.

Formal treatments, such as the Cha et al. cost analysis framework [16] and Hartig and Pérez's semantic study [17] establish theoretical foundations for query optimization. Industrial adopters highlight GraphQL's scalability: Netflix's federation architecture [5] and Airbnb's Apollo-powered migration [18] showcase real-world success at massive scale. Despite this rich literature, additional investigation can contribute new knowledge and insights in the form of systematic comparisons of GraphQL and REST data API implementations for serverless application data persistence, a gap our study addresses.

III. Experimental Setup

A. API Architectures

To comprehensively evaluate performance and scalability, we implemented three distinct data APIs as middleware to provide access to a back-end relational database:

- **REST API (Baseline):** Our baseline data API is a traditional REST API implemented using a standard serverless architecture. We leveraged the Amazon API Gateway to provide a REST interface to trigger Node.js AWS Lambda serverless functions to perform database queries [19]. This represents a conventional approach to building data APIs in a serverless environment.
- **AWS AppSync (Managed GraphQL):** Our first GraphQL implementation uses AWS AppSync, a fully managed GraphQL service [20]. We configured AppSync with direct Amazon Aurora data sources, allowing it to use built-in resolvers to fetch data from our back-end Amazon Aurora DB. This architecture represents a highly integrated, managed approach to GraphQL.
- **Apollo Server (Self-Hosted unmanaged GraphQL):** To contrast the managed service, our second GraphQL implementation uses Apollo Server, a popular open-source spec-compliant GraphQL server, deployed on a c7i.8xlarge AWS Elastic Compute Cloud (EC2) instance with 32 vCPUs and 64 GB memory [21], [22]. This server runs a Node.js application where GraphQL resolvers execute database queries. This architecture represents a self-hosted, unmanaged approach, giving us more control at the cost of manual management.

B. Database Infrastructure

We executed all experiments against an Amazon Aurora PostgreSQL 16.4 cluster (instance class `db.r5.4xlarge`) with 16 vCPUs and 128 GB of memory populated with the 2018 U.S. Centers for Medicare & Medicaid Services (CMS) Open Payments dataset (6.5 GB uncompressed) [23]. Aurora is a managed, cloud-based relational database service provided as part of Amazon Web Services that offers high performance, availability, and scalability [24]. Aurora provides a relational database service for MySQL and PostgreSQL, enabling developers to leverage existing tools and applications built against these common back-end databases. Aurora is a part of the Amazon relational database service (RDS), which automates the management of various aspects of database administration, including backups and failover. To contrast the performance of GraphQL data interfaces, we built a REST data API consisting of endpoints hosted using the Amazon API

TABLE I: Client Environment Specifications

Client	Provider	Cores	CPU Model	Mem	OS	Kernel	Details
AWS EC2	AWS	32	Intel Xeon Platinum 8488C	64 GiB	Ubuntu 24.04	6.8.0-1024-aws	c7i.8xlarge, us-east-2a
GCP VM	Google Cloud	8	Intel Xeon @ 2.80 GHz	32 GiB	Ubuntu 24.04	6.14.0-1006-gcp	n2-standard-8, us-east1-d
AWS Lambda	AWS	2	Intel Xeon @ 2.50 GHz	512 MB	Amazon Linux 2	5.10.235--247.919.amzn2.x86_64	Function: client-collector; mem config: 512 MB; timeout: 5 min
Local Machine	Bare-metal Local Machine	14	13th Gen Intel Core i5-13600K	64 GiB	Arch Linux	6.13.7-arch1-1	—

Gateway, where calls were passed through to AWS Lambda functions to implement identical database operations. Lambda functions were located in the us-east-2 region, with a memory setting of 2048 MB to ensure they have access to at least 1 full vCPU (i.e. 100% vCPU timeshare for one vCPU core) [25]. AppSync, Apollo, and REST connect to the Aurora RDS, which provides a public DB endpoint. We also tested performance using a private Aurora DB endpoint deployed using a virtual private cloud (VPC). As the results for the private DB endpoint are similar, we focus our presentation on the performance using a public DB endpoint.

