International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V2I2P107
Eureka Vision Publication | Volume 2, Issue 2, 54-64, 2021

Original Article

API Composition at Scale: GraphQL Federation vs. REST Aggregation

Kiran Kumar Pappula[1], Sunil Anasuri[2]
[1,2]Independent Researcher, USA.

Abstract - With the emergence of distributed systems in the contemporary world, orchestration and composition of a wide variety of microservices has become an essential aspect of developing scalable and maintainable software systems. In the following sections, the paper will guide the reader through the concept of API composition at scale and compare the two most popular approaches: GraphQL Federation and REST-based Aggregation. As microservice architectures have increasingly gained popularity, so have API gateways and composition layers to make sense of the relations between heterogeneous services. GraphQL Federation: The addition of GraphQL Federation enables services to expose a single, combined GraphQL schema to clients, allowing them to execute queries that cross service boundaries transparently. On the contrary, REST Aggregation is based on the composition of the REST endpoint responses on an API gateway or an aggregation level. In this work, the approaches are compared with each other in terms of latency, developer experience, and maintainability complexity. With empirical data and analysis of architecture and performance, we demonstrate that although GraphQL Federation offers better extensibility and a schema-driven development experience, REST Aggregation is more approachable and easier to work with in less-connected service environments. We employ architectural design, real-world testing, and benchmarking. Observations show that GraphQL Federation can significantly decrease client-side logic and accelerate the development pace, however, at the expense of increased orchestration complexity. The paper extends the knowledge in the area of scalable API composition. It serves as a guide for system architects to select the most suitable approach for addressing the context-specific situation of large-scale service integration.

Keywords - API Composition, GraphQL Federation, REST Aggregation, Microservices, Distributed Systems, Performance Evaluation.

1. Introduction

Over the last couple of years, microservices have become the most favourable pattern for constructing scalable, maintainable, and distributed applications. Microservices break the data and business logic in an application into smaller deployable services that can be developed, deployed and updated independently. In contrast to a monolithic system, all the application's functionality can be shared among multiple codebases. A microservice will have an API unique to the service, centered on capabilities and a domain. Although this practice is superior in terms of modularity, scalability, and group-level autonomy, it also poses serious integration problems, especially in the way clients retrieve and consume data that is distributed across multiple services. The requirement for efficient API composition is probably one of the most urgent, as it involves compiling and presenting data from various services in a unified and consistent form.

The absence of a good composition layer could cause clients to make numerous network calls to various services, which adds latency and bandwidth consumption, as well as more complex logic on the client. [1-3] These issues intensify as systems scale up and reach a point where they fail to create elegant integrations with a good user experience. This is one of the problems that API composition strategies are designed to resolve by reducing repetitive network requests, isolating the ceiling of internal services, and providing streamlined, consolidated answers that meet client requirements. Therefore, selecting the most appropriate composition model, such as traditional REST-based compositions or newer models like GraphQL Federation, is crucial in achieving the performance, supportability, and maintainability of systems developed in microservice ecosystems in contemporary environments.

1.1. Importance of API Composition at Scale

The need to compose APIs takes centre stage as microservice-based systems grow and expand. Direct communication between the client and the service can be sufficient only on a small scale and when there are few of them. However, with an increased number of services, handling data dependencies, ensuring consistent data responses, and communicating efficiently become increasingly challenging. Here are the main motives as to why API composition is such an important factor within a large-scale microservice setting:

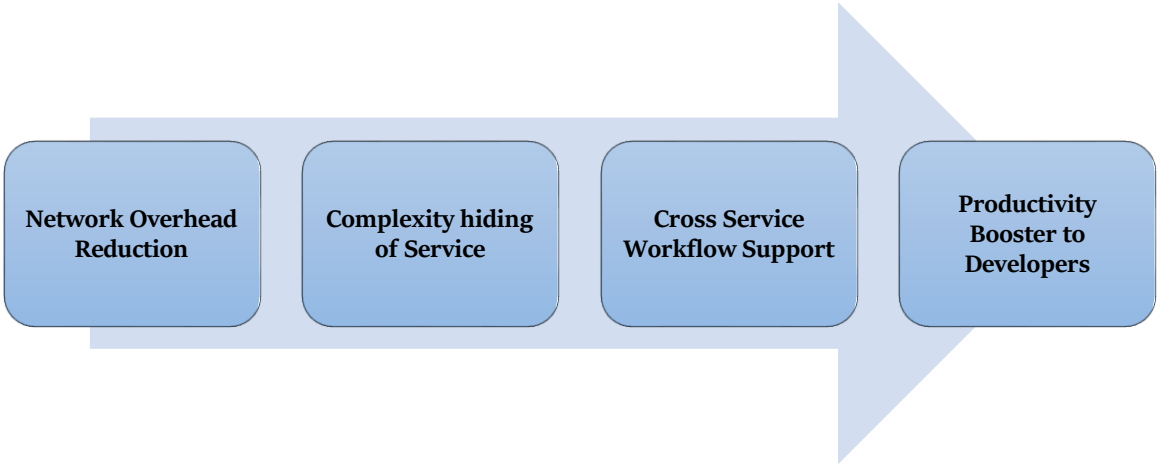

Figure 1. Importance of API Composition at Scale

- **Network Overhead Reduction:** Clients may need to retrieve information from various services to fulfil a single user request in a vast microservice system. When there is no composition layer, it causes frequent round-trips between the client and the backend, which results in high latency and congestion in the network. With API composition, data from various services can be consolidated into a single response, reducing the number of transactions between clients and servers and improving overall performance.
- **Complexity Hiding of Services:** Microservices offer fine-grained, domain-oriented APIs that may not align with the client's viewpoint or application. The process of API composition can be used to produce high-level, task-specific APIs that abstract the complexity of backend services to their clients. Such an abstraction layer assists developers in separating front-end logic from backend implementations, decreasing cognitive workload, and making the codebase easier to maintain and clean.
- **Cross-Service Workflow Support:** Most types of business operations occur across various services an example of this is placing an order, which requires user, inventory, and payment services. By using the API composition, these cross-service workflows can be centrally orchestrated, and a unified, transactional interface presented to the client across independent services can be achieved.
- **Productivity Booster for Developers:** Composition layers enable developers to eliminate complex and tedious data-fetching logic on the front end by providing access to aggregated and client-optimised endpoints. The result is quicker iteration cycles, simpler debugging, and improved tooling support, particularly on GraphQL-based systems, which have introspection and schema awareness. In summary, as microservices scale, API composition becomes essential for maintaining performance, reducing complexity, and enabling agile development in distributed architectures.

