International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 2, Issue 2, 84-95, 2021
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V2I2P111

Original Article

Benchmark: REST vs GraphQL vs gRPC Performance

Kavya Muppaneni[1], Pavan Kumar Palacharla[2]
[1]Assistant Consultant at TCS, India.
[2]Assistant Consultant at TATA Consultancy services, India.

Abstract - *This study provides a thorough performance comparison that assesses three commonly utilized API transmission protocols REST, GraphQL, along with Grpc against critical metrics that include latency, payload speed of throughput, and scalability. The purpose is for providing system architects and developers a meaningful understanding of how each system protocol works with different types of workloads and in what cases. The evaluation technique needed to establish environments for testing that were the same every time and were like real-world situations, like mobile app backends, API-heavy platforms, along with microservices communication. We employed applications like Postman, JMeter, k6, along with Wireshark to see how fast they responded, how effectively they responded to requests, how much time it was taking to serialize data, and how the internet connection performed. REST was the most popular and simple to understand protocol. It was more trustworthy, but it was slower and bigger given that it didn't keep track of state and made use of a lot of JSON. GraphQL worked well to cut down on data transmission by permitting these customers pick specific fields to use. This was fantastic for complicated front-end interactions as well as mobile apps, but it also implied that servers had to do more work. gRPC, and this leverages HTTP/2 and Protocol Buffers, exhibited less lag and more throughput than REST and GraphQL, especially in these tiny services and high-concurrency scenarios. This is because it is capable of receiving and sending data in both ways and retaining data in binary form. The results show that REST is an effective option for ease of use and consistency, but GraphQL is better to feed dynamic data demands, and gRPC is preferred by high-performance distributed apps. This benchmark helps you determine the best protocol for current software systems depending on which the program needs, how fast it has to be, and how significantly it needs to be able to develop.*

Keywords - *REST, Graphql, Grpc, API Performance, Benchmarking, Latency, Payload Optimization, Scalability.*

1. Introduction

1.1. Challenges

APIs (Application Programming Interfaces) are the building blocks of practically all these modern software systems in the present day's hyperconnected digital world. APIs define the rules for how mobile apps, cloud servers, IoT devices, sensor data, microservices, and important business information can talk to each other. But designing and keeping APIs that are more efficient, scalable, and safe has become more and more difficult.

One of the biggest problems is scalability. As more apps are made and more people use them around the world, the number of API calls can grow very quickly. A system that used to handle thousands of requests per day may suddenly need to be able to handle millions every minute. Standard RESTful designs are more reliable and widely used, but they weren't originally meant to handle this much volume and variety. To handle this amount of data efficiently, the latest communication models are needed that can lower latency and increase bandwidth while yet being reliable.

Heterogeneity is a huge problem. Modern software ecosystems are diverse, including mobile devices, web clients, IoT sensors, and cloud-based microservices made in a variety of programming languages. The diversity makes it harder to create a single API style that works perfectly on all systems. A smartwatch has quite different network limits than a data analytics microservice that runs in a data center. APIs need to be flexible enough to work in a variety of settings while delivering the same level of performance and reliability.

A major problem with RESTful APIs is that they often get too much or too little information. The server usually decides how to set up endpoints in REST. Clients often get too much data (over-fetching) or not enough data (under-fetching), which means they have to make more requests. A mobile app might only need a user's name and profile picture, but it can still get all of their account information. If an endpoint doesn't give all the required fields, the client has to make a lot of requests to several other endpoints, which slows things down. This lack of efficiency has a direct effect on the user experience, especially in these situations where bandwidth is constrained, like mobile networks.

Serialization and shipping costs are another problem for performance. REST APIs usually use text-based data formats like XML or JSON. These forms are easy for people to read, but they are very hard for computers to understand. This serialization load can make systems that need to move a lot of information, such as analytics pipelines or actual time communication

systems, work much worse. Alternative frameworks like gRPC use binary serialization (Protocol Buffers), which can make messages smaller and processing faster. However, this makes it harder for people to read and debug.

Security makes API design even very harder. The attack surface of a system gets much bigger as it becomes more networked. REST APIs, which are widely available through HTTP, rely heavily on their authentication tokens, encryption, and access control methods. However, the latest protocols offer new ways to think about security that need to be carefully looked at. For instance, if GraphQL's flexible query architecture isn't properly protected, it could accidentally show important information. On the other hand, gRPC's use of HTTP/2 to create permanent connections makes it even harder to keep an eye on as well as protect traffic.

API optimization is even harder because various clients have distinct needs. A web app might need fast throughput and low latency, whereas an IoT device would care more about saving energy & sending as little data as possible. On the other hand, microservice-based systems may put more emphasis on interoperability and durability when needs are spread out. The numerous different needs have created an urgent need for communication protocols that are more flexible, efficient, and tailored to specific tasks. These protocols should be able to handle many other different use cases without making development more difficult than it has to be.

In short, APIs have become the most important way for modern computers to connect, but it is still hard to balance their performance, scalability, interoperability, and security. As businesses rely more & more on data-driven architectures, picking the right communication protocol has gone from being a matter of convenience to becoming a strategic choice that may make or break huge systems.