Table I describes the hardware and network configurations of our test clients used to evaluate our data APIs. The EC2 instance, and Lambda functions are co-located in AWS `us-east-2`, minimizing cross-region latency; the GCP VM resides in `us-east1-d`, offering a useful cross-cloud comparison, and the local machine leveraged a personal home network environment (460 Mbps download / 175 Mbps upload) in the state of Washington, USA.

C. Dataset

We use three tables from the CMS dataset: general payments, research payments, and physician ownership information [23]:

- **General Payments:** 10.9 M rows, 91 columns (5.8 GB CSV).
- **Research Payments:** 0.791 M rows, 252 columns (729.4 MB CSV).
- **Physician Ownership Information:** 3.6 K rows, 30 columns (1.4 MB CSV).

We configured simple keys and indexes to increase query performance. Each table uses `recordId` as its primary key, and the general payments and research payments tables link to physician ownership via `physicianProfileId` and `teachingHospitalId` in the physician ownership table.

D. API Endpoints

We created nine database API endpoints that cover lookup, filter, count, and aggregation patterns and investigated their performance using REST and GraphQL APIs. Our focus was on evaluating raw database query performance using REST vs. GraphQL data APIs as opposed to aggregate query performance, where multiple queries are combined into a single round trip. Aggregate queries are not natively supported by REST APIs.

- **`generalPaymentById`**: lookup a general payment by **`recordId`** (177 byte response, average RTT 3.2ms w/ Apollo Server).
- **`ownershipPaymentById`**: lookup an ownership payment by **`recordId`** (197 byte response, average RTT 3.6ms w/ Apollo Server).
- **`researchPaymentById`**: lookup a research payment by **`recordId`** (187 byte response, average 3.3ms RTT w/ Apollo Server).
- **`generalPaymentsByPhysicianId`**: fetch all general payments for **`physicianProfileId`** (8,769 byte response, average RTT 4.9ms w/ Apollo Server).
- **`generalPaymentsByTeachingHospitalId`**: fetch all general payments for **`teachingHospitalId`** (51 byte response, average RTT 3.2ms w/ Apollo Server).
- **`aggregatedGeneralPaymentsByPhysicianId`**: compute SUM and COUNT per **`physicianProfileId`** (121 bytes response, average RTT 3.5ms w/ Apollo Server).
- **`uniqueParties`**: list up to 1000 distinct physicians and hospitals (170,821 byte response, average RTT 19.0ms w/ Apollo Server).
- **`countPhysicians`**: count the total number of physicians (35 bytes response, average RTT 1.25s w/ Apollo Server).
- **`countHospitals`**: count the total number of hospitals (32 bytes response, average RTT 787ms w/ Apollo Server).

E. Workload Generation & Metrics

To investigate **RQ-1**, we tested all our database API endpoints using a custom Node.js client script (Node v20). For each client and each endpoint, we executed three load scenarios: 30 runs x 1 thread, 50 runs x 10 threads (5 runs per thread), and 150 runs x 50 threads (3 runs per thread). Each thread performed three warm-up runs prior to actual runs. Data from warm-up runs was discarded for the analysis to mitigate performance effects from cold-starts. These tests did not slowly scale the number of client threads, so the back-end database was less able to adapt quickly. To investigate **RQ-2**, we scaled up the number of client threads by 1 from 1 to 100 using concurrent calls to an AWS Lambda function. Using Lambda functions as a client avoids any potential bottleneck which may occur when using a single VM for a multi-threaded scalability test. To ensure equality in our comparisons, we used default GraphQL server configurations and we did not enable or configure special caches. We note that GraphQL servers support optional caching optimizations which can be tuned to