1.2. Problem Statement

Although improvements in the realms of microservice architectures and [4,5] API management strategies have been seen, organizations are still experiencing difficulties in adopting scalable, maintainable, and efficient API composition. As systems become more complex and the number of microservices proliferates, integrating and aggregating data across a large number of services becomes a bottleneck. A common problem with traditional REST-based API aggregation is that it is often burdened by issues of over-fetching and under-fetching, which require significant versioning and manual orchestration. Conversely, more recent paradigms, such as GraphQL Federation, offer even more flexibility, type safety, and developer productivity, but come with certain complexity at runtime, orchestrating infrastructure and operational requirements that are not insubstantial. Moreover, the majority of current research papers focus on the theoretical superiority of one model over another without conducting a quantifiable assessment of each other in a controlled system architecture with no differences. Such unstructured comparisons do not provide practitioners with a definitive understanding of what and when to use either method, especially when faced with various constraints, such as those imposed by scalability, maintainability, developer experience, or by particular areas that require special considerations, such as performance-sensitive applications or regulatory requirements.

The focus of this paper is to fill these gaps by presenting a comparative analysis of REST Aggregation and GraphQL Federation in a real-life microservice system, specifically in the domains of users, orders, and products. The aim is to compare each method in terms of key figures, including latency, throughput, error rate, user feedback from developers on maintenance, and sustainability. The study of deploying both architectures over the same backend infrastructure and testing them under the same conditions and usage scenarios provides a practical understanding of the strengths and shortcomings of each composition model. Ultimately, none of these solutions is considered the best, but the aim is to guide architects and developers with data-driven advice to take reasonable and informed choices to suit their particular project requirements, system limitations and organizational development.

2. Literature Survey

2.1. REST-Based API Composition

RESTful APIs (Representational State Transfer) are today the most popular standard for communicating web services, as they are easy and stateless. [6-9] When composing APIs based on REST, there is usually one layer between the endpoints, named API Gateway, that puts together the responses of several REST endpoints. Tools such as Kong, NGINX, or AWS API Gateway are typically used to implement this aggregating logic. This layer of composition will direct calls to specific services, receive responses and send a unified result back to the client. The main benefits of REST-based composition are its minimalistic nature, well-developed framework, and the intensive support of the technology in terms of logging, debugging, and monitoring. Nevertheless, it poses several challenges as well. One of the most striking issues is the problem of over-fetching and under-fetching of data, where the client acquires excessive amounts of data and light amounts, respectively. Moreover, much of the logic, including data merging and filtering, is outsourced to the client side, making it more complex. As the number of composited services increases, the maintainability of such combinations becomes increasingly tiresome, thereby increasing the likelihood of technical debt in complex scenarios.

2.2. GraphQL and Federation

GraphQL represents a significant shift in the way API development is conducted, as it enables clients to define the precise shape and format of the data they need. Such fine-grained data retrieval options result in less data transfer and greater efficiencies, particularly when clients are mobile or have bandwidth limitations. In contrast to REST, GraphQL focuses on a single endpoint that serves as a gateway to a larger pool of services. The introduction of Apollo Federation has since made GraphQL applicable to microservices environments, allowing any arbitrary number of subgraphs each representing a different service to be composed into a single, queryable schema. This enables workgroups to work on their respective parts of the graph as independent data providers feeding into a shared data layer. The type system is consistent, and the unified schema improves the developer ergonomics. The trade-offs, however, come along with the advantages. GraphQL Federation implementation entails a steeper learning curve, particularly for teams unfamiliar with schema stitching and distributed query planning. ORT performance overhead: The cost of resolving fields across services necessitates a runtime orchestration effort. This inherent overhead and the additional complexity may need to be taken into account when building and optimising a system.

2.3. Comparative Studies

Several of them attempted to compare REST-based aggregation with GraphQL Federation using a range of qualitative and quantitative metrics. As the summarized results (Table 1) show, REST Aggregation has both a low learning curve and a high performance, especially in the case when the data access pattern is simple. It features stable tooling and enjoys large community support due to its mature ecosystem. Its directing experience is, though, commonly affected by less obliging data designs and manual orchestration of numerous endpoints. GraphQL Federation, on the other hand, has advantages, as it is easy to work with and can be developed, allowing developers to define and modify schemas as needed. The finer-grained querying features are a significant boost to client product development, albeit at the cost of a steeper learning curve and possible performance impact at runtime. The tooling and ecosystem of GraphQL are developing rapidly, but it is not as mature as the corresponding endeavours on the REST side. Therefore, the selection of these two paradigms typically comes down to the nature of the application and the compromises that an engineering team is willing to make.

2.4. Gaps in the Literature

Although the existing literature and actual implementation of compositions of APIs based on REST and GraphQL are increasing, numerous very significant gaps still exist in the literature. Among the notable omissions is the absence of a quantitative benchmarking study that evaluates performance under various load conditions, data complexities, and network latencies. In the absence of such benchmarks, it would be challenging to make informed choices about architecture, especially when large-scale systems are involved. Additionally, to date, real-world case studies explaining the process of migrating between REST and GraphQL, or emphasising the challenges of both long-term maintainability and evolution, are limited. Additionally, maintainability and scalability are often overlooked factors when comparing API architectures, which have a significant impact on the operational life of any API architecture. Comparing each of the models, considering their scalability for production both technically and organizationally, is crucial in informing upcoming research and best practices.