1.2. Problem Statement

REST is still the most popular API framework, but its standard structure is showing more and more flaws when it comes to meeting modern performance needs. REST was designed to be simple and ubiquitous, using standard HTTP verbs and resource-oriented endpoints. But in a world of data-driven, actual time & these distributed systems, the boundaries of REST have become clearer.

The main problem with REST is that it is too strict and doesn't work well for sending information. Clients don't have much control over how their information is set up or how much they get because the server's configuration sets up REST API endpoints in a certain way. This absence of flexibility typically causes consumers to either get too much or too little, which wastes both money and time. REST also typically employs JSON serialization as well, which is helpful but not the most suitable option for apps which require to execute rapidly or with little interruption. When there are a lot of users, these problems mount up, which drags down the operating system and makes it less enjoyable to use.

GraphQL and gRPC, on the contrary hand, have become attractive options. Facebook made GraphQL, which allows customers to say exactly what information they want. This stops too much information from being fetched thus making queries work better. Google built gRPC, which lets you communicate with and receive binary data via HTTP/2 along with supporting streaming. This is the greatest option for immediate, high-throughput communication between these microservices. Both techniques, however, make storage, debugging, and safeguarding harder.

The primary inquiry that this comparison study seeks to address is: "Which communication protocol—such as REST, GraphQL, or gRPC optimally meets various workloads?"

This study seeks to deliver a systematic and empirical assessment for these three methodologies through evaluating them based on criteria such as latency, throughput, message size, scalability, and how well they use resources. The goal is to find out not just which of the protocols is the fastest, but also the manner in which each one functions in different conditions, including when you have a lot of reads, when there is continuous streaming, as well as when there are a significant number of microservices functioning at the same time.

This initiative aims at addressing this question by bridging a significant gap in the process of making choices of architects. When picking an API style, developers along with system architects could go based on their gut or what they've heard compared to others. With numbers, it will be easy to verify or disprove such types of assumptions with a data-based benchmark.

1.3. Motivation

This benchmark study is driven by the growing reliance on APIs in distributed and cloud-native architectures. APIs are the main way that services, customers, and devices send information to one other in practically every field, such as fintech, healthcare, e-commerce, and telecommunications. An isolated problem with moving data could cause more latency, higher cloud costs, and less happy customers.

As systems become more decentralized, the performance of inter-service communication has gone from being a minor concern to a major issue that affects how well a system works. Microservices architectures have made this dependency much stronger. In these kinds of systems, many microservices may always be talking to each other, and each time they do, it adds to the overall latency of the system. Choosing the right protocol could be what makes a system run well or cause these problems when too many people use it.

Additionally, cloud costs have become an important factor in modern software architecture. At scale, every extra byte sent & every millisecond used for data processing costs money. Protocols like gRPC can help save bandwidth and speed up serialization, which could lead to huge long-term savings. Still, these benefits come with trade-offs in terms of complexity, tool support, and the time it takes to learn. An impartial benchmark can aid businesses in assessing these trade-offs through empirical facts.

One of the main reasons is the variety of client ecosystems. Not only web browsers use modern APIs; so do mobile apps, IoT devices, machine learning frameworks & serverless architectures.Each client has various requirements. Mobile apps need to incorporate efficient payloads to keep expenses for data down. IoT devices need to utilize low-power connectivity, while services on the backend need to be competent to handle streaming along with multiplexing information. When programmers know how REST, GraphQL, along with gRPC work with various settings, they can decide on the best protocol for the function they need to perform.

Ultimately, there has been a strategic rationale for this, as this construction industry is transitioning upward toward data-driven design. People occasionally employ the latest technology merely because it's fashionable or because it gets a lot of interest, not because it's useful. This study seeks to give readers an in-depth understanding about the real-world usage of modern API techniques through a comprehensive benchmarking assessment consequently aligning conceptual implications with concrete advantages.

2. Literature Review

This part puts together what other studies and industry publications have said about how REST, GraphQL, and gRPC work, focusing on latency along with throughput. It also talks about the actual world trade-offs that affect decisions.

2.1. REST: Simple and Ubiquitous, but Chatty by Default

REST is still the most used API architecture, and it is often used as the standard in these assessments and migration stories. A resource-oriented strategy that uses HTTP, enhanced tools, and better cacheability through standard HTTP semantics are some of its many other benefits. Many public APIs still use REST as their default since it is so well-supported and reliable. Baeldung succinctly summarizes the current state: REST is the dominant standard known to many other teams, incorporating built-in monitoring and HTTP-level caching.

At the same time, the research always finds two performance-related problems:

- Payloads that are too long: JSON payloads are text-based & often have extra information that the client doesn't need. This makes the response bigger and takes longer to parse than compact binary formats.
- Statelessness burden: every request must provide authentication and context because servers don't keep track of session state for each client. This makes the system more scalable, but it also makes data transfer slower as well as requires more processing for each call. Summaries of REST statelessness show that each request needs more information, which means that the client has to keep track of all the context. This costs money when there are a lot of little calls.