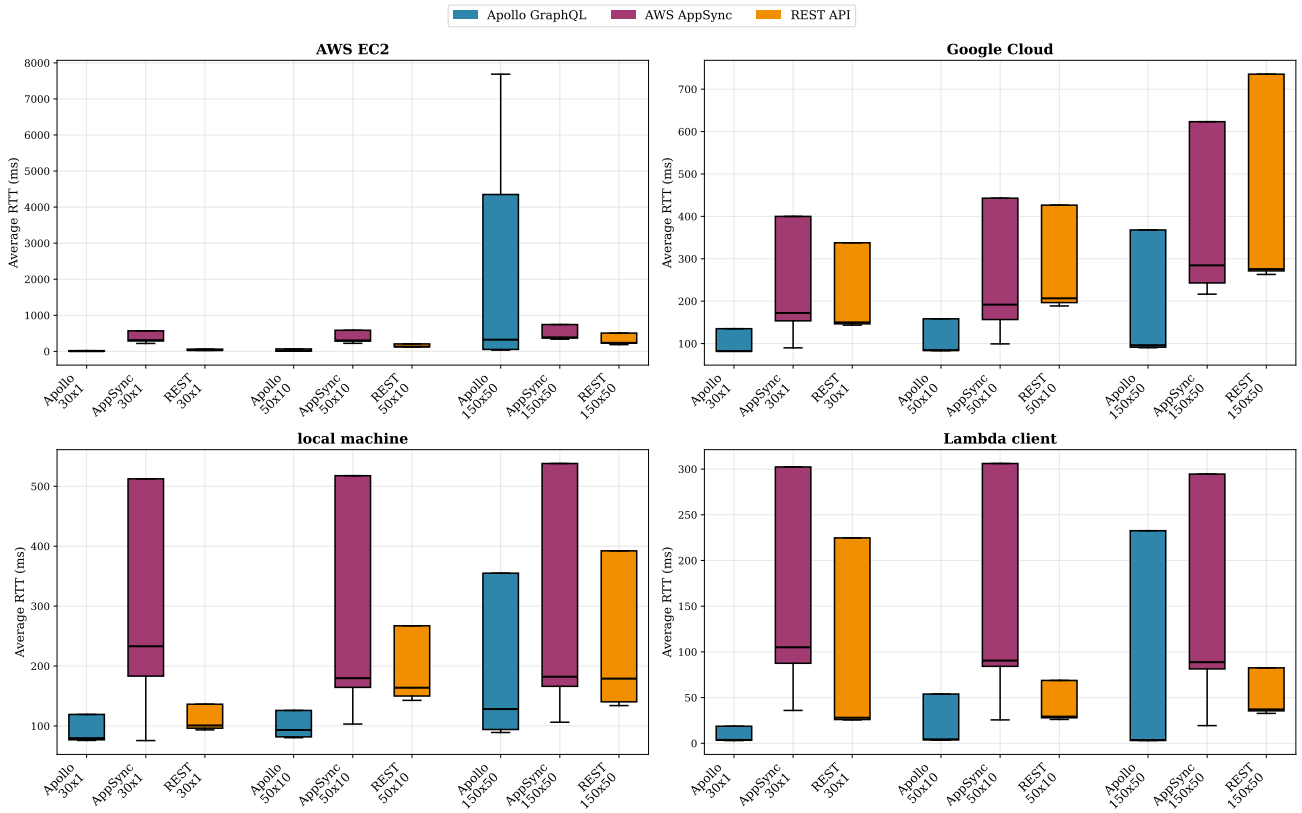

Fig. 1: RTT distributions with clients hosted on alternate platforms with increasing load scenarios. RTTs are averaged for nine endpoints across servers and testing scenarios. Apollo RTT appears more dependent on the client's host platform with consistently good RTTs. AppSync has consistent but higher latency, and REST performance falls in between with moderate cross-platform variation.

specific customer and use case requirements [26], [27]. We evaluated the following metrics:

Round-trip Time: End-to-end RTT observed by the client, includes two-way network latency, bootstrapping (i.e., cold start, if applicable) and back-end processing.