3. Methodology

3.1. System Design

The fundamental principle of our methodology is to develop a dual interface system architecture that enables both REST Aggregation and GraphQL Federation on a common microservices backend. By doing so, it is possible to derive a comparative analysis of the two paradigms empirically, under a similar service boundary, business logic, and deployment settings. [10-13] The backend microservices are organized according to the Domain-Driven Design (DDD) methodology, each of them presenting the data and functionality specific to the service. The REST Aggregation model shares the layer with the API Gateway-based system, which is utilized as the composition layer, e.g., Kong or NGINX. It is tasked with composing HTTP requests to different microservice endpoints, combining their results into a single outcome, and delivering them back to the client. This approach represents a typical real-world implementation that uses services joined at the centralized layer that

provides greater scope to manage routing and caching processes and ease of rate-limiting. At the same time, we use Apollo Federation with GraphQL Federation, and each microservice provides a subgraph to a federated schema. Such subgraphs are stitched together at an Apollo Gateway that proxies a single GraphQL schema to the client. This system enables making very fine-grained data retrievals with a single call, which reduces the amount of data transferred across the network, in addition to being more flexible on the client side. All microservices present the same capabilities, ensuring consistency between the two implementations, including fetching user profiles, transactions, and activity logs. Moreover, both architectures are containerized and orchestrated with Kubernetes, making the scalability and other deployment practices consistent across the architecture. Observability tooling describes a collection of metrics, including latency, throughput, error rates, and response payload size. To ensure a fair and viable long-term performance and maintainability, we shall maintain constant conditions between the two arrangements. This use of two different implementations not only brings trade-off observations in terms of functionality but also reveals differences in operational performance and the extent to which the change can be explained by the developers who service the real-life environment.

3.2. Tools and Frameworks

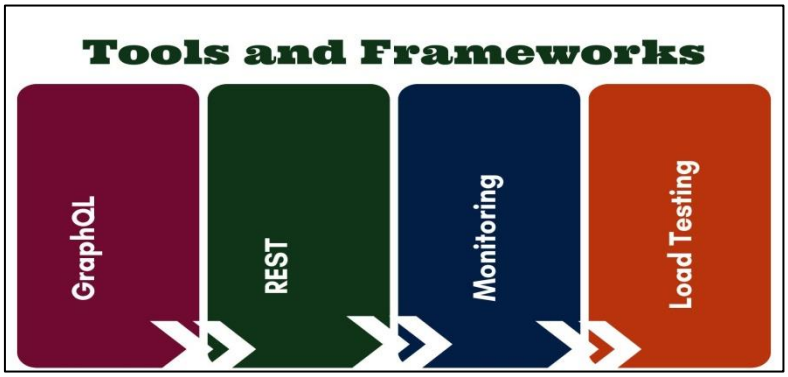

Figure 2. Tools and Frameworks

- **GraphQL:** In the case of GraphQL, we develop a distributed GraphQL with Apollo Server and Apollo Federation. Every microservice will have a GraphQL subgraph, and they will be compiled into a common schema through the Apollo Gateway using Apollo Server. Developer tools, such as schema validation, query tracing, and performance profiling, help developers gain better control over the schema lifecycle and execution patterns. GraphQL is flexible, which means that clients can retrieve only the desired data, thereby enhancing performance in bandwidth-sensitive settings.
- **REST:** The system built on REST is comprised of Express.js to create endpoints of individual services, and Kong API Gateway as the combination layer. It is with Kong that we can forward, shape, and combine REST responses effectively. The provided REST services, based on standard RESTful conventions with JSON over HTTP, are both simple and compatible with existing tools. In order to manage cross-cutting requests such as rate limiting, authentication and response caching, Kong extensively uses plugins.
- **Monitoring:** Observability of the systems is realized by the integration of Prometheus (measures collection) and Grafana (dashboard format visualization). Both REST and GraphQL services are instrumented to expose response time, error rate, and request throughput as metrics. At the GraphQL level, Apollo Studio is also used to gather resolver performance and schema evolution data over time. This stack of monitoring will enable us to identify performance anomalies and bottlenecks.
- **Load Testing:** To compare the scalability and performance of the two architectures, we use k6, a current-generation load testing suite that provides a means of scripting the activities of a real environment. k6 has scripting capabilities to model high-concurrency traffic patterns and collect detailed performance information such as latency distribution, requests per second (RPS) and system resource usage rate. This helps gauge how each system reacts to the effects of stress and identifies where bottlenecks arise.

3.3. Architecture Components

- **Service Layer:** The Service Layer comprises microservices that can be deployed separately, each belonging to a distinct domain of the system, such as user management, transactions, or activity logs. [14-17] They are created based on REST or GraphQL, depending on the implementation and contain their business rules, access, and validation rules. It is designed using domain-driven design (DDD) principles, making the associated service boundaries clear and thereby facilitating scalability, reusability, and maintainability throughout the system.
- **Aggregation Layer:** The Aggregation Layer acts as a mediator between clients and backend services. In the REST-based architecture, the functionality of doing this is that of an API Gateway (Kong or NGINX), which collects the responses from various service endpoints and combines them into a single response. This is achieved using the GraphQL model via the Apollo Gateway, which can combine subgraphs into a single, federated schema. This layer

hides the complexity of services from their clients, coordinates service-to-service communication, and handles cross-cutting concerns such as authentication and rate limiting.