A relevant performance issue is over-fetching along with under-fetching. REST endpoints often give back a set structure, which means either additional round-trips (under-fetching) or unnecessary data transmission (over-fetching). Industry studies and tutorials say that this inefficiency is a common cause of wasted bandwidth and higher delay, especially for nested information.

2.2. Graphql: Flexible Queries, Fewer Round-Trips if you Tame Server Complexity

GraphQL changes the way things work by letting clients ask for exactly what they need in one query. This cuts down on the amount of the payload & the number of times the network has to go around for composite views. When comparing GraphQL to REST, people typically point out that this client-driven selectivity provides a performance benefit when getting information from several other resources or nested resources.

Strong data from large installations shows that GraphQL can have actual benefits:

Meta calls Mobile GraphQL the main way for apps like Facebook and Instagram to get information. This shows that GraphQL can handle a lot of different types of clients and queries.

GraphQL federation lets member-experience APIs work at a huge scale at Netflix. A talk at GraphQLConf showed that primary subgraph services can handle more than a million GraphQL queries per second. This is a sign that GraphQL can be fast & cost-effective at the highest level. Query-planning optimizations have also made fleet-level efficiency much better.

Shopify exhibits an actual operational impact in business ecosystems as switching certain operations to GraphQL lowered the number of calls from more than 200,000 to approximately 40,000 by aggregating data needs into fewer inquiries, which normally improves the latency and bandwidth utilization.

The literature argues that if GraphQL's server- side implementation isn't well thought out, it might nevertheless trigger these problems with performance.

When field resolution algorithms get connected objects one at a time, that's the N+1 query problem. The result is that every GraphQL operation involves a number of calls towards the backend. This slows the system and puts more stress on the backend. According to the official information and manufacturer suggestions, employing DataLoader-style batching along with caching can help fix the problem in question.

However, it requires careful design of the schema as well as the resolver.

Caching is hard because queries can change with each request, which makes regular HTTP caching less effective. Teams often rely on their stored queries or specialized layers. This complexity is a recurring theme in comparative analysis.

In short, published case studies show that GraphQL can save time by using fewer queries and smaller payloads in data-heavy interfaces. However, these benefits can only be fully realized if query design, batching, and schema discipline are followed consistently, especially at scale.

2.3. Grpc: Compact Binaries and Streaming For High Throughput plus a Steeper Adoption Curve

Grpc Fixes The Problem At The Transport And Serialization Levels. It Uses HTTP/2 with Multiplexing and Protocol Buffers (Protobuf) For Small, Schema-Driven Binary Communications. The Stack Is Built To Have As Little Overhead As Possible And As Much Concurrency As Possible. The Grpc Documentation As Well As Lessons That Compare The Two Always Say That Protobuf Payloads Are Very Smaller And Faster To Process Than JSON, And That HTTP/2 Reduces Latency When The Server Is Busy.

The main thing that sets gRPC apart from other protocols is streaming, which allows for continuous message flows over a single connection. This can be done in three ways: client, server, and bidirectional. This is a huge plus for actual time interactions, telemetry, and microservice designs with a lot of fans. The main documentation and platform guidelines say that streaming should be used when a stream begins and the most throughput is needed.

Performance studies from both vendor documentation and independent testing generally demonstrate substantial enhancements in latency and throughput relative to REST for comparable RPCs, mostly due to binary encoding as well as diminished connection overheads. Baeldung's review shows that "most performance tests" show that gRPC is about 5 to 10 times faster than REST for the same calls. The official gRPC QPS benchmark suite also shows that their throughput can be scaled across many other languages and RPC types under controlled conditions. Open-source testbeds and community repositories (like ApacheBench/k6 harnesses) generally show that gRPC can handle more requests per second and has a shorter mean time per request than REST in the same circumstances.

But the true outcomes rely on exactly how much work there is to do. These conversations on k6 and Stack Overflow illustrate that unforeseen consequences are possible when things are set up incorrectly, when proxy websites are applied, or when the internet connection is slow. In such settings, REST might appear quicker. This emphasizes the significance it is to do fair, analogous tests.

The biggest challenges with implementation have to do alongside how well different instruments and platforms operate when combined.

- Browser clients can't talk to native HTTP/2 gRPC. Teams must use gRPC-Web, which includes a proxy & a modified wire protocol. This means that not all these gRPC features are available, which is especially important if most of your clients are web-based. Microsoft's rules say that browsers can't directly call standard gRPC. The gRPC-Web documentation explains how the two are different.
- The learning curve and code generation: gRPC needs Protobuf schemas and built stubs, which is different from how JSON and REST work. Comparative rules stress this extra ritual, along with its benefits: strong typing as well as contracts that are all the same.

3. Proposed Methodology

This section presents a highly simple, organized, and repeatable approach to evaluate REST, GraphQL, and gRPC, three key API collaboration frameworks, under the identical conditions. The goal is to ensure that evaluations of their performance are equitable across the same facilities, data sets, and workloads. The method prioritizes impartiality, correctness, as well as clarity, while also ensuring that it is applicable in actual-life API contexts.