Throughput: Throughput is computed as:

$$\frac{\text{total successful requests}}{\text{wall-clock time after warm-up}} \quad [\text{requests/s}].$$

F. Clients

To evaluate the performance of REST and GraphQL data APIs as middleware alternatives, we tested clients implemented with serverless functions and a mix of local and cloud-based VMs. These clients are further described in Table I.

1. AWS Lambda: For scalability testing of Apollo and AppSync, we orchestrated concurrent calls to an AWS Lambda client function which invoked GraphQL backends to test performance under heavy load.

2. Local Machine: Testing was conducted using a local desktop computer. This test case benchmarked REST and GraphQL performance from a office or home-like environment where the cloud is accessed using a shared internet connection with potentially higher network latency and lower bandwidth.

3. Amazon EC2 VM: A c7i.8xlarge ec2 instance in us-east-2a was used to test REST and GraphQL performance when the client and backend shared a common cloud network.

4. GCP VM: We leveraged a Google Cloud Platform (GCP) n2-standard-8 VM in us-east1-d to test REST and GraphQL cross-cloud interface latency [28]. In this configuration, the client and backend use cloud networks from different cloud providers.

IV. Results

A. Cross-Platform Performance Analysis

Empirical results for **RQ-1** revealing performance characteristics for each of the tested client environments are shown in Figure 1. Our unmanaged Apollo GraphQL API demonstrated exceptional performance on AWS Lambda function clients, achieving median RTTs under 10ms across all load scenarios, while also supporting competitive performance on AWS EC2 under light loads. However, Apollo shows more variability on AWS EC2 under high load (150×50), with RTT distributions spanning 50-7500ms.

AWS AppSync exhibits consistently high median RTTs in the 100-500ms range across all of the tested clients and load conditions, trading stability for performance. Our REST data API performance also varied by client platform, ranging from 50-70 ms on AWS EC2 under light load to 300-400ms on Google Cloud and a local machine under high load.

The results reveal different performance trade-offs: Apollo GraphQL delivers superior performance for light loads with