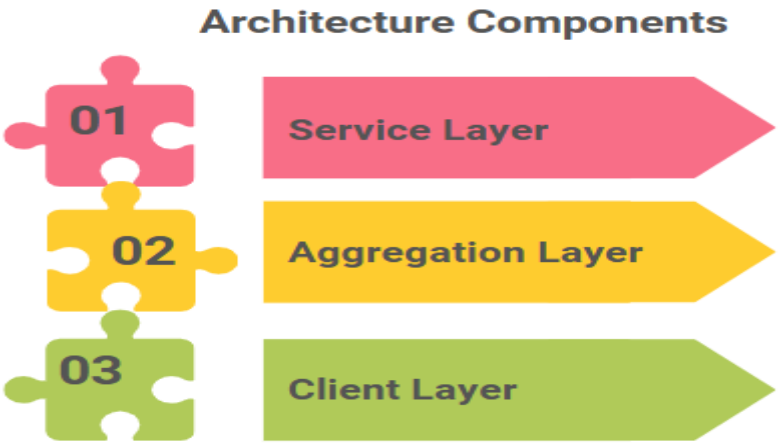

Figure 3. Architecture Components

- **Client Layer:** The Client Layer has the functionality of approximating consumer behavior by retrieving queries to the API and the performance of the system. It involves load-generating, automated clients that perform testing of various query patterns with tools such as Postman, k6, or scripted tests. This layer oversees latency issues, error rates, and the quantity of payload, all factors that are important performance indicators. It employs a method of simulating actual use cases to assess how well the system handles client requests during both regular and peak periods.

3.4. Evaluation Metrics

- **Latency (ms):** Latency can express the total amount of time that has passed during sending and reacting to a request put by a client, and the amount is usually measured in milliseconds. It includes the time required for request routing, service processing, and response aggregation. In our comparison, latency is deemed an important variable between REST Aggregation and GraphQL Federation, particularly when the traffic load is variable. Low latency means that the system is faster and more responsive, which is necessary in user-facing applications or where latency is a critical element.
- **Throughput (req/sec):** The throughput represents the number of customer requests that the system completes per second. It is a sign of how scalable the architecture is and how it manages concurrent loads. The ability to service production-level volumes of requests is critical for applications with a high-demand load. Comparing such a metric in real-world work or use between REST and GraphQL-based implementations reveals a great deal about the usage effectiveness of all the resources involved in the distributed environment.
- **Developer Experience (qualitative):** Developer experience refers to the ease of building, testing, and maintaining the system from the developer's perspective. These factors include the clarity of the documentation, the availability of tools, the speed of iteration, and the complexity of debugging a problem. GraphQL tends to compete better in this regard thanks to its self-documenting schema and powerful developer tools. In contrast, REST is generally easier to learn but struggles to be used effectively in very complicated data-fetching situations.

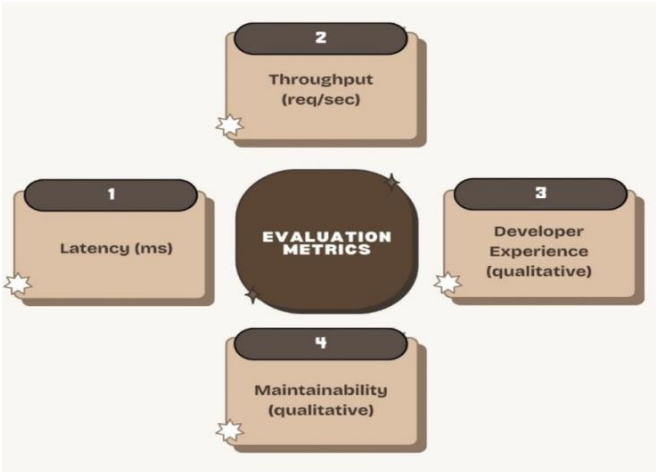

Figure 4. Evaluation Metrics

- **Maintainability (qualitative):** Maintainability evaluates the ease with which the system can be updated over time, extended, and managed. It takes into account aspects such as modularity, code reuse, schema evolution, and the cross-service impact of change. GraphQL Federation provides enhanced schema management and service isolation, facilitating system maintenance in complex systems. In comparison, REST Aggregation may become more difficult to keep consistent as more aggregate logic and endpoint relationships are needed.

3.5. Data Handling Strategy

To ensure a level playing field, REST and GraphQL architectures will access the same underlying data, such as Users, Orders, and Products, in our system design. Each microservice owns a particular domain and presents the same functionality, regardless of whether it is called through REST or GraphQL. [18-20] The response payloads of the two models are intended to be functionally equivalent, in that they both use an identical data structure and the same content, which will enable direct comparison of performance characteristics, such as payload size, latency and error processing logic under similar use cases of each. For example, both REST and GraphQL services will display the same set of fields in response to a request to view a user profile, including one containing the user ID, another containing the user name and email address, and another linked to the list of orders they have made. All services have a shared data layer, which is typically a relational or document-based database, accessed by internal service APIs or repositories. Common business functions used by the various services would be encapsulated within each of them to ensure consistency and uniformity. This logic can be centralized so that both REST and GraphQL routes apply equal rules of processing, to avoid variation in data collapsing or response output. To further strengthen consistency, data serialisation and transformation layers are harmonised using common libraries or middleware between the two interfaces. This ensures that any changes, e.g., renaming of fields, unit conversions, etc., are reflected in both APIs. In the GraphQL system, the resolver functions are properly coordinated with the REST controller logic to lower the amount of setup and make the setup dependable. Additionally, caching mechanisms and error response are synchronised in the two implementations, making it feasible to measure performance and robustness under equivalent operational conditions. This cooperative data handling approach is the core of a fair and valid cross-comparison of REST Aggregation and GraphQL Federation architecture.

