GraphQL vs. REST: Performance and Scalability Analysis for Serverless Applications

Runjie Jin

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Systems

University of Washington

2025

Committee:

Wes J. Lloyd

Dongfang Zhao

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

GraphQL vs. REST: Performance and Scalability Analysis
for Serverless Applications

Runjie Jin

Chair of the Supervisory Committee:
Wes J. Lloyd
School of Engineering and Technology

This thesis presents a comprehensive performance, scalability, and cost comparison of GraphQL and Representational State Transfer (REST) APIs within the context of serverless computing. While REST is the conventional choice for API implementation, its architectural style which is designed for network-based applications, specifically client-server applications, can lead to inefficiencies, such as over-fetching and under-fetching, leading to potential performance and price penalties in pay-per-use serverless environments. This work investigates GraphQL as a flexible and efficient interface alternative for two distinct and representative serverless application use cases: a CPU-bound image processing pipeline and a data-intensive relational database application.

For the CPU-bound pipeline, experimental results demonstrate that GraphQL reduces client-perceived Round Trip Time (RTT) by eliminating network latency associated with multiple client-to-server round trips required to orchestrate the workflow with REST. For the data-intensive workload, GraphQL implementations show content-dependent performance compared to REST, with Apollo demonstrating 25-67% performance improvements over REST on most operations, but worse scalability than REST under very high workloads.

Collectively, these findings illustrate that GraphQL provides advantages for serverless applications. The nature of these advantages is context-dependent, from orchestrating tasks in multi-step CPU-bound workflows to data-fetching from a relational database, establishing

GraphQL as a compelling architectural alternative for modern cloud-native applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to the University of Washington Tacoma, School of Engineering and Technology faculty and staff who provided guidance and support throughout the research. Special thanks to my advisor Wes J. Lloyd, who guided me by his expertise and insights.

DEDICATION

To my family and mentor who supported this journey.

Chapter 1

INTRODUCTION

Serverless computing has fundamentally changed how modern applications are deployed and managed by abstracting the underlying infrastructure. With Function-as-a-Service (FaaS) platforms, such as AWS Lambda, developers no longer manage servers or scaling, as these tasks are automated by the cloud provider. This paradigm enables developers to focus primarily on application logic, while server resources are adapted on demand. Serverless applications are typically event-driven and billed only for the resources consumed during execution, making them cost-efficient and scalable.

Traditionally, applications provide interfaces to serverless functions using RESTful APIs, which consist of standard HTTP methods like GET, POST, PUT, and DELETE. Representational State Transfer (REST) is widely recognized for its simplicity and has become the de facto standard for web services, with extensive support from cloud providers. However, while REST is a good baseline, its widespread adoption does not mean it is optimal for all serverless use cases. REST interfaces can lead to inefficient data exchange, such as over-fetching (retrieving unnecessary data) and under-fetching (requiring multiple API calls to retrieve sufficient data) [9]. These inefficiencies can increase execution time, resource consumption, and client-side complexity—critical considerations in a serverless environment where performance and resource utilization directly translate to cost.

This thesis investigates GraphQL as a powerful alternative to REST for serverless applications. GraphQL is a query language for APIs, backed by an execution engine, that allows clients to request exactly the data they need. Developed by Facebook and open-sourced in 2015 [10], it offers a more dynamic and efficient approach. By enabling fine-grained control over data and consolidating requests for multiple resources into a single client-server round trip, GraphQL is particularly well-suited to serverless environments where minimizing overhead is critical.

Despite the growing adoption of both serverless computing and GraphQL, there remains a gap in research to rigorously compares GraphQL's performance against RESTful APIs for representative serverless application workloads. This thesis contributes to fill this gap by evaluating these API paradigms using two distinct use cases: a CPU-bound image processing pipeline and a data-intensive relational database application. This approach allows for a nuanced understanding of how each API style (GraphQL vs. REST) performs under different types of stress unique to different use cases.

While RESTful APIs serve as a functional baseline for serverless computing, this thesis investigates the extent of performance, scalability, and cost advantages afforded by GraphQL for serverless applications, identifying tradeoffs based on the nature of the workloads. We investigate two representative serverless application use cases, showing how GraphQL's architectural benefits contribute differently but effectively for both CPU-bound and data-intensive tasks.

1.1 Research Questions

This thesis is built upon two distinct studies, each with its own focus, which collectively provide a comprehensive comparison of GraphQL and REST in serverless environments. To guide this investigation, the following research questions are addressed, drawing directly from the two use cases.

1. **RQ-1:** *(API performance - processing pipeline)* For a multi-step, CPU-bound serverless workload (an image processing pipeline), how do GraphQL and REST APIs compare in terms of end-to-end performance (RTT) and client-side orchestration complexity?

2. **RQ-2:** *(GraphQL managed vs. unmanaged - processing pipeline)* What are the performance, scalability, and cost trade-offs between a managed GraphQL service (i.e. AWS AppSync) and a self-hosted, unmanaged GraphQL server for hosting GraphQL APIs for our multi-step, CPU-bound workload?

3. **RQ-3:** *(API performance - relational data API)* For a data-intensive serverless API

(a relational database), how do GraphQL APIs and a traditional REST API compare in terms of end-to-end performance (RTT, throughput) under various load scenarios?

4. **RQ-4:** *(GraphQL managed vs. unmanaged - relational data API)* How do the GraphQL services and REST architectures scale under increasing concurrency for this data-intensive workload, and what are the implications for RTT stability and maximum achievable throughput?

1.2 Thesis Structure

This thesis is organized as follows: Chapter 2 provides background on the core technologies and reviews related work in the field. Chapter 3 presents our first study, a performance and cost comparison for a CPU-bound serverless image processing pipeline. Chapter 4 details our second study, which investigates performance and scalability for a data-intensive relational database application. Chapter 5 provides a synthesis and discussion of the findings from both studies, comparing and contrasting the results. Finally, Chapter 6 concludes the thesis and suggests directions for future work.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter provides the necessary background on the key technologies investigated in this thesis and reviews prior research in the fields of GraphQL, REST, and serverless computing.

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.

The programming language for writing these resolver functions is determined by the specific GraphQL server implementation. For instance, AWS AppSync offers native support for JavaScript and the Velocity Template Language (VTL), while also accommodating any language compatible with AWS Lambda to provide data for AppSync, such as Python, Go, and Java. Similarly, the Apollo server, though centered around JavaScript/TypeScript, uses its federation architecture to support subgraphs written in other languages, including

Go, Python, Ruby, and C. Consequently, developers can implement resolvers in a variety of modern programming languages by selecting a compatible server or platform service.