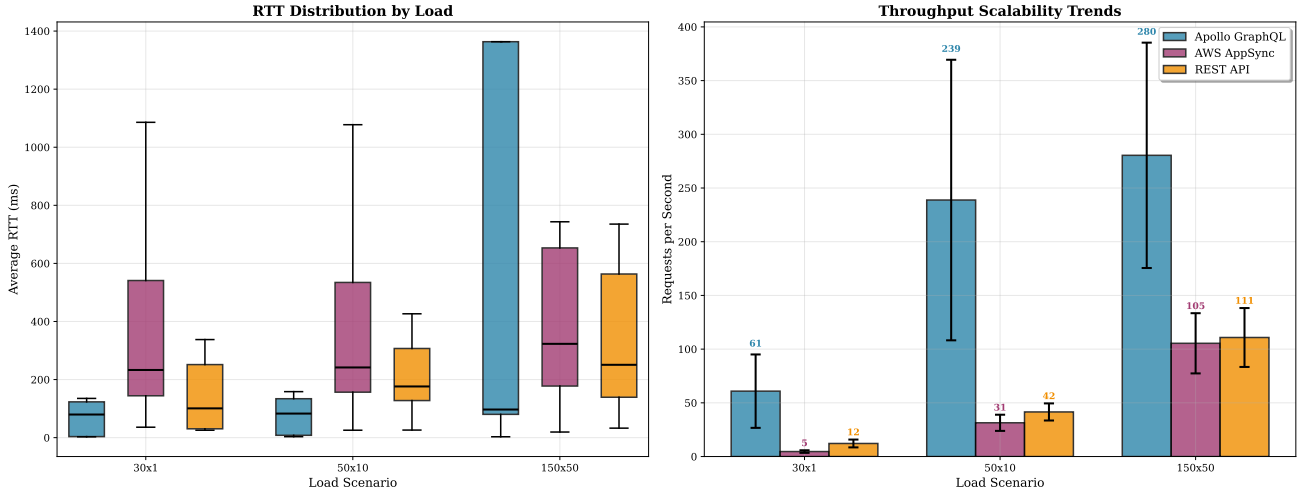

Fig. 2: Average RTT and throughput for all nine data endpoints and clients combined across load scenarios showing RTT distributions (left) and throughput scalability trends (right). Apollo GraphQL has good performance with low-latency performance and throughput growth compared to REST and AWS AppSync.

limited concurrency offering very good performance (e.g. EC2 and AWS Lambda), whereas AppSync provides consistent but slower responses, while our REST data APIs provided performance in between that of Apollo GraphQL and AWS AppSync GraphQL with moderate sensitivity to the client type.

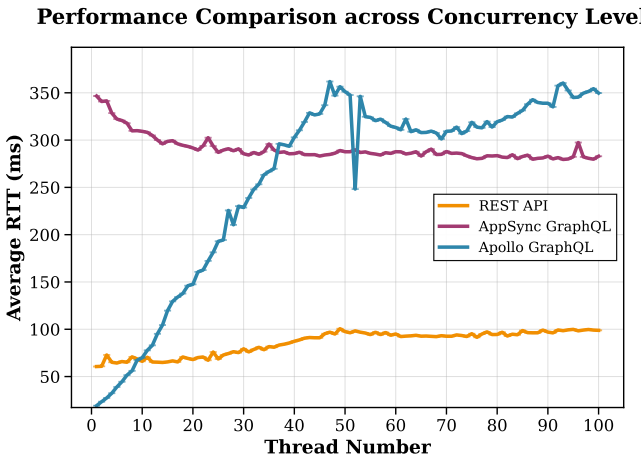

Fig. 3: Performance comparison across concurrency levels showing REST API's better scalability versus GraphQL implementations. Apollo GraphQL demonstrated the best low-concurrency performance but experienced performance degradation under load.

B. Load Scalability Characteristics

Figure 2 depicts average RTT and throughput for all nine data APIs combined for each of our three load scenarios in support of **RQ-1** and **RQ-2**. The graph shows that Apollo GraphQL sustains good throughput, achieving 280 requests/second with 50 concurrent clients under the 150×50 load scenario. At the same time, Apollo GraphQL maintains consistently low RTT distributions, with median response times remaining below 150 ms across all load scenarios while exhibiting the highest stability. In contrast, our REST API and AWS AppSync data APIs had similar throughput for concurrent load scenarios, both landing around 120 requests/second at peak load. The advantage of Apollo could be attributed to its efficient resolver execution and batching capability, making it a good choice when using GraphQL for database purposes.

C. Concurrency Performance Analysis

Figure 3 compares performance of our three data APIs under increasing load in support of **RQ-2**. We scaled from 1 to 100 concurrent client threads. Each thread sent 30 consecutive query requests resulting in an increasing total number of queries from 30 to 3000.

Our REST API demonstrated the best scalability, maintaining consistent 60-100 ms response times across all concurrency levels, providing 65-72% lower RTT than GraphQL variants at high loads (80-100 threads).

Apollo GraphQL had exceptionally good performance at low-concurrency (20-45ms RTT, 1-5 threads), but experienced dramatic degradation to 320-360ms RTT (10x increase) under medium-to-high loads. This performance drop is understandable since we deployed Apollo on a single EC2 instance, which lacks the ability to elastically scale to handle higher levels of concurrency. We note that Apollo Server performance can be scaled by deploying multiple instances behind a load balancer to handle increased request volume, but with the tradeoff of additional infrastructure cost and management burden. AWS AppSync GraphQL exhibited performance between that of our Apollo and REST data APIs, stabilizing around 285-290ms after an initial warm-up period. AppSync consistently outperformed Apollo GraphQL by 10-15% at higher concurrency levels, above 36 concurrent requests. We note that our Apollo Server, hosted on the 32 vCPU c7i.8xlarge ec2 instance, becomes over-provisioned above ∼32 concurrent requests.