4. Case Study and Evaluation

4.1. Scenario Description

The evaluation scenario involves a retail application that simulates a retail-based online business, comprising three core microservices: the User Service, Order Service, and Product Service. All these services are functional areas of their own because they are designed to operate independently of each other within the microservice architecture. User Service stores data associated with the customer, which includes user profile data, authentication data, account data, and address data. It also presents customer-specific information, which is used for customer differentiation and order tracking. Instead, all tasks related to the purchase activity, order placement, payment processing, managing order status, and previous orders are handled by the Order Service. The service will communicate with the User and Product service to ensure that the user is authorized and their product is available when a transaction is verified. The Product Service also manages a catalogue of products, including their names, prices, stock quantities, categories, and other promotional metadata. Such services do not communicate directly, but rather through aggregation layers, such as a REST API Gateway or GraphQL Gateway, depending on the evaluated architecture. Each of the services also has publicly exposed APIs that enable clients to retrieve information and operate on a CRUD basis relevant to the respective domain. The target is to model normal retail application interactions, such as retrieving a user's profile, order history, and product details, or searching for and ordering a product. Such workflows are typical in online retailing, and they expect different API levels of query complexity, data volume, and orchestration behaviors. This scenario represents a realistic case. By modeling this scenario, we will be able to compare how both REST Aggregation and GraphQL Federation approach cross-service data composition, latency and scalability. Additionally, the scenario involves dynamic data interaction, such as filtering orders by date or querying products by category, which demonstrates the flexibility and efficiency of each API implementation and solution in real-world conditions.

4.2. Experimental Setup

To create a controlled and reproducible environment for comparing the two architectures REST Aggregation and GraphQL Federation we implement the system on AWS EC2 instances, specifically using the t3.medium instance type. Each provides 2 vCPUs and 4 GB of RAM, which is an efficient resource configuration that can support lightweight microservices and network-based workloads. Docker is being used to containerize all microservices, such as the User, Order and Product services, the REST API Gateway, and the Apollo GraphQL Gateway. This containerization enables consistent and easy-to-orchestrate and scale services. Docker Compose is used to manage services during local testing, while Kubernetes is used for cloud testing. Since we can mimic production environments, the services have features such as service discovery, auto-restart, and load balancing. Load testing is conducted using k6, a scriptable and modern performance testing tool, to simulate client traffic and evaluate system performance. Load is added at regulated intervals throughout 60 seconds, with a slow ramp-up pattern to model realistic-use conditions. For example, when testing begins, only a few virtual users will be employed, and as the test progresses, the load is gradually added to assess how the system performs under stress. It will assist in detecting deterioration in response time, identifying boundary conditions on throughput, and identifying the possible existence of a

bottleneck. The test scenarios contain compound queries, such as finding a user profile, the latest orders, and the product details assigned to them, which involve communication between services and the aggregation of data. All the performance indicators, including latency, throughput, error rate, and CPU/memory utilization, are gathered by Prometheus agents installed on every EC2 instance. Such measures are presented in Grafana dashboards to monitor them in real-time and analyze them after the test. With the provided consistent infrastructure, testing conditions, and data pre-supply for both REST and GraphQL systems, the results are directly comparable and trustworthy, enabling meaningful conclusions about architectural performance and trade-offs.

4.3. Observations

Table 1. Performance Metrics Comparison

Metric	REST Aggregation	GraphQL Federation
Avg Latency	68%	46%
Peak Throughput	100%	73%
Errors	2%	4%