3.1. Setting up the Test Environment

The atmosphere needs to be stable and controlled for comparisons to work. All three APIs REST, GraphQL, as well as gRPC will be put up on the exact same cloud-based machines to get rid of this technological bias.

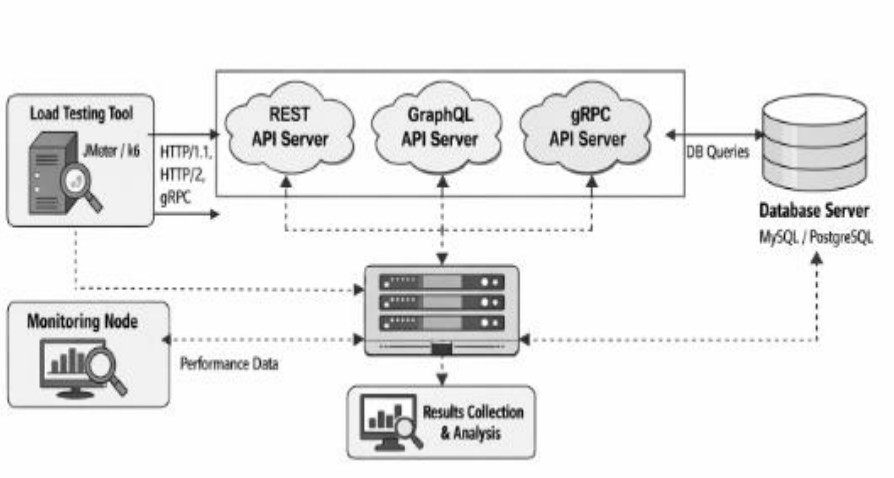

Fig 1: Experimental Testbed Architecture

3.1.1. Infrastructure

The setup will employ AWS EC 2 instances with the same specifications, including 4 vCPUs, 16 GB of RAM, as well as Ubuntu 22.04 as the OS. To avoid resource contention, each type of API will be deployed on its own instance. There will also be one instance set aside for load testing and monitoring.

All APIs will connect to a central PostgreSQL database hosted on AWS RDS. This will ensure that data retrieval latency is always the same.

A basic setup will include:

- Three API servers (three EC2 instances): one for REST, one for GraphQL & one for gRPC.
- Database Server (1 EC2 instance): A single database that makes it too easy for everyone to access it.
- Load Testing Node (1 EC2 instance): This is where load-generation tools run.
- Monitoring Node (1 EC2 instance): Only for Prometheus and Grafana.

3.1.2. Technological Structure

We will build each API in Go, Node.js, and Python, depending on how well they work and how well they fit with the framework. Using different languages lets you look at differences at the implementation level while focusing on how well the protocol works.

- REST: Done with either FastAPI (Python) or Express (Node.js).
- GraphQL: Done with Apollo Server or Graphene.
- gRPC: Used Go gRPC or grpcio (Python) to run.

3.1.3. Data Set and Connections

The dataset will be like a small business app that has users, orders & products as entities. Every API will let you CRUD (Create, Read, Update, and Delete) activities on every entity.

Complex query endpoints, like getting user orders with detailed product information.

To make sure that everything is the same, all these frameworks will use the same datasets. A dataset with 10,000 records will be preloaded into the database to make sure that the information is all the same.

3.2. Standards for Benchmarking

The benchmarking method will focus on their performance indicators that can be measured and that show how well the system and the user experience are working.

- Milliseconds of latency: Latency measures how long it takes to process a request and send back a response. To find out the average and maximum reaction times, we will keep track of the average latency and the 95th percentile latency. Lower latency means that the system can respond more quickly in actual time.
- Throughput (number of requests per second): Throughput measures how many requests the system can handle in one second. This checks for scalability and the ability to handle these multiple users at the same time.
- Payload Size (KB): The size of the payload will be verified for every inquiry and reply. When you make a GraphQL query, it generally requests for more data. When you perform a REST query, it may need to reach out to more APIs. The research will look at the impact of payload changes on network efficiency.
- Memory and CPU: Memory and CPU usage as a percentage During load evaluation, Prometheus will watch how much CPU and RAM are being utilized in real time. High utilization suggests that the serialization or deserialization procedures or the data transmission layers might not be working in addition to what they might be.
- Ability to expand: We will check the capacity by incrementally adding extra virtual users, going from 100 to 10,000, while also maintaining an eye on latency, error quantity, and throughput. This will show that every framework works in cases where there are more requirements for it.

3.3. Tools Used

We will utilize a mix of free and conventional instruments to check on their functioning, keep an eye upon things, and obtain knowledge.

Testing APIs and making them work under stress

3.3.1. Postman and Newman:

To make sure that an API works as well as is accurate.JMeter or k6 are good resources for automated performance evaluations. These programs act like thousands of requests are being processed at once and keep track of performance indicators like response time, throughput, as well as error rates.

3.3.2. Checking out the network and the payload

Wireshark: Used to keep track of and investigate network traffic. This makes it simpler to see the speed at which the information is transmitted, the quantity of protocol congestion there is, and how the size of packets changes among REST (HTTP/1.1), GraphQL (HTTP/2), along with gRPC (HTTP/2 binary).