Our findings indicate that while Apollo GraphQL offers compelling advantages for low-traffic scenarios, REST APIs

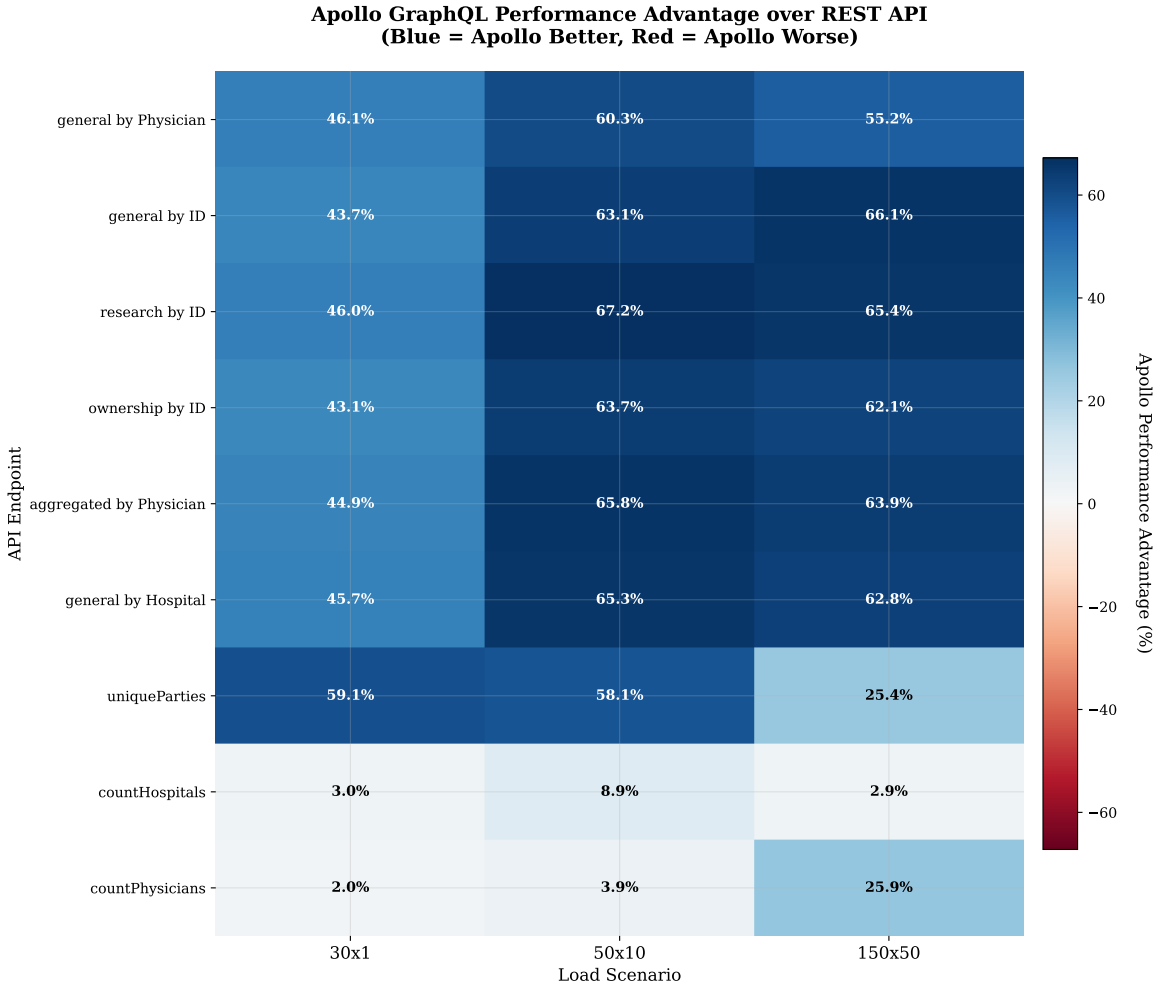

Fig. 4: Apollo GraphQL average RTT advantage heatmap over REST API showing consistent 25-67% improvements across endpoints and load scenarios, with peak advantages at medium concurrency levels.

offer better performance for workloads exceeding moderate concurrency thresholds. Performance of GraphQL interfaces excelled in low-concurrency scenarios, but REST is preferable for high concurrency.

D. Endpoint Performance comparison

Figure 4 depicts Apollo GraphQL's performance advantage over REST for each of the endpoints and load scenarios. Apollo demonstrates 25-67% lower average RTT, with peak advantages of 58-67% occurring at medium loads (50 runs × 10 threads). The slowest `countHospitals` and `countPhysicians` endpoints exhibit minimal average RTT improvement (3-9%), likely because most of the RTT is accounted for by time spent performing the query within the database engine.

The `uniqueParties` endpoint exhibits 58-59% lower average RTT. This query involved complex data retrieval involving multiple joins. These performance improvements, shown for diverse operation types provide empirical validation that GraphQL, especially Apollo, is a desirable choice when designing serverless relational database applications.

V. Discussion and Conclusion

In this paper, we investigated performance and scalability of GraphQL and REST data APIs by comparing the performance using nine different data API endpoints. Our investigation leveraged four distinct clients implemented with Node.js to invoke our data APIs. Clients included serverless AWS Lambda functions, a local machine, an Amazon VM, and a Google VM. These clients leveraged our data APIs that interfaced with a managed Amazon Aurora PostgreSQL database server to support answering **RQ-1** and **RQ-2**.