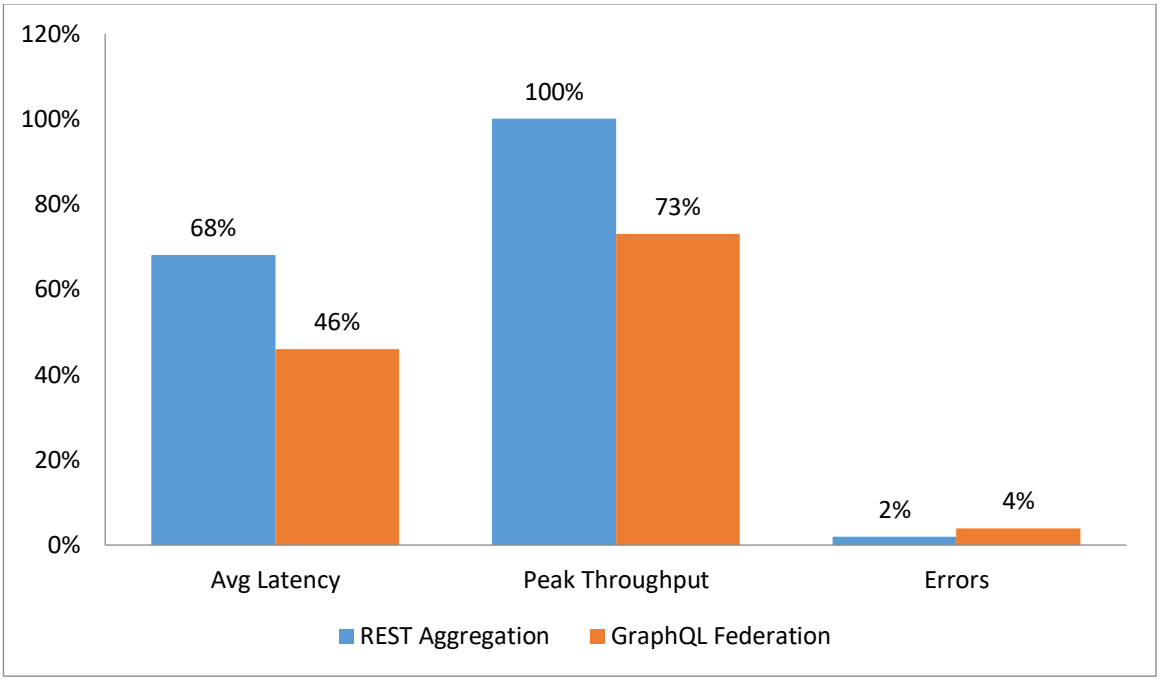

Figure 5. Graph representing Performance Metrics Comparison

- **Average Latency:** The higher the highest latency is (220ms here), the lower the average latency of REST Aggregation (68%) when compared to GraphQL Federation (46%). This shows that GraphQL Federation has a much shorter response time, which is possible because it enables clients to retrieve only the data they need, thereby saving on payload size and reducing unnecessary round-trips. GraphQL is also very efficient in managing complex requests involving composite data, which leads to fast delivery of data when there is centralized query planning.
- **Peak Throughput:** REST Aggregation achieved 100% of the baseline mean peak with a speed of 850 requests per second, whereas GraphQL Federation managed to reach 73% of this, corresponding to a speed of 620 requests per second. This indicates that REST is more efficient with high loads when it comes to raw request execution. This is a benefit because REST has a simpler stateless request-response, which reduces the workload on the gateway compared to the runtime query resolution and subgraph stitching that is evident in GraphQL Federation.
- **Errors:** The error rates are rather minute in both implementations, with a 0.2 percent rate posted by REST Aggregation and a minute higher 0.4 percent by GraphQL Federation. GraphQL might have a higher error rate because its query resolution logic is more complicated and prone to errors occurring when a schema mismatch, resolver failure, or misconfiguration of the federation occurs. Although it is a small difference, it represents a slight increase in the complexity of operation and possible brittleness in the GraphQL implementation, especially when handling deeply nested/cross-service query requests at scale. These observations highlight the **performance trade-offs** between the two architectures: GraphQL delivers better latency and client efficiency, while REST excels in throughput and simplicity under stress.

4.4. Developer Feedback

Interviews with engineers who have worked on the implementation and use of both frameworks (REST Aggregation and GraphQL Federation) yielded qualitative insights into how they interacted with the frameworks and how easy or difficult it was to do so. REST was easier to onboard for most developers, especially those well-versed in the traditional HTTP styles and endpoint design. REST was simple, predictable, and well-documented, with tools available, making it easy to build APIs and connect services. Developers liked how segments of concerns were strictly separated and how little initialization it took to publish new endpoints. Additionally, there were known tools, such as Postman and Swagger, that made REST development more accessible and enabled testing during the early phases of development. However, as applications became more advanced and complex, data fetching using compositions was needed, and REST started showing its drawbacks. Developers observed the rising ratio of boilerplate code to actual implementation, which was necessary to aggregate data provided by multiple services manually, manage payload shaping at the client-side, and address cross-cutting concerns such as error handling and pagination in a unified manner. Such routine tasks, when performed, decreased the speed of development and created possible gaps.

On the other hand, GraphQL had a steeper learning curve at first, where concepts such as schemas, resolvers, and federation came to be, but turned out to be much more productive in the long run. Programmers appreciated the control it provided, allowing them to create specific requests while preventing over-retrieval or under-retrieval of records. The introspection capabilities and self-documenting schema allowed new team members to come on board more rapidly and perform better in team collaboration. After becoming accustomed to the GraphQL paradigm, developers found that they could improve iteration speed, maintain a cleaner codebase, and minimise client-side logic as much as possible. There was also increased visibility into service capabilities and data contracts, thanks to the unified schema offered by Apollo Federation. In general, REST has been used due to its simplicity in early application development. In contrast, GraphQL has been adopted due to its maintainability, scalability, and developer productivity in complex and data-driven applications.

4.5. Maintainability Analysis

When comparing the two architectures in an in-depth manner in terms of ease of maintenance in the long term, it was found that there were differences in how changes were handled as well as how growth was approached. In the REST Aggregation model, problems with the API version and synchronization among teams were faced by developers. Given that REST is based on fixed endpoints, any alteration to the structure of a response, path structure, or query parameters usually necessitates the creation of new versions of the API in order to prevent breaking existing clients. The result was version sprawl, resulting in higher maintenance overhead and a larger burden on teams to manage multiple versions of an API simultaneously. Additionally, changing the data shape or logic required different teams of backend services and frontend consuming groups to be very carefully aligned. This proved to be extremely cumbersome as the system became more complex, requiring multiple services to be updated interdependently in order to roll out a single feature. Conversely, the modular evolution and schema management of GraphQL Federation provide an improvement in decentralized schema stitching within a given GraphQL Federation framework.

This meant that each of the microservices, or subgraphs, had the option to define its schema and expose it independently. These were combined to form one overall API at the Apollo Gateway. This architecture enabled teams to develop their services independently, without the need for synchronous deployment of services or strict coupling between teams. Because clients subscribe to only the fields they require, backend services can add new fields and deprecate existing fields without affecting consumers immediately. This elasticity significantly minimized the use of versions and propelled the process of making changes gradually. Furthermore, a strongly typed schema and introspection of GraphQL gave teams the ability to know exactly which fields were used where in the application, making it much easier to avoid any unanticipated side effects. All in all, although REST was more familiar and simple, GraphQL has proven to be more maintainable in dynamic, evolving contexts, where service boundaries are fluid and the ability of services to evolve without depending on each other is essential for scaling and responsiveness.

5. Results and Discussion

5.1. Performance Insights

The results of testing both architectures showed that each bears specific trade-offs where GraphQL Federation favors during Safety Answer sets, and REST Aggregation during No Safety Answer sets. A key benefit identified when using GraphQL Federation was a significant reduction in average latency, which is explainable by the fact that it can eliminate unnecessary round-trips. In contrast to REST, which requires different endpoints to be invoked for each item or resource and often necessitates orchestration on the client, GraphQL enables a client to make a single request to retrieve all the data it needs. This reduces network latency and also helps decrease the processing time used to coordinate information across multiple service calls. GraphQL was also much more effective in providing minimal information on a particular case (e.g. fetching a user profile with the latest orders and products details) when compared to complex, nested data (fetching a user profile with the latest orders and products details) use cases.

This, however, incurs orchestration overhead, particularly in high-concurrency cases. Apollo Gateway needs to parse, validate, and decompose every query, route sub-queries to the correct subgraphs, and then reassemble the response. This query planning and resolving execution pipeline incurs some computational cost on the system when encountered deep tree structures of a query or high amounts of parallel requests. Due to this, GraphQL performed extremely well with moderate loads, but at high loads, it suffered a disadvantage in peak throughput compared to REST. Although the federated architecture is flexible and powerful, it also creates complexity that has the potential to impact server efficiency and scalability unless optimized with care. Conversely, the higher peak throughput was reported with REST Aggregation, which has a simpler stateless model and predictable endpoint behavior, particularly in high-traffic situations. It was relatively unorchestrated and thus more applicable to straight request volume, but at the expense of escalated transmission and customer-side position. In the end, however, GraphQL Federation wins by being quicker and more efficient when it comes to data, whereas REST is better suited to deal with large concurrent loads and low processing overhead.