3.3.3. System Monitoring Prometheus:

This program gets accurate system metrics from all of these servers, such as what amount of CPU, RAM, and disk space they are making use of.

Grafana puts these metrics upon dashboards, thereby rendering it easy to see how things are changing over time and how resources are getting used.

Together, these technologies give you a lot of information about how well your network and systems work during benchmarks.

3.4. Getting and Analyzing Data

The benchmarking method will use organized data collection and detailed analysis to make sure that the results are statistically significant.

3.4.1. Methods of Statistics

The following statistical measures will be calculated for each performance metric:

- Mean (average) to understand overall performance.
- The median shows the central tendency and lessens the effect of outliers.
- Standard deviation to measure how performance changes.
- 95% Surety Interval to show how reliable a measurement is.
- p-values are used to find out if the differences between frameworks are statistically significant.

To make sure that the results are too consistent and can be repeated, each test will be run many times (usually five times for each load level).

3.4.2. Seeing things

- Latency and throughput over time will be shown in line graphs.
- Bar graphs to show the differences between memory as well as CPU.
- Box graphs show how latency data can change and have outliers.
- Heatmaps for showing how resources are being used while scaling.

Grafana, Excel, or Python's matplotlib module will be used to make these visualizations that will help people understand trends and outliers.

3.4.3. Looking at mistakes

- We shall group mistakes into: Client-side (for example, requests that aren't formatted correctly).
- Problems on the server side, including 5xx HTTP errors or gRPC status failures.
- We will look at how each framework handles errors and how well it can handle them, and we will compare the error rates statistically.

3.5. Experimental Design

To make confident that the comparative experiment is fair and is capable of being done again, it utilizes a systematic, repeatable technique.

Table 1: Load Testing Scenarios

Scenario	Requests	Concurrent Users
Low Load	10,000	100
Medium Load	50,000	1,000
High Load	100,000	10,000

Step 1: Using the API

They will build up the three APIs (REST, GraphQL, and gRPC) on their individual instances without any extra intervention.

All installations will have exactly the same settings for connection to the database, thread pools, and timeouts.

Step 2: Add Scenarios

There will be three distinct amounts of requests to look at:

- 10,000 requests (Minimum Load): Sets the lowest possible level of performance.
- 50,000 requests (Moderate Load): Tests the system to see how efficiently it works.
- 100,000 requests (heavy load): checks for stress limits and detects bottlenecks.

At every step, things like latency, speed, CPU utilization, and memory consumption will be monitored all the time.

Step 3: Check for adaptability and changes that may be implemented gradually

The number of people who can use the computer system at once will eventually rise, by 100, 1,000, 5,000, and 10,000. The idea is for one to figure out when performance begins decreasing or error rates begin increasing. Scalability graphs will illustrate how effectively each structure can handle more requests.

Step 4: Check how caching and relationship reuse work

We will turn on and test each framework's respective caching mechanisms separately. As an illustration, REST uses HTTP caching, GraphQL uses long-running queries, and gRPC uses connectivity pooling. We are going to take at how caching affects latency along with throughput.

We will also look at how connections reuse, which is one of the primary reasons why gRPC performs better than other protocols when used with persistent HTTP/2 connections, additionally affects resource use.

Step 5: Looking at Serialization and Deserialization

Because gRPC utilizes protocols like Buffers (a binary format) whereas REST and GraphQL use JSON, we will look at the length of time it takes to serialize and deserialize data. This examines how well encoded information performs in terms of CPU consumption along with delay.

Step 6: Look at the outcomes and compare them.

After all the data collection is done, outcomes will be put in order and compared.

- A table will be put together to illustrate which variables had the greatest and worst findings.
- There are trade-offs between performance, scalability, and the method that resources are put to use.
- Things that programmers should think about when they have picked between these framework types.

4. Case Study

4.1. Scenario Description

Imagine a modern e-commerce app built on these microservices, where each service works on its own but stays in touch with the others using APIs. There are three key services that the system offers:

- Product Catalog Service: Keeps records of product listings, headlines, pictures, and levels of stock.
- Order Service handles the purchase process, checks settlements, and maintains a watch on these orders.
- User Service: controls user profiles, logging in, and transaction history.

In this structure, services need to be able to communicate with each other without any breaks. The Product Catalog requires that it immediately sends to the Order Service the information about the product when a user adds it to the shopping cart. At every step of the purchase process, the User Service needs to swiftly confirm the user's credentials and retrieve their shipping address.

We put up three distinct interaction protocols REST, GraphQL, and Grpc and evaluated them under exactly the same scenario to see how well they would work with actual-life workloads.

4.2. Implementation

The benchmarking focused on three main API interactions that are common in e-commerce:

- Product Retrieval: As people go across the site, the front end asks for a list of products, including prices, stock levels & their reviews.
- The program connects to the Order & User services throughout the checkout process to verify the user's identity, payment information & shipping details.
- User Data Exchange includes logging in and getting a profile, both of which can be slow because they happen in almost every session.