2.1.1 GraphQL Platforms

AWS AppSync is a fully managed GraphQL service offered by AWS [4]. 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.

Apollo Server is an open source GraphQL server designed to simplify the process of building, deploying, and maintaining GraphQL APIs [3]. 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.

2.2 Prior Comparative Studies

Prior research has conducted comparisons of GraphQL and REST, often focusing on specific use cases, surveys, or non-serverless environments.

2.2.1 Qualitative and Case Study Comparisons

Early empirical comparisons focused on traditional server environments. Vadlamani et al. [23] evaluated response-time trade-offs 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 have measured performance in niche contexts: Mohammed et al. [22] and Vázquez-Ingelmo et al. [24] utilized GraphQL for an API to access medical records and an observatory data API, respectively. Similarly, Hartina et al. [17] examined a university information system, Lee et al. [20] assessed mobile ESS data servers, and Lawi et al. [19] analyzed high-volume manage-

ment systems. While insightful, these investigations predominantly targeted standalone or VM-based deployments rather than modern serverless architectures, leaving open the issue for how FaaS environments shape API performance.

2.2.2 Performance Analysis and Benchmarks

Several research efforts have analyzed the performance of GraphQL APIs [11, 14, 18, 21]. 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 [14] offers a standardized test suite for evaluating GraphQL server implementations. Belhadi et al. [11] introduced a testing framework based on the open-source EvoMaster tool to automatically generate test cases.

Formal treatments—such as Cha et al.'s cost analysis framework [12] and Hartig and Pérez's semantics study [16]—establish theoretical foundations for query optimization. Industrial adopters highlight GraphQL's scalability: Netflix's federation architecture [8] and Airbnb's Apollo-powered migration [7] showcase real-world success at massive scale.

Despite this rich literature, no prior work to the best of our knowledge systematically compares GraphQL and REST for serverless workloads across varied client platforms and concurrency levels, considering both CPU-bound and data-intensive workloads — a gap our study addresses.

Chapter 3

PERFORMANCE FOR A CPU-BOUND IMAGE PROCESSING PIPELINE

To investigate our research questions in the context of a multi-step, computationally intensive workload, this chapter details a performance and cost comparison using a serverless image processing pipeline. This use case is designed to evaluate how GraphQL and REST handle the orchestration of a sequence of functions, a common pattern in serverless applications.

3.1 Methods

3.1.1 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, CPU-bound 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 in sequence on the server side, eliminating multiple client-server round-trips. For our REST implementation, the client orchestrates the control flow by calling the corresponding serverless functions sequentially, requiring multiple client-server round-trips to apply the requested filters.

This use case is well-suited for evaluating the performance of GraphQL and REST interfaces. A workflow involving a series of computationally intensive, independent functions that require data exchange between stages is an ideal scenario to compare the orchestration

efficiency of GraphQL versus REST.

3.1.2 REST and GraphQL Servers

Image processing functions were hosted using the serverless AWS Lambda FaaS platform [5]. ARM64 Graviton Lambda functions without simultaneous multi-threading (SMT) were used to reduce function performance variance [13]. Function REST interfaces were implemented using the AWS API Gateway, a managed service designed to implement scalable REST APIs [2].

To provide GraphQL interfaces for serverless functions, we investigated AWS AppSync and Apollo Server. 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) [6].

Table 3.1: 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	–

3.1.3 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 3.1.

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, we orchestrated concurrent calls to an AWS Lambda client function which invoked GraphQL backends to test performance under heavy load.

3.2 Results

3.2.1 GraphQL vs. REST Performance Comparisons

To investigate the performance aspect of **RQ-1** for this workload, 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.

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, 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.

3.2.2 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 3.2 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 statistical data are displayed in each graph, with μ representing the mean, σ for standard deviation, and CV for coefficient of variation. The total average RTT are

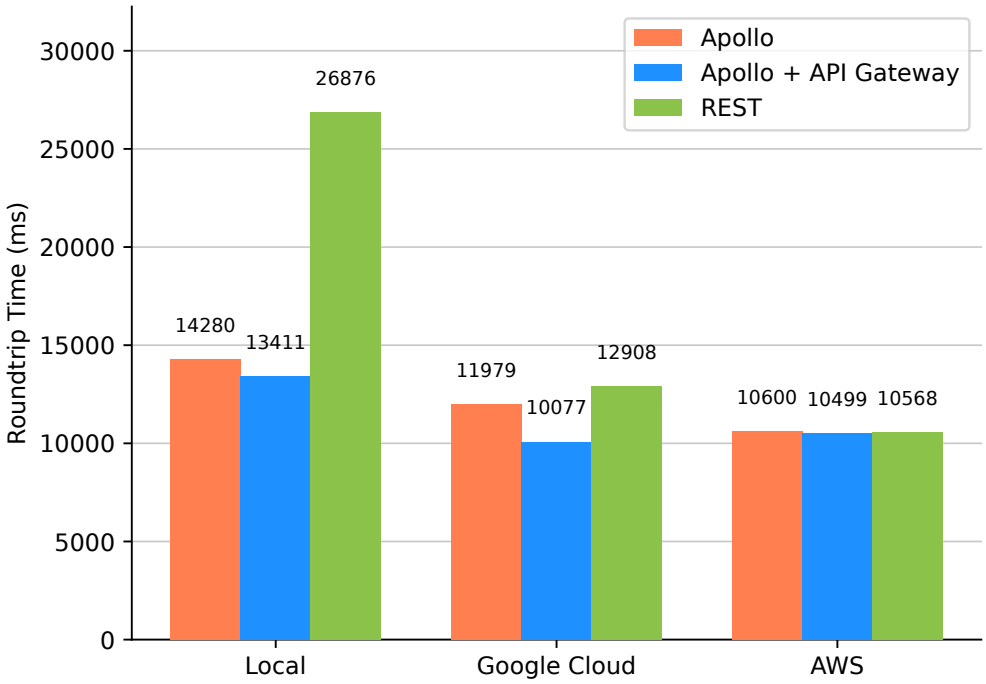

Figure 3.1: RTTs (s) of different clients in GraphQL and REST settings, clients: local desktop, AWS EC2, GCP VM (us-west)

13,038ms for Apollo, 12,070ms for Apollo and API Gateway, and 13,355ms for REST.