Regarding **RQ-1**, which examines performance under various load scenarios, our findings demonstrate that the best API choice depends on the specific load conditions and deployment context. Apollo GraphQL exhibited 25-67% lower average RTT vs. REST for most endpoints and load scenarios, with particularly lower average RTT for complex data retrieval operations involving multiple joins. However, because our deployment of Apollo Server was limited to 32 vCPUs, this performance advantage was best at low to medium concurrency levels, enabling Apollo to maintain consistently low

RTT distributions with medians below 150ms while achieving a good throughput of 320 requests per second compared to REST's 120 requests per second.

For **RQ-2** on scalability, while Apollo GraphQL demonstrated outstanding low average RTTs of 20-45ms with low-concurrency, it experienced significant performance degradation under loads with higher concurrency, reaching 320-360ms average RTTs (10x higher). In contrast, REST APIs exhibited better scalability, maintaining consistent 60-100ms RTTs across all concurrency levels up to 100 threads while providing 65-72% lower average RTTs vs. GraphQL implementations at high loads exceeding 80-100 concurrent threads. AWS AppSync's performance fell in between REST and Apollo, exhibiting more stable performance than Apollo under high concurrency while maintaining average RTTs around 285-290ms.

Our findings suggest that the choice between GraphQL and REST for data-intensive serverless workloads should be informed by expected traffic patterns and concurrency requirements. GraphQL, particularly Apollo, represents a good choice for applications with moderate traffic and complex data requirements, while REST is a robust option for high-throughput, high-concurrency scenarios where consistent performance under load is important.

ACKNOWLEDGMENTS

This research has been supported in part by AWS Educate Cloud Credits.