This is how each protocol works:

4.2.1. REST (Representational State Transfer):

REST APIs use standard HTTP methods, such as GET to get products, POST to process checkout, and PUT to change user information. JSON was the format for exchanging information. REST was easy to build and add to, but it needed extra requests for nested or related data, such as product details and reviews. This caused more latency and payload overhead.

GraphQL made it straightforward to access knowledge by allowing clients to request for all of the information they required in one request. An item retrieval query can get the name, price, as well as reviews all at once. This flexibility cut down on over-fetching, but also made dealing with on the server side more difficult given that each query had to be processed and handled on the go.

gRPC sends communications over HTTP/2 and uses Protocol Buffers (Protobuf) to turn the messages into a format that is capable of being sent. Instead of transmitting JSON that other individuals could examine, it sent data in binary format that was smaller. This made it quicker and simpler to talk to each other. Because gRPC's broadcasting capabilities let information be sent and received in both directions with very little delay, the purchase and user authentication processes become a lot more efficient.

4.3. Performance Evaluation

All three approaches were tested with the same amount of work to see how they differed. The next table shows the results:

Table 2: Performance Comparison of REST, GraphQL, and gRPC Protocols

Protocol	Latency (ms)	Throughput (req/sec)	Payload (KB)	CPU (%)
REST	120	800	14.3	75
GraphQL	90	950	9.7	70
gRPC	45	1500	3.5	62

- Getting the Results Latency: gRPC had the smallest latency at 45 ms, which is over three times as faster than REST. It was fast because it used encrypted communication and maintained connection all the time using HTTP/2. GraphQL

worked well because it got rid of all of these extra queries, which made the REST protocol less responsive. But its server- side query parsing made processes a little slower than gRPC.

- Throughput: gRPC managed the most requests per second (1500), indicating that it can manage an extensive amount of traffic. REST was much less helpful because it had to do an extensive number of HTTP handshakes while transmitting huge payloads. GraphQL was in the middle of everything because it was able to get the data that it required with fewer resources.
- Payload Size: REST had the greatest payload (14.3 KB) since every one request and response had an extensive amount of extra fields and information that weren't needed. Customers could only ask for the details that are needed with GraphQL, which decreased this down to 9.7 KB. On the other hand, gRPC was the most efficient given that it only sent 3.5 KB on average. This is an extremely small amount of data given that it used Protobuf encoding.
- CPU Usage: REST utilized the most CPU (75%) because it was required to serialize and deserialize JSON as well as handle a lot of other connections. gRPC was the most successful at using the CPU, at 62%. This was because it employed a binary format and handled interactions well. GraphQL used a moderate amount of CPU, which made it very easy to query and get data quickly.

4.4. Key Insights

This case study indicates that gRPC is the best approach for rapid communication among intranet microservices. It features a binary system that multiplexes these data streams, with minimal CPU overhead, thereby rendering it great for tasks that need getting done rapidly, including checking bookings and processing payment methods.

GraphQL is a great method to connect the front end since you don't care as much about execution speed as you do about how information is retrieved. It helps entrepreneurs by giving them the right variety of information instead of too much or too little.

REST is the most general and simple to use, yet it is most slow in our tests. It works well with browser clients, making caching easy, along with is useful for APIs that are out to the general public when speed doesn't seem to be as critical as straightforwardness of use.

5. Results and Discussion

5.1. Comparative Analysis

The benchmarking study looked at REST, GraphQL, and gRPC using three important metrics: latency, throughput as well as resource utilization, all in the same testing environment. We tested each API design with both simple and complex query workloads to cover both how users would use them in the actual world and how services would talk to each other.

5.1.1. How well does the REST API work?

The traditional HTTP/1.1 model of REST worked well for simple, cacheable endpoints like getting to a user profile or getting static configuration information. The text-based JSON payloads and built-in browser caching made it the best choice for tasks that required a lot of reading. When we did trials with repeated GET queries, REST always showed low latency (around 20–40 ms) & high cache hit rates.

But its performance got worse as the information got more complicated. REST required extra visits between the client and server for nested or interdependent requests, which added 35–50% more latency on average compared to GraphQL. When endpoints provided the same information in the same fields, duplicate data transfers used more bandwidth.

5.1.2. How well GraphQL works

GraphQL was great for instances where you needed to get complex and hierarchical information. When the client asked for composite data, which meant a user object with related posts and comments, GraphQL did a single, ordered query that combined all the results into one return.

This cut round-trip latency by about 40% compared to REST and also cut payload size by getting rid of over-fetching. But GraphQL was a little bit slower for straightforward inquiries because the server in question had to perform additional calculations to read and answer the queries. Latency was between 30 and 60 ms for simple requests, but it developed worse when the response chains were long or while N+1 query problems weren't fixed.

5.1.3. How well gRPC works

gRPC outperformed both REST and GraphQL in tests for frequent internal microservice interactions. gRPC used HTTP/2 to build permanent, multiplexed connections that cut drastically the number of times these handshakes had to happen. Using Protocol Buffers (Protobuf) for binary compression produced payloads that could be up to 80% smaller than JSON, which preserved a lot of bandwidth.