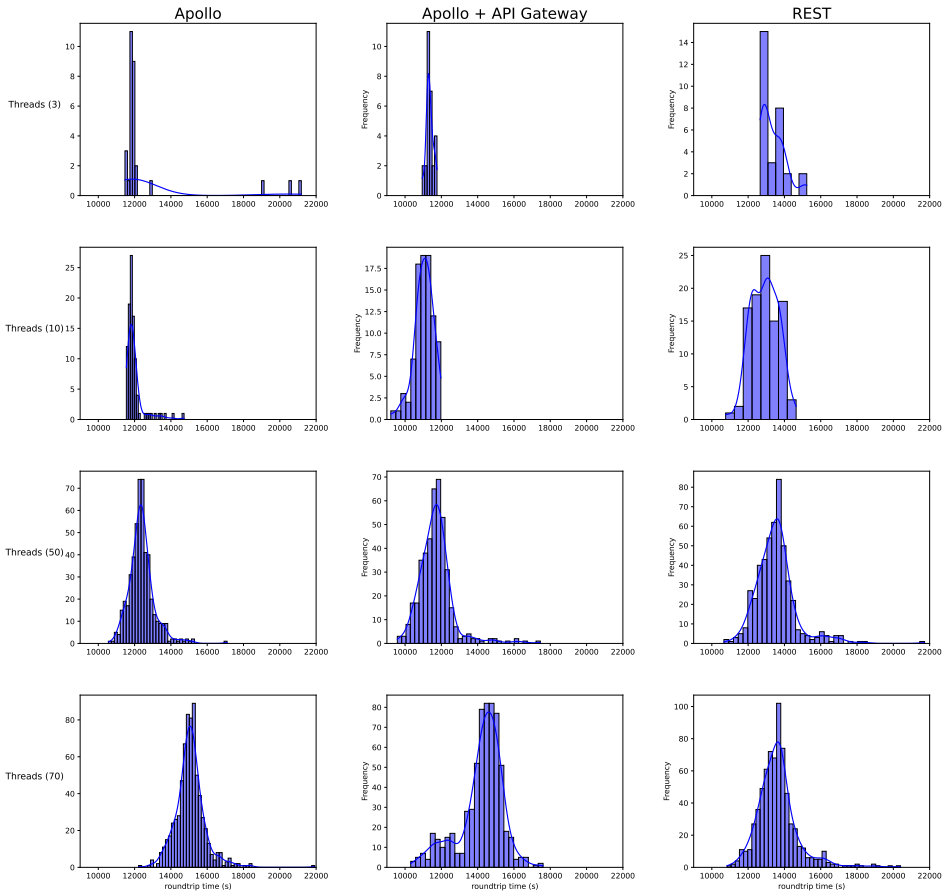

Figure 3.2: Distributions of RTTs with different number of threads, client: Google Cloud VM in us-west-2

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. 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.

3.2.3 Performance of 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 3.3 compares mean RTT in seconds of Apollo and AppSync as the number of concurrent requests was scaled from 1 to 75.

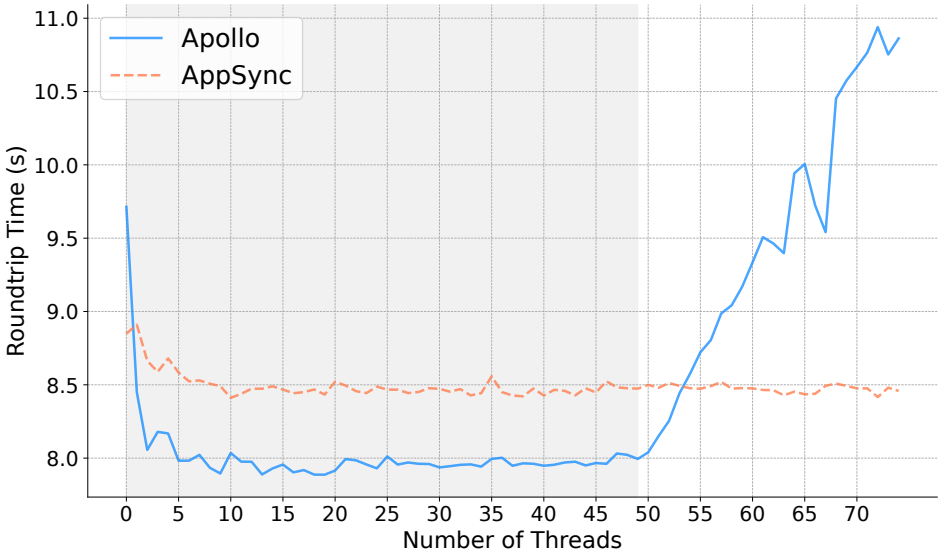

Figure 3.3: Apollo vs. AppSync when increasing thread counts, client: AWS lambda functions

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 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.

3.2.4 Cost Comparison of 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. 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 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 substantial cost differential of hosting a GraphQL API (unmanaged vs. managed), and also the performance limitations when using Apollo Server.

Figure 3.4 illustrates the cost difference between AppSync and Apollo for increasing numbers of requests. 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.**