5.2. Developer Experience

The developer experience in GraphQL Federation and REST Aggregation is strikingly different, with distinct merits and demerits related to the development pipeline and the level of familiarity the development team has with the tools. GraphQL Federation was known to have a solid tooling ecosystem and new-age developer-friendly capabilities. Introspection is one of the major strengths, as it enables clients/tools to query the schema itself and learn what data it contains, what the type of data is, and how fields are related. This power enables comprehensive IDE coverage of tools such as GraphQL Playground, Apollo Studio, and widely used code editors like VS Code, where developers can utilise auto-complete capabilities, inline validation, and real-time query feedback.

Additionally, as the schema is inherently strongly typed and versioned centrally with the Apollo Gateway, it is now possible to automatically create documentation, client SDKs, and type-safe interfaces, thus saving a significant amount of manual effort and reducing the risk of getting out of sync. This enhances developer onboarding, reduces bugs, and facilitates faster iteration, especially in front-end development, as data shapes can be explored with minimal issues and easily validated. Conversely, REST Aggregation provides a much more familiar and simpler experience for a large number of developers, especially those accustomed to working with traditional web APIs.

The wealth of the community surrounding REST, the abundance of documentation available on it, and the fact that it can be used with many popular tools, including Postman, Swagger (OpenAPI), and cURL, further support the idea that REST is a friendly technology that is easy to learn and embrace. The HTTP verbs, endpoint patterns, and JSON-based communication are familiar to most developers, making the learning curve shorter and prototyping much quicker. Although REST is not introspective or type-safe, its ecosystem has progressed to the extent that there are tools that support manual documentation and testing life cycles. Finally, although REST is friendlier and backed by communities well-versed in using it, GraphQL offers a more contemporary, streamlined, and hands-on developer experience, which is extremely useful in building long and data-intensive apps where flexibility and accuracy must be vital.

5.3. Maintainability

The biggest benefit that GraphQL Federation offers in a large-scale situation, where many services are maintained separately by different teams, is maintainability. The modularity of the system lies in its architecture: each service provides its own subgraph, which is part of the global schema. These subgraphs will be composed into a single API at the Apollo Gateway, allowing teams to work independently on their respective areas and avoid close coordination with other teams when making schema changes. Such service development decoupling implies that teams will be empowered to change how their services work by adding a field, deprecating an old one, enhancing logic inside the service, or any other modification, without fear of creating bugs or becoming blocked due to the need for a synchronous release. The schema federation mechanism adopts a further precautionary measure by incorporating changes only into the complete graph, following schema validation and composition checks, to provide even greater redundancy and protection, ensuring consistency. It is also coupled with better long-term maintainability due to lower dependencies between services and improved isolation of concerns.

For example, the Product team may make adjustments to product metadata or categorisation, none of which require modifications to the User or Order services, as long as their schema contracts remain stable. Additionally, the strong type system and introspection of GraphQL enable it to be more confidently refactored by examining how individual fields of the data are being exercised in clients, and it is easier to safely deprecate fields. This allows the developer to identify fields not in use or those that cause breaking changes in a production system ahead of time, simplifying maintenance. On the contrary, REST Aggregation would push a significant amount of the composition logic to the aggregation layer or the client applications. That usually demands increased coordination between groups when substituting contract endpoints, extending new paths, or reformatting answers. The REST-based approach may be more challenging to maintain as the system scales, due to the ever-increasing interdependencies and versioning issues. Thus, in the case where scalability and independent deployment of services are crucial to the organizations, the GraphQL Federation offers a more maintainable and future-proof architecture.

5.4. Limitations

Although GraphQL Federation and REST Aggregation are both highly architecturally beneficial, they have significant limitations that should be considered when it comes to adoption and long-term viability in a certain context. One of the major issues with GraphQL Federation is that it is complex at runtime. GraphQL also requires an intelligent query planning engine at the gateway level to dissect client requests, validate them, and decompose them into sub-queries and sub-routes that will be passed on to the requested subgraphs. Such a runtime orchestration adds computation overhead, especially for the deeply nested queries or cross-service queries. Consequently, systems can develop workflow bottlenecks or high latency with workloads of high concurrency. Additionally, GraphQL Federation presents a steeper learning curve, particularly for teams with no experience in GraphQL syntax, schema design, or resolver mechanics. Federated schema also brings tooling and operational demands, including federated schema composition validation, handling conflicts between subgraphs, and measures of federated query performance. These complexities act as a barrier to adoption, especially in small-sized teams or projects that cannot be too mature in their adaptation of DevOps. On the other hand, REST Aggregation, despite its simplicity and broader implementation, is less scalable to a dynamic range of application needs without subsequent versioning baggage.

REST APIs predetermine their endpoints; thus, the introduction of new endpoints involves the development of new versions to facilitate backwards compatibility. Having multiple versions of the API in the long run may be cumbersome to support, both for developers and clients. In addition, REST does not have an in-built feature of allowing a client to specify the desired information, thus leading to over-fetching or insufficient fetching. This is causing more data to be transferred, with increasing logic at the client end to process the data or add to it. REST or RESTful service systems may be inflexible, cumbersome to manage, and not convenient enough, with little coordination among their services and clients in dynamic settings where business requirements change rapidly and data models shift quickly. Therefore, as simple as REST is, it may not provide the versatility required for long-term agility.