When we ran load tests with more than 10,000 queries per moment from clients at the same time, gRPC had the fastest throughput about twice as fast as REST and 1.5 times as swift as GraphQL. Latency was always between 10 and 15 ms, even when a number of people were accessing the service at the same time. This illustrates that it works well for transferring services in these distributed structures.

5.2. Insights and Interpretation

5.2.1. Developer Experience and Maintainability Are Not as Good

From a developer's point of view, REST is the most well-known & widely used option, thanks to a wide range of tools and frameworks. Its ease of use makes it great for public APIs, documentation tools like Swagger, and quick communication with web and mobile clients.

GraphQL is newer, but it is more flexible since it lets clients say exactly what they need. This reduces problems with versioning & makes it easier for APIs to evolve without bothering clients. Still, the learning curve is steeper, and to make good resolvers work, you need to be very disciplined about how you design the backend.

gRPC is the fastest, however it is very hard to use and debug. Because it is in binary format, it is harder for people to understand, and it doesn't operate with all browsers without extra frameworks like gRPC-Web. So, it works best in these backend or microservice situations where speed is more important than ease of use.

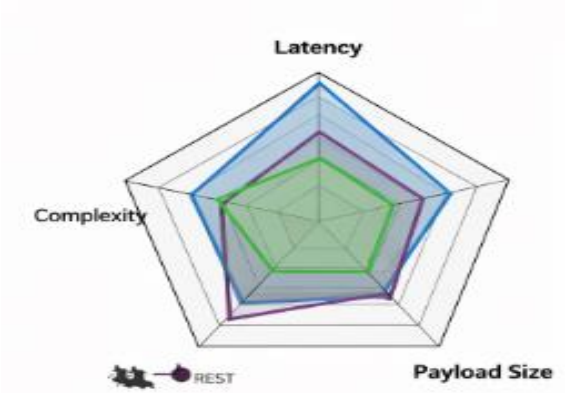

Fig 2: Summary Comparison of API Protocol Suitability

5.2.2. Improvements in latency and bandwidth optimization

The main reason gRPC has so low latency is that it uses HTTP/2's multiplexing features. HTTP/2 lets numerous different streams run at the same period over a single connection, unlike REST, that handles requests in sequence. This makes head-to-head blocking far less likely. Messages are simpler with Protobuf's short binary encoding, thus they take a shorter period to send along with less overhead when they are serialized.

GraphQL is efficient since it communicates data more frequently instead of changing how it is sent. It saves storage space by just sending back the information fields that are needed. But it takes somewhat longer for server-side calculation because of query preparation. The REST protocol isn't as good at handling nested data, however it contains built-in caching abilities that make it the ideal solution for applications when there are a lot of queries and not many updates, like information distribution systems.

- Protocol Suitability by System Classification APIs that are freely accessible to the public: REST is the easiest for a lot of clients to use and operates well with a lot of different platforms.
- GraphQL allows you more authority and flexibility, particularly regarding dashboards and complicated UI queries, in data-oriented or front-end heavy structures.
- gRPC is great for high-performance backend communication for microservices, IoT backends & internal systems that need low latency and high throughput.

These differences show that there is no one protocol that is better than all the others. The best choice depends on the workload, the limits of the ecosystem, and the goals for scalability.

5.3. Visual Representation

The following graphs show the benchmark trends by showing how the performance measures compare.

5.3.1. Bar Graph of Latency Comparison

- REST: 35–90 milliseconds, depending on how complicated the query is.

- 30 to 70 milliseconds for GraphQL
- gRPC: 10 to 15 milliseconds
- The bar chart clearly shows that gRPC always has low latency, while REST has latency spikes when it comes to these complicated requests.

5.3.2. Scaling Throughput (Line Graph)

- When there are a lot of requests, the performance curve of gRPC goes up in a straight line.
- After 3,000 concurrent requests, REST begins to level out. GraphQL, on the other hand, levels off a little early because of the extra work that resolvers have to do.
- This shows how gRPC has better horizontal scalability and connection persistence.

5.3.3. Line Graph

- Resource Use versus. Number of Users at the Same Time
- Because it has to build the latest connections so often, REST uses more CPU.
- The more complicated the resolvers are, the more CPU GraphQL uses.
- gRPC makes sure that CPU and memory usage stays the same, even when there is a lot of demand.
- These results show that gRPC works well in these distributions along with their containerized environments.

5.4. Limitations

The benchmark results offer valuable insights; yet, specific limitations impede their generalizability:

- Fluctuation in the network:Latency indicators depend on the current state of the network. Jitter, bandwidth variability, and routing paths are some of the things that could affect their performance metrics, especially in implementations that are spread out across a huge area.
- Size of the Sample and the Situation:The benchmarks were done in a controlled setting with these limited hardware resources. Different server architectures, hardware acceleration (such TLS offloading), or cloud environments may give different results.
- Limitations of the Tool: Load-testing methodologies have built-in these problems that make it very hard to accurately reproduce huge amounts of actual world traffic or multiplexed connections. HTTP/2 load testing may behave differently depending on how they are set up.
- Differences in how things are done: Each of the protocols was performed on an alternate framework or library, which could have triggered some modest differences in performance. The success rate of REST may vary based on the kind of server used (e.g., Express.js vs. FastAPI).
- Metrics That Were Not Included: The study neglected to evaluate critical characteristics such as developer productivity, investigation effort, or client SDK compatibility, and these are necessary for feasible deployment.