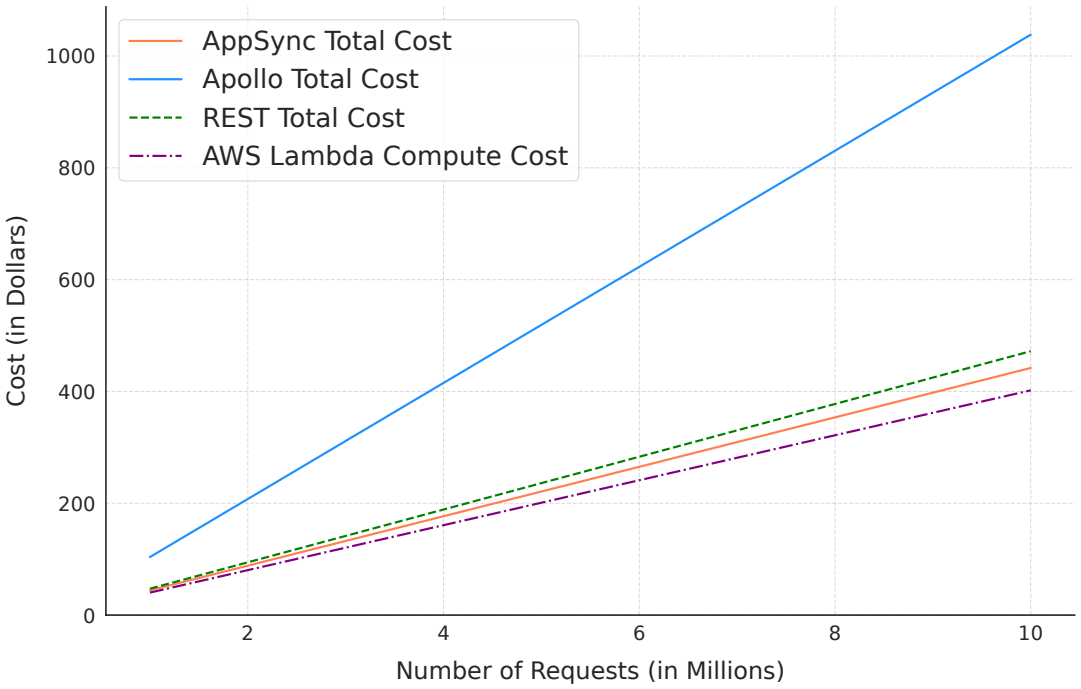

Figure 3.4: Apollo, AppSync and REST cost estimation comparison when increasing the request number under 53 threads, client: AWS lambda functions

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.

3.3 Chapter Conclusions

This chapter presented our investigation into the trade-offs between GraphQL and REST for a CPU-bound serverless pipeline, providing answers to our first two research questions.

In addressing **RQ-1**, our findings show that for multi-step workflows, GraphQL provides a distinct performance advantage by reducing client-side orchestration complexity. By bundling the seven-step image processing sequence into a single API call, GraphQL eliminates multiple client-to-server roundtrips, which significantly reduces the overall client-perceived RTT. This benefit is most pronounced for clients operating on high-latency networks.

For **RQ-2**, we compared a managed versus an unmanaged GraphQL solution. The results demonstrate that for this workload, a managed service like AWS AppSync offers superior scalability and cost-efficiency at scale. While the self-hosted Apollo Server showed slightly better RTT at very low concurrency, it was significantly more expensive and hit a clear scaling bottleneck under load, making AppSync the more practical choice for production serverless applications.

Chapter 4

PERFORMANCE FOR A DATA-INTENSIVE RELATIONAL DATABASE APPLICATION

To address research questions **RQ-3** and **RQ-4** and investigate GraphQL vs. REST interfaces in the context of a data-intensive serverless use case, this chapter evaluates the performance and scalability of GraphQL versus REST interfaces for providing a relational database application. Database operations are a fundamental component of modern cloud applications, and this study focuses on how each API paradigm handles high-concurrency data fetching and aggregation tasks.

4.1 Methods

4.1.1 Database Infrastructure

We executed all experiments against an Amazon Aurora PostgreSQL 16.4 cluster (instance class `db.r5.4xlarge`) populated with the 2018 U.S. Centers for Medicare & Medicaid Services (CMS) Open Payments dataset [1]. GraphQL requests target an AWS AppSync API (key-authenticated, built-in JavaScripts (JS) resolvers connected directly supported by AppSync–Aurora integration). Aurora is a managed, cloud-based relational database service provided as part of Amazon Web Services that offers high performance, availability, and scalability. Aurora provides a relational database service for MySQL and PostgreSQL, enabling developers to leverage existing tools and applications built against these common database backends. 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 performance of GraphQL data interfaces, we built a REST data API consisting of endpoints hosted using the API Gateway, where calls were passed through to AWS Lambda functions to implement identical database operations. Lambda functions are located in the us-east-2 region, with a memory setting of 2048 MB to make

sure they are given enough cpu resources [15]. AppSync, Apollo, and REST connect to the Aurora RDS which provides a public endpoint. While we also tested metrics from a private endpoint, the result is similar to the results of a public endpoint. As in chapter 3, we also hosted the Apollo GraphQL server on a c7i.8xlarge EC2 instance as the unmanaged GraphQL server for comparison.

Table 4.1: 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	—

Table 4.1 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.

4.1.2 Dataset

We use the three Open Payments tables from the CMS dataset: general, research, and ownership payments tables [1]:

- **General payments:** 10.8 M rows, 75 columns (about 6 GB CSV).
- **Research payments:** 0.58 M rows, 176 columns (about 500 MB CSV).
- **Ownership payments:** 3.3 K rows, 29 columns (about 1.5 MB CSV).

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

4.1.3 API Endpoints

We created nine database API endpoints covering lookup, filter, count, and aggregation patterns and investigate their performance using REST and GraphQL APIs. Our focus was on evaluating raw database query performance using REST vs. GraphQL 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 in apollo).
- `ownershipPaymentById`: lookup an ownership payment by `recordId` (197 byte response, average RTT 3.6ms in apollo).
- `researchPaymentById`: lookup a research payment by `recordId` (187 byte response, average 3.3ms RTT in apollo).
- `generalPaymentsByPhysicianId`: fetch all general payments for `physicianProfileId` (8,769 byte response, average RTT 4.9ms in apollo).
- `generalPaymentsByTeachingHospitalId`: fetch all general payments for `teachingHospitalId` (51 byte response, average RTT 3.2ms in apollo).
- `aggregatedGeneralPaymentsByPhysicianId`: compute SUM and COUNT per `physicianProfileId` (121 bytes response, average RTT 3.5ms in apollo).
- `uniqueParties`: list up to 1000 distinct physicians and hospitals (170,821 byte response, average RTT 19.0ms in apollo).
- `countPhysicians`: count the total physician number (35 bytes response, average RTT 1.25s in apollo).

- `countHospitals`: count the total hospital number (32 bytes response, average RTT 787ms in apollo).

4.1.4 Workload Generation & Metrics

We tested all of 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 where the data 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 database backends are less able to adapt quickly. This configuration is in contrast to our scaling experiment where the number of client threads increased by 1 for each iteration. 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 backend processing.

Throughput: Throughput is computed as:

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

4.2 Results

4.2.1 Cross-Platform Performance Analysis

Figure 4.1 reveals performance characteristics across different client environments. Apollo GraphQL demonstrates exceptional performance on Lambda clients, achieving median RTTs under 10ms across all load scenarios, and maintains 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 50-1000ms. AWS AppSync exhibits consistently high RTTs in the 100-500ms range across all environments and load conditions, trading stability for performance. REST's performance also varies 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.