REFERENCES

[1] "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, 2024.
[2] "Wikipedia: Graphql," https://en.wikipedia.org/wiki/GraphQL, 2024.
[3] "graphql-js," https://github.com/graphql/graphql-js, 2024.
[4] "Github graphql api," https://docs.github.com/en/graphql, 2024.
[5] "How netflix scales its api with graphql federation," https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2, 2024.
[6] S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, "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)*. IEEE, 2021, pp. 10–17.
[7] S. Mohammed, J. Fiaidhi, D. Sawyer, and M. Lamouchie, "Developing a graphql soap conversational micro frontends for the problem oriented medical record (ql4pomr)," in *Proceedings of the 6th International Conference on Medical and Health Informatics*, 2022, pp. 52–60.
[8] A. Vázquez-Ingelmo, J. Cruz-Benito, and F. J. Garcí a Peñalvo, "Improving the oeeu's data-driven technological ecosystem's interoperability with graphql," in *Proceedings of the 5th Int. Conference on Technological Ecosystems for Enhancing Multiculturality*, 2017, pp. 1–8.
[9] D. A. Hartina, A. Lawi, and B. L. E. Panggabean, "Performance analysis of graphql and restful in sim lp2m of the hasanuddin university," in *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)*. IEEE, 2018, pp. 237–240.
[10] E. Lee, K. Kwon, and J. Yun, "Performance measurement of graphql api in home ess data server," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2020, pp. 1929–1931.
[11] A. Lawi, B. L. Panggabean, and T. Yoshida, "Evaluating graphql and rest api services performance in a massive and intensive accessible information system," *Computers*, vol. 10, no. 11, p. 138, 2021.
[12] S. Cheng and O. Hartig, "Lingbm: A performance benchmark for approaches to build graphql servers (extended version)," *arXiv preprint arXiv:2208.04784*, 2022.
[13] A. Belhadi, M. Zhang, and A. Arcuri, "Evolutionary-based automated testing for graphql apis," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2022, pp. 778–781.
[14] S. Karlsson, A. Čaušević, and D. Sundmark, "Automatic property-based testing of graphql apis," in *2021 IEEE/ACM International Conference on Automation of Software Testing (AST)*. IEEE, 2021, pp. 1–10.
[15] G. Mavroudeas, G. Baudart, A. Cha, M. Hirzel, J. A. Laredo, M. Magdon-Ismail, L. Mandel, and E. Wittern, "Learning graphql query cost," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1146–1150.
[16] A. Cha, E. Wittern, G. Baudart, J. C. Davis, L. Mandel, and J. A. Laredo, "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*, 2020, pp. 257–268.
[17] O. Hartig and J. Pérez, "Semantics and complexity of graphql," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 1155–1164.
[18] "How airbnb is moving 10x faster at scale with graphql and apollo," https://medium.com/airbnb-engineering/how-airbnb-is-moving-10x-faster-at-scale-with-graphql-and-apollo-aa4ec92d69e2, 2024.
[19] "Aws lambda," https://aws.amazon.com/lambda, 2024.
[20] "Aws appsync," https://aws.amazon.com/appsync, 2024.
[21] "Apollo graphql," https://www.apollographql.com, 2024.
[22] "Amazon elastic compute cloud documentation," https://docs.aws.amazon.com/ec2/, 2025.
[23] "CMS Open Payments — openpaymentsdata.cms.gov," https://openpaymentsdata.cms.gov/.
[24] "Relational database – amazon aurora mysql postgresql – aws," https://aws.amazon.com/rds/aurora/, 2025.
[25] R. Cordingly, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with cpu time accounting," in *2022 IEEE international Conference on Cloud Engineering (IC2E)*. IEEE, 2022, pp. 104–115.
[26] "Server-side caching - apollo graphql docs," https://www.apollographql.com/docs/apollo-server/performance/caching/, 2025.
[27] "Configuring cache backends - apollo graphql docs," https://www.apollographql.com/docs/apollo-server/performance/cache-backends/, 2025.
[28] "Compute engine — google cloud," https://cloud.google.com/products/compute/, 2025.