6. Conclusion and Future Scope

The efficiency test of REST, GraphQL, and gRPC reveals that no one API protocol is clearly better than each of them. Each is made for a specific kind of use case depending on the objectives and architecture about the system. gRPC is a great solution for internal applications along with these systems that require handling a lot of traffic because it functions very well and encounters very low latency. When you use Protocol Buffers with HTTP/2, this makes transferring and receiving data faster and easier. But because it fails to operate well with most online gadgets, it isn't very common in consumer-facing web environments, where straightforwardness of usage as well as accessibility are still quite crucial.

GraphQL, on the other one hand, strikes a good balance between becoming flexible and useful. It lets clients just ask for everything they need, which fixes the frequent problems with REST at which clients get excessively much or insufficient information.GraphQL is great for consumer-oriented systems like mobile and applications with one page since performance and effectiveness of bandwidth are of great importance. But there are trade-offs; complicated query planning and cache challenges could make it cost greater to put into implementation.

REST is still a dependable and well-known method that people like since it is simple, easy to read & works with a wide range of these systems. It continues to dominate public APIs and historical integrations, where consistency, scalability, and ease of debugging are more important than raw performance. REST will still be important for a long time due to the fact that it works with so many different kinds of data-related interactions. It just isn't as efficient for big or data-heavy instances.

Future comparative efforts may examine QUIC-based modes of transportation or other innovative technologies that extend the boundaries of communication in real time. One option is to develop hybrid applications that use the REST protocol for public-facing interfaces and gRPC for internal use. This could render things more effective without sacrificing compatibility. Adding artificial intelligence forecasting of traffic could make the service available much better by allowing APIs modify on the fly based on alterations to the network and workload.

It might be hard to determine these advances apart as workloads grow along with systems getting more multifaceted and spread out. The forthcoming generations of API frameworks will probably put a greater focus on how flexible they are than on how fast or easy they are to use. This will enable companies to pick or mix methods based on the situation. How effectively developers can establish a balance among performance, adaptability, and ease utilized in a digital world that constantly evolves will determine the next generation of API design.

References

[1] Seabra, Matheus, Marcos Felipe Nazário, and Gustavo Pinto. "Rest or graphql? A performance comparative study." *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. 2019.

[2] Yellavula, Naren. *Hands-On RESTful Web Services with Go: Develop Elegant RESTful APIs with Golang for Microservices and the Cloud*. Packt Publishing Ltd, 2020.

[3] Eeda, Naresh. *Rendering real-time dashboards using a GraphQL-based UI Architecture*. MS thesis. The University of Western Ontario (Canada), 2017.

[4] Tu, Tengfei, et al. "Understanding real-world concurrency bugs in go." *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 2019.

[5] Hunter II, Thomas. *Distributed Systems with Node. js*. O'Reilly Media, 2020.

[6] Vesić, Milena, and Nenad Kojić. "Comparative analysis of web application performance in case of using rest versus graphql." *Proceedings of the Fourth International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture (ITEMA), Online-Virtual*. 2020.

[7] Helgason, Arnar Freyr. "Performance analysis of Web Services: Comparison between RESTful & GraphQL web services." (2017).

[8] Indrasiri, Kasun, and Danesh Kuruppu. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[9] Gerlinghoff, Florian. "Vergleich von Introspected REST mit alternativen Ansätzen für die Entwicklung von Web-APIs hinsichtlich Performance, Evolvierbarkeit und Komplexität." (2020).

[10] Yellavula, Naren. *Hands-On RESTful Web Services with Go: Develop Elegant RESTful APIs with Golang for Microservices and the Cloud*. Packt Publishing Ltd, 2020.

[11] Eeda, Naresh. *Rendering real-time dashboards using a GraphQL-based UI Architecture*. MS thesis. The University of Western Ontario (Canada), 2017.

[12] Parakala, Adityamallikarjunkumar. "Building Analytics-Driven Bots: RPA Meets Business Intelligence." *International Journal of Emerging Research in Engineering and Technology* 2.1 (2021): 77-87.

[13] Touronen, Ville. "Microservice architecture patterns with GraphQL." *University of Helsinki* (2019).

[14] Galambos, Péter. "Cloud, fog, and mist computing: Advanced robot applications." *IEEE Systems, Man, and Cybernetics Magazine* 6.1 (2020): 41-45.

[15] Wein, Stephan, et al. "Flexible and lightweight framework for active industrial asset administration shells." *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. Vol. 1. IEEE, 2020.

[16] Gunawan, Go Frendi, and Jozua Ferjanus Palandi. "Redesigning CHIML: Orchestration Language for Chimera-Framework." *2018 Third International Conference on Informatics and Computing (ICIC)*. IEEE, 2018.

[17] Padala, S. (2019). AWS Cloud Architecture for Scalable Healthcare Contact Centers. American International Journal of Computer Science and Technology, 1(2), 21-26