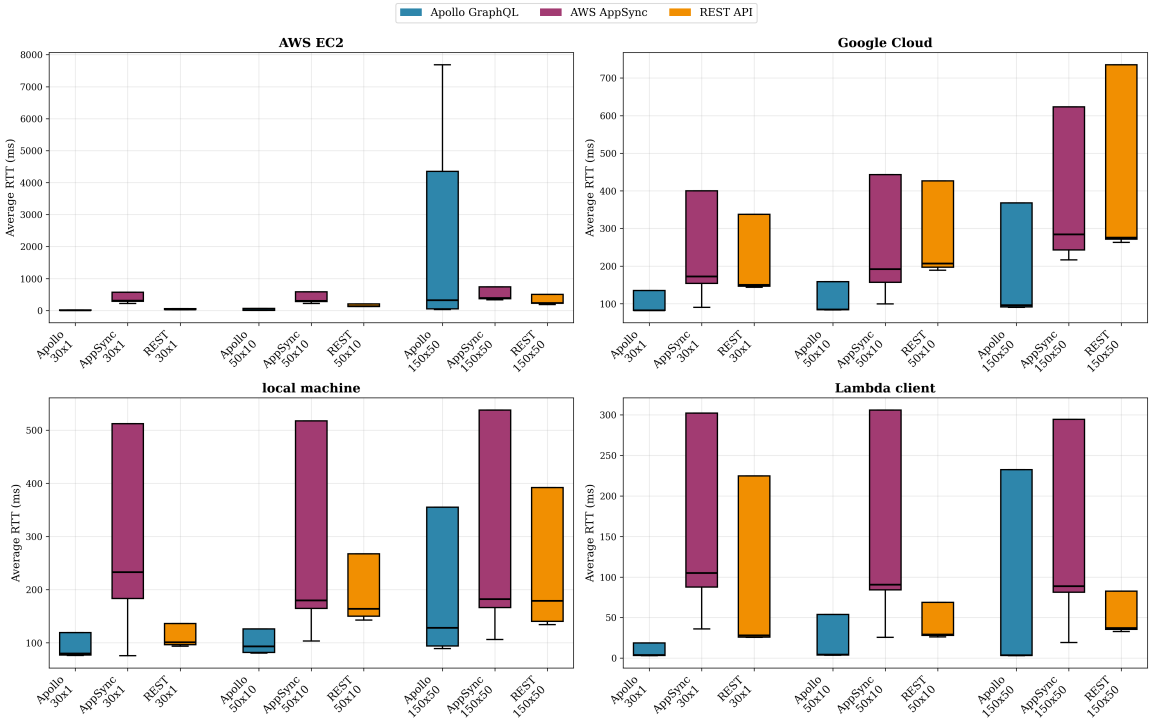

Figure 4.1: RTT distributions with clients hosted on alternate platforms with increasing load scenarios. RTTs are averaged for nine endpoints across servers and testing scenarios. Apollo RTT appears more dependent on the client's host platform with consistently good RTTs. AppSync has consistent but higher latency, and REST performance falls in between with moderate cross-platform variation.

The results reveal different performance trade-offs: Apollo GraphQL delivers superior performance with some clients affording even higher performance (e.g. EC2 and Lambda), whereas AppSync provides consistent but slower responses, while REST performance appears in an intermediate position with moderate sensitivity to the client's platform.

4.2.2 Load Scalability Characteristics

Figure 4.2 shows RTT and throughput for our alternative data APIs under increasing load conditions. The graph combines results for all 9 queries and shows Apollo GraphQL shows

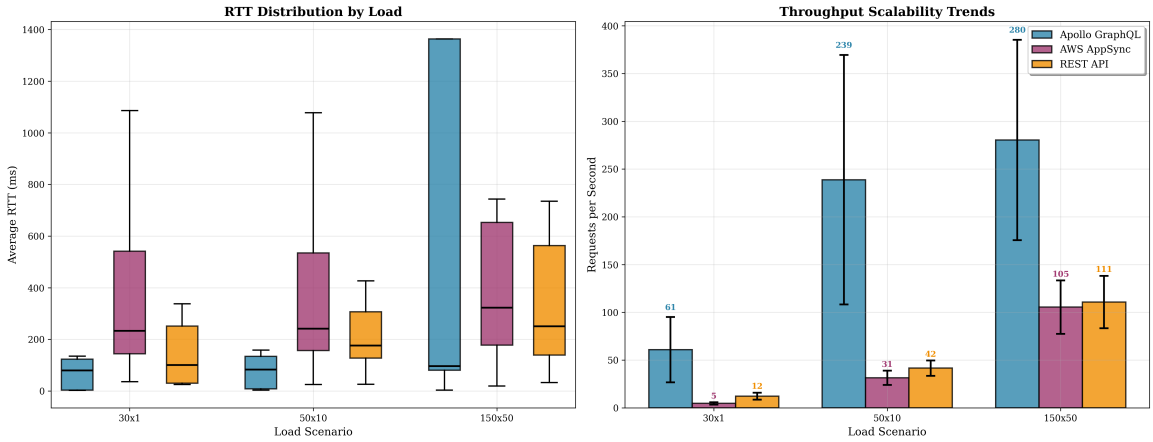

Figure 4.2: Average RTT and throughput for all nine data endpoints and clients combined across load scenarios showing RTT distributions (left) and throughput scalability trends (right). Apollo GraphQL has good performance with low-latency performance and throughput growth compared to REST and AWS AppSync.

good throughput scaling, achieving 280 requests/second at 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 interface had similar throughput under load, 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.

4.2.3 Endpoint Performance comparison

Figure 4.3 depicts Apollo GraphQL's performance advantage over REST across endpoints and load scenarios. Apollo demonstrates 25-67% performance improvements, with peak advantages of 58-67% occurring at medium loads (50×10). The slowest `countHospitals` and `countPhysicians` endpoints exhibit minimal RTT improvement (3-9%), likely because most of the total RTT is accounted for by time spent performing the query within the database engine.

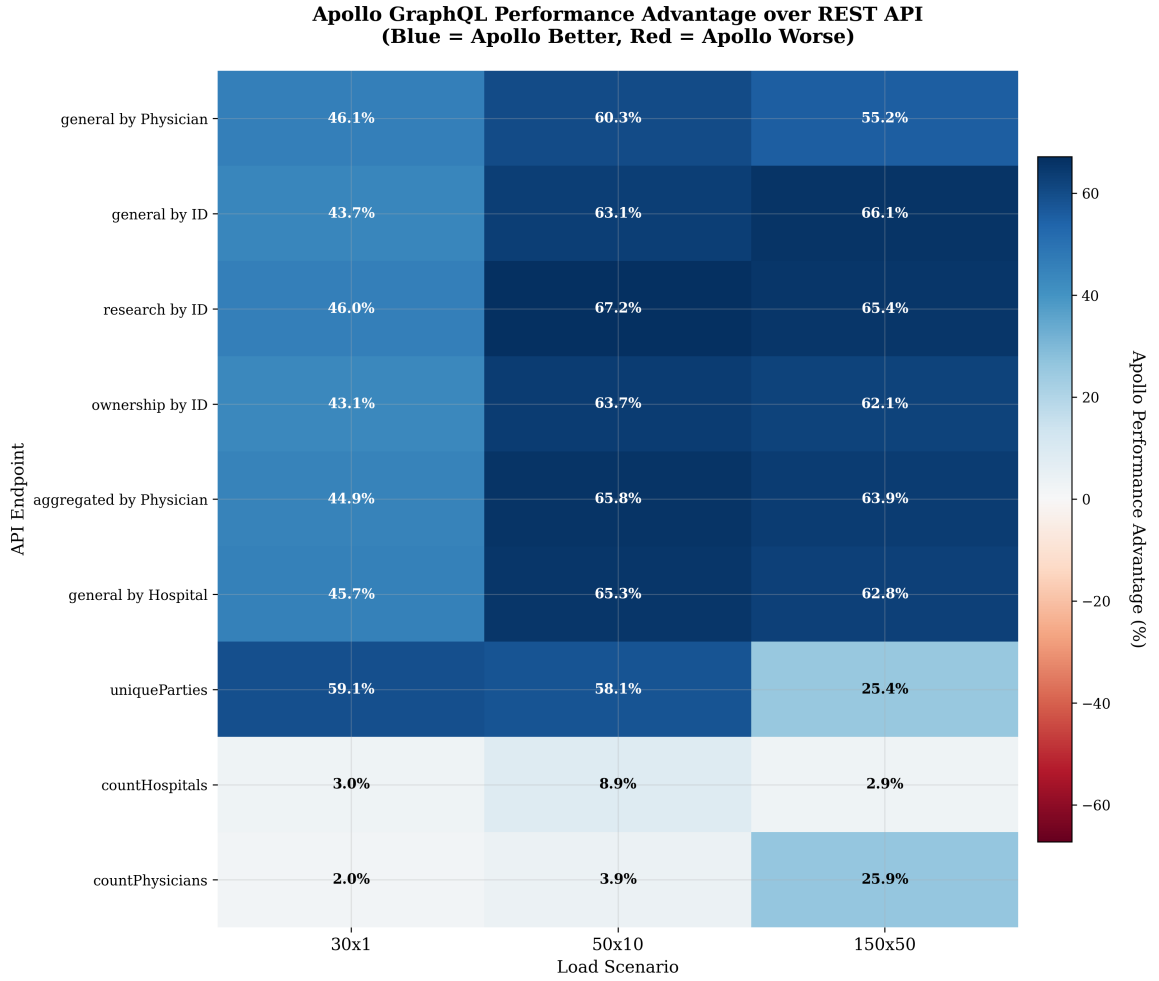

Figure 4.3: Apollo GraphQL average RTT advantage heatmap over REST API showing consistent 25-67% improvements across endpoints and load scenarios, with peak advantages at medium concurrency levels.

The `uniqueParties` endpoint exhibits a 58-59% advantage. 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.

4.2.4 Concurrency Performance Analysis

Figure 4.4 compares performance of our three database interface alternatives under increasing load. We scaled from 1 to 100 concurrent client threads. Each thread sent 30 sequential query requests resulting in an increasing number of queries from 30 to 3000 in total.

Our REST API demonstrated the best scalability, maintaining consistent 60-100 ms response times across all concurrency levels, representing a 65-72% performance advantage over GraphQL variants at high loads (80-100 threads).

Apollo GraphQL is exceptionally good at low-concurrency performance (20-45ms at 1-5 threads), but is followed by dramatic degradation to 320-360ms under medium-to-high loads. This performance drop is understandable since we are deploying Apollo on a single EC2 instance, which lacks the ability to elastically scale to handle higher levels of concurrency. AppSync GraphQL exhibited performance in the middle between Apollo and REST, 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.

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

4.3 Chapter Conclusions

This chapter's investigation of GraphQL versus REST interfaces for a data-intensive serverless relational database reveals performance characteristics that provide answers to research questions **RQ-3** and **RQ-4**.

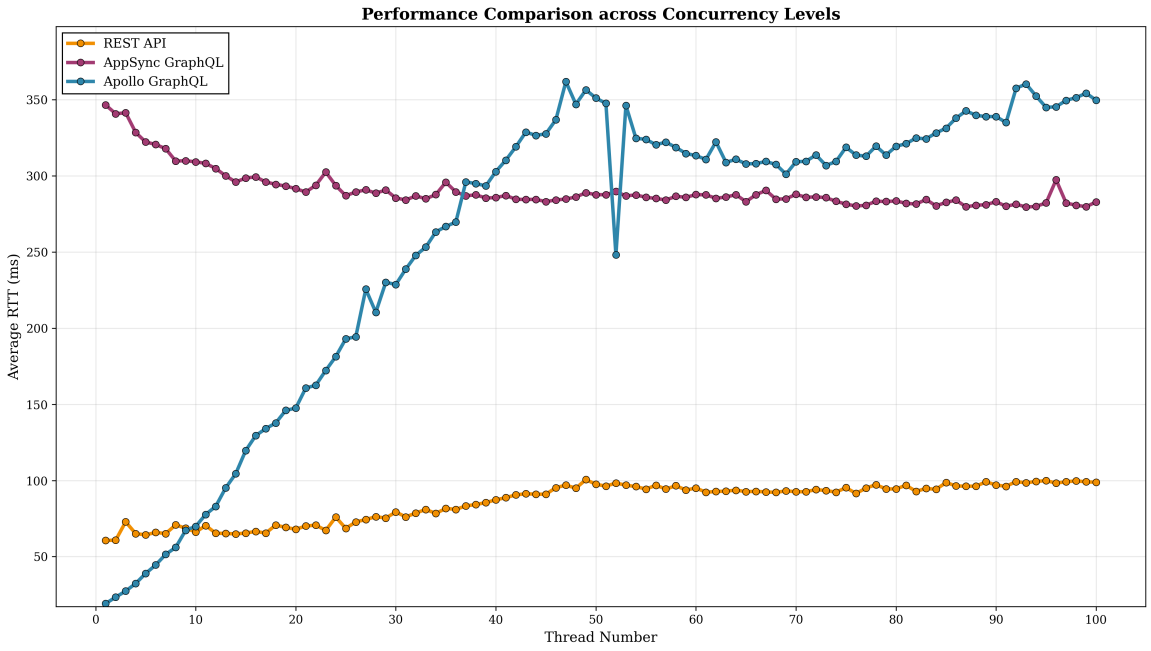

Figure 4.4: Performance comparison across concurrency levels showing REST API's better scalability versus GraphQL implementations. Apollo GraphQL demonstrated the best low-concurrency performance but experienced performance degradation under load.

Regarding **RQ-3**, 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 exhibits exceptional RTT performance advantages of 25-67% over REST across most endpoints and load scenarios, with particularly strong benefits for complex data retrieval operations involving multiple joins. However, this performance advantage was best at low to medium concurrency levels, where Apollo maintains 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.

In addressing **RQ-4** on scalability characteristics, the results reveal a more complex picture than initially anticipated. While Apollo GraphQL demonstrates outstanding low-concurrency performance with response times of 20-45ms, it has significant performance degradation under high concurrent loads, reaching 320-360ms response times. In contrast, REST APIs exhibit better scalability, maintaining consistent 60-100ms response times across all concurrency levels while providing a 65-72% performance advantage over GraphQL implementations at high loads exceeding 80-100 concurrent threads. AWS AppSync's performance falls in between REST and Apollo, exhibiting more stable performance than Apollo under high concurrency while maintaining reasonable response times around 285-290ms.

These 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 remains the more robust option for high-throughput, high-concurrency scenarios where consistent performance under load is important.

Chapter 5

DISCUSSION AND SYNTHESIS

The two studies presented in this thesis, while distinct in their application, provide a complementary and comprehensive view of the trade-offs between GraphQL and REST in serverless environments. This chapter synthesizes the findings from the CPU-bound image processing pipeline (Chapter 3) and the data-intensive relational database (Chapter 4) to answer our four research questions and provide a holistic perspective on the role of API design in serverless applications.

5.1 Performance Gains

This section addresses **RQ-1** and **RQ-3**, which investigate how GraphQL and REST compare in terms of end-to-end performance for the two different workload types. Our findings show that while GraphQL consistently outperforms REST, the reason for its superiority is context-dependent.

For the multi-step, CPU-bound workload (**RQ-1**), the primary performance advantage came from GraphQL's role as a server-side orchestrator. The REST implementation required the client to make several separate network calls to execute the image processing pipeline, accumulating network latency with each step. GraphQL, on the other hand, organizes this complex workflow into a single request, abstracting the logic for the client. This elimination of additional network round trips was the key factor in reducing client-perceived RTT, in a local network environment reducing RTT from around 26s to around 14s, an advantage that was most pronounced for clients on high-latency networks.

For the data-intensive workload (**RQ-3**), GraphQL has more complex performance characteristics than the CPU-bound workload. Apollo GraphQL showed a 25-67% RTT performance gain for database operations especially complex data retrieval operations with multiple joins. However, this advantage was most obvious at low-to-medium concurrency

levels. At high concurrency level, AppSync delivered a higher and more stable performance than Apollo (280ms vs. 350ms and increasing), while REST maintains the lowest RTT.

5.2 Scalability: Managed vs. Unmanaged and REST

This section shows our findings on scalability and architectural trade-offs, addressing **RQ-2** and **RQ-4**. The results from both studies showed us that managed serverless architectures provide better scalability and cost-efficiency.

Our investigation of the data-intensive workload (**RQ-4**) showed that scalability characteristics depend highly on the concurrency level and implementation choices. While Apollo delivered exceptional performance at low concurrency, its performance degraded steadily with increasing load. On the other hand, serverless REST maintained consistent 60-100ms RTT across concurrency levels. This highlights that architectural choice must align with expected traffic patterns and concurrency requirements.

This finding is also reinforced by our comparison of managed and unmanaged GraphQL solutions for the CPU-bound workload (**RQ-2**). The self-hosted Apollo server, despite running on a powerful EC2 instance, hit a performance bottleneck at 53 concurrent requests and was an order of magnitude more expensive to operate at scale than the pay-per-use AppSync service. This demonstrates that even within the GraphQL ecosystem, a managed, serverless implementation could be better aligned with the scalability and cost-efficiency goals of cloud-native applications. Whether comparing GraphQL to REST or to itself, the managed service proves to be the more scalable and economical choice.

5.3 The Holistic View on API Design for Serverless Applications

When viewed together, the answers to our four research questions inform us that GraphQL is a capable and versatile tool for a wide range of serverless applications.

- For complex, multi-step workflows (**RQ-1**), GraphQL excels by simplifying client logic and reducing network overhead through server-side orchestration.

- For high-concurrency data access (**RQ-3, RQ-4**), A managed GraphQL service provides stable yet higher overall RTT compared to traditional REST when not using

GraphQL's aggregation ability, an unmanaged Apollo server performs well at low workload, but degrades heavily when workload increases.

- In terms of a CPU-bound pipeline job, a managed implementation of GraphQL (AppSync) is the most performant, scalable, and cost-effective option, outperforming both self-hosted GraphQL servers (**RQ-2**) and REST APIs.

Ultimately, this work demonstrates for developers building modern serverless applications, the choice of API paradigm is a critical architectural decision that will impact performance and cost. When delivered through a managed service, GraphQL's ability to aggregate data and orchestrate requests directly addresses the performance and scaling challenges inherent in distributed applications, making it a compelling choice for building efficient and robust cloud-native applications.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This thesis conducted a rigorous, two-part investigation into the performance, scalability, and cost of GraphQL versus REST APIs for serverless applications. By examining two distinct and representative use cases—a CPU-bound image processing pipeline and a data-intensive relational database application — our findings contribute specific results to support our guiding research questions.

For our first study on a **CPU-bound workload**, we found that GraphQL offers a more efficient alternative to REST (**RQ-1**). Its ability to act as a server-side orchestrator for a multi-step workflow eliminated multiple network round-trips, reducing client-perceived RTT, especially over high-latency networks. Furthermore, when comparing GraphQL implementations (**RQ-2**), a managed serverless solution like AWS AppSync proved demonstrably superior to a self-hosted server in terms of scalability and cost-efficiency for high-volume workloads. It should be noted that observed performance limitations in a self-hosted Apollo Server can be overcome by horizontal scaling. This is achievable through two approaches: orchestrating instances with Kubernetes or building a distributed system with Apollo Federation.

Our second study on a **data-intensive workload** revealed that an unmanaged GraphQL API delivers better performance and scalability over a traditional REST architecture (**RQ-3**) when serving a low to moderate number of concurrent client requests. Under increasing client load, Apollo Server GraphQL APIs suffered from higher RTT and performance degraded. When scaling to high concurrency (**RQ-4**), REST delivered the lowest RTT, while AppSync GraphQL delivered a stable but higher RTT, and unmanaged Apollo server, a vertical deployment, suffered from lack of elasticity.

Collectively, these findings demonstrate that GraphQL is not just a viable alternative

to REST, but a compelling alternative for building modern serverless applications. Its architectural flexibility directly addresses key performance challenges, from simplifying client orchestration in complex workflows to enabling highly scalable data access. The architectural choice however, should be made based on actual application requirements and client load patterns. Design decisions should be made carefully with good understanding of workload and traffic patterns.

6.2 Future Work

Building on the findings of this thesis, several paths for future research has potential to further deepen the understanding of API design in serverless environments.

- **Broader Workload and Use Case Analysis:** This thesis focused on two distinct archetypes (CPU-bound and data-intensive). Future work can investigate more complex, hybrid workloads that combine both multi-step processing and intensive database queries within a single request or other serverless application use cases. In particular, performance investigation of aggregate database queries, where GraphQL can combine multiple data results into a single client-server round-trip should be contrasted with REST API equivalents.

- **Performance of Advanced GraphQL Features:** Our study concentrated on the performance of core GraphQL queries. Future work could also evaluate the performance and cost implications of more advanced GraphQL features like data streaming in a serverless context.

- **Multi-Platform and Cross-Cloud Benchmarking:** The experiments in this thesis were conducted within the AWS ecosystem. To generalize the findings and provide a more complete view, future research could include other cloud platforms like Google Cloud and Microsoft Azure.

- **Apollo Federation and Kubernetes:** Future work can expand on the investigation of a static Apollo Server deployment by investgating a Kubernetes cluster based

deployment of Apollo Server with failover and horizontal scaling. Future work can also investigate Apollo Federations. An Apollo Federation enables combination of multiple APIs into a single federated GraphQL API. An Apollo Federation serves as an API orchestration layer, where clients make a single GraphQL request to a single entry point called the router. The router intelligently orchestrates and distributes the request across a set of GraphQL APIs and returns a unified aggregated response providing a mechanism to distribute computation across multiple Apollo servers.

Pursuing these future directions can extend our comparison of REST vs. GraphQL for additional use cases and scenarios to offer a more comprehensive guide to designing highly performant API architectures in the evolving landscape of serverless computing.

BIBLIOGRAPHY

[1] CMS Open Payments — openpaymentsdata.cms.gov. `https://openpaymentsdata.cms.gov/`.

[2] Amazon api gateway. https://aws.amazon.com/api-gateway/, 2024.

[3] Apollo graphql. `https://www.apollographql.com`, 2024.

[4] Aws appsync. `https://aws.amazon.com/appsync`, 2024.

[5] Aws lambda. `https://aws.amazon.com/lambda`, 2024.

[6] Aws sdk for python. https://aws.amazon.com/sdk-for-python/, 2024.

[7] 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.

[8] How netflix scales its api with graphql federation. https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2, 2024.

[9] Rest apis' exhaustion signs. `https://www.programmersinc.com/over-fetching-and-under-fetching-rest-apis-exhaustion-signs`, 2024.

[10] 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.

[11] Asma Belhadi, Man Zhang, and Andrea Arcuri. Evolutionary-based automated testing for graphql apis. In *Proc of the Genetic and Evolutionary Computation Conf Companion*, pages 778–781, 2022.

[12] Alan Cha, Erik Wittern, Guillaume Baudart, James C Davis, Louis Mandel, and Jim A Laredo. A principled approach to graphql query cost analysis. In *Proc of the 28th ACM Joint Meeting on European Soft Eng Conf and Symposium on the Foundations of Soft Eng*, pages 257–268, 2020.

[13] Xinghan Chen, Ling-Hong Hung, Robert Cordingly, and Wes Lloyd. X86 vs. arm64: an investigation of factors influencing serverless performance. In *Proceedings of the 9th International Workshop on Serverless Computing*, pages 7–12, 2023.

[14] Sijin Cheng and Olaf Hartig. Lingbm: A performance benchmark for approaches to build graphql servers (extended version). *arXiv preprint arXiv:2208.04784*, 2022.

[15] Robert Cordingly, Sonia Xu, and Wes Lloyd. Function memory optimization for heterogeneous serverless platforms with cpu time accounting. In *2022 IEEE international conference on cloud engineering (IC2E)*, pages 104–115. IEEE, 2022.

[16] Olaf Hartig and Jorge Pérez. Semantics and complexity of graphql. In *Proc of the 2018 World Wide Web Conf*, pages 1155–1164, 2018.

[17] Dewi Ayu Hartina, Armin Lawi, and Benny Leonard Enrico Panggabean. Performance analysis of graphql and restful in sim lp2m of the hasanuddin university. In *2018 2nd East Indonesia Conf on Computer and Information Technology (EIConCIT)*, pages 237–240. IEEE, 2018.

[18] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. Automatic property-based testing of graphql apis. In *2021 IEEE/ACM Int Conf on Automation of Soft Test (AST)*, pages 1–10. IEEE, 2021.

[19] Armin Lawi, Benny LE Panggabean, and Takaichi Yoshida. Evaluating graphql and rest api services performance in a massive and intensive accessible information system. *Computers*, 10(11):138, 2021.

[20] Eunggi Lee, Kiwoong Kwon, and Jungmee Yun. Performance measurement of graphql api in home ess data server. In *2020 Int Conf on Information and Communication Technology Convergence (ICTC)*, pages 1929–1931. IEEE, 2020.

[21] Georgios Mavroudeas, Guillaume Baudart, Alan Cha, Martin Hirzel, Jim A Laredo, Malik Magdon-Ismail, Louis Mandel, and Erik Wittern. Learning graphql query cost. In *2021 36th IEEE/ACM Int Conf on Automated Soft Eng (ASE)*, pages 1146–1150. IEEE, 2021.

[22] Sabah Mohammed, Jinan Fiaidhi, Darien Sawyer, and Mehdi Lamouchie. 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*, pages 52–60, 2022.

[23] Sri Lakshmi Vadlamani, Benjamin Emdon, Joshua Arts, and Olga Baysal. 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)*, pages 10–17. IEEE, 2021.

[24] Andrea Vázquez-Ingelmo, Juan Cruz-Benito, and Francisco J Garcí a Peñalvo. 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, pages 1–8, 2017.