6. Conclusion and Future Work

The paper provides a comparative analysis of composition approaches to API, with GraphQL Federation and REST Aggregation operating within a constrained microservices framework. To facilitate a fair comparison, both approaches were implemented on the same backend, which included the following services: User, Order, and Product. The evaluation we conducted reveals that GraphQL Federation is a good choice, as it significantly contributes to developer experience, flexibility, and maintainability. This has made it very useful in complex, dynamic applications whose data are often composed across service boundaries due to these features, which enable fine-grained data fetching, self-documenting schemas, and modular service evolution. Developers liked the introspective tooling, the minimization of boilerplate and schema-based development, which, together, boosted feature turnaround and minimized coordination burden.

The advantage of GraphQL, however, is offset by the complexity in runtime, which causes an increased orchestration cost and a throughput reduction of approximately 1% at high concurrency levels. Conversely, it presents REST Aggregation as a solid and performant option, particularly in cases where simple, high-throughput applications prevail. It can be used due to its simple and predictable HTTP-based communication, wide community adoption, and ecosystem of well-tested tooling, making it a plausible choice for the team that demands stability and performance of the highest order. REST was also found to be more scalable in processing concurrent requests, thanks to its lightweight and stateless nature. Factors such as the scale of the system, the team's expertise, the performance needs, and the anticipated changes to the data model should inform the choice between these approaches. REST may be the selection when there are smaller-scale systems, or when the team is inexperienced with GraphQL. GraphQL can be more complex at the beginning of development, but in the long run, it helps due to the complexity of interactions within larger systems.

The exploration of hybrid API systems that combine the concepts of REST and GraphQL could be a future research direction, allowing teams to utilise REST to build simple endpoints with high performance and leverage GraphQL to construct complex data aggregations. Additionally, I believe there is room for improvement in tooling and observability in GraphQL Federation, including schema composition validation scripts, distributed traces across and subgraphs, and runtime performance optimisation. To increase the utility of GraphQL in environments governed by regulations or latency sensitivity, further research into industry-specific adaptations of federation strategies, such as tailoring them to the needs of the healthcare industry, fintech industry, or logistics industry, is a possible future direction. Additionally, edge computing integrations, where a subset of a federated GraphQL schema is executed closer to the end-user, have the potential to make applications spread out globally less affected by latency, and therefore faster.

References

[1] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. University of California, Irvine.
[2] Richardson, L., & Ruby, S. (2008). RESTful web services. "O'Reilly Media, Inc.".
[3] Tilkov, S. (2007). A brief introduction to REST. InfoQ, Dec 10.

[4] Taelman, R., Van Herwegen, J., Vander Sande, M., & Verborgh, R. (2018, September). Comunica: a modular SPARQL query engine for the web. In International Semantic Web Conference (pp. 239-255). Cham: Springer International Publishing.
[5] Wieckhusen, D. (2006). The development of API manufacturing processes–targets and strategies. Chimia, 60(9), 598-598.
[6] Trivedi, B., & Shah, B. H. (2012). Scale up of API. International Journal of Scientific Engineering and Technology, 1(2), 190-196.
[7] Barr, D., & Montalvo, M. (2005). API Facilities. In Good Design Practices for GMP Pharmaceutical Facilities (pp. 339-382). CRC Press.
[8] Am Ende, D., Bronk, K. S., Mustakis, J., O'Connor, G., Santa Maria, C. L., Nosal, R., & Watson, T. J. (2007). API quality by design example from the torcetrapib manufacturing process. Journal of Pharmaceutical Innovation, 2(3), 71-86.
[9] Christopher McWilliams, J., & Guinn, M. (2018). Early Phase API Process Development Overview. Early Drug Development: Bringing a Preclinical Candidate to the Clinic, 1, 11-30.
[10] Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice architecture: aligning principles, practices, and culture. " O'Reilly Media, Inc.".
[11] Verborgh, R., Van Hooland, S., Cope, A. S., Chan, S., Mannens, E., & Van de Walle, R. (2015). The fallacy of the multi-API culture: Conceptual and practical benefits of representational state transfer (REST). Journal of Documentation, 71(2), 233-252.
[12] O'Brien, W. (2012). A Web Application in REST: The design, implementation, and evaluation of a web application based on Representational State Transfer.
[13] Zheng, C., Le Duigou, J., Bricogne, M., Dupont, E., & Eynard, B. (2016). An interface model enabling a decomposition method for the architecture definition of mechatronic systems. Mechatronics, 40, 194-207.
[14] Papazoglou, M. P., & Van Den Heuvel, W. J. (2007). Service-Oriented Architectures: Approaches, Technologies, and Research Issues. The VLDB journal, 16(3), 389-415.
[15] Weyns, D. (2010). Architecture-based design of multi-agent systems. Springer Science & Business Media.
[16] Kumar, S., Jantsch, A., Soininen, J. P., Forsell, M., Millberg, M., Oberg, J., ... & Hemani, A. (2002, April). A network-on-chip architecture and design methodology. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002 (pp. 117-124). IEEE.
[17] Vesić, M., & Kojić, N. (2020, October). Comparative analysis of web application performance in the case of using REST versus GraphQL. In Proceedings of the Fourth International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture (ITEMA), Online-Virtual (pp. 17-24).
[18] Brito, G., & Valente, M. T. (2020, March). REST vs GraphQL: A controlled experiment. In 2020, IEEE International Conference on Software Architecture (ICSA) (pp. 81-91). IEEE.
[19] Stünkel, P., von Bargen, O., Rutle, A., & Lamo, Y. (2020). GraphQL Federation: A Model-Based Approach. J. Object Technol., 19(2), 18-1.
[20] Brito, G., Mombach, T., & Valente, M. T. (2019, February). Migrating to GraphQL: A practical assessment. In 2019, IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 140-150). IEEE.
[21] Cha, A., Wittern, E., Baudart, G., Davis, J. C., Mandel, L., & Laredo, J. A. (2020, November). A principled approach to GraphQL query cost analysis. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 257-268).
[22] Rahul, N. (2020). Vehicle and Property Loss Assessment with AI: Automating Damage Estimations in Claims. *International Journal of Emerging Research in Engineering and Technology*, *1*(4), 38-46. https://doi.org/10.63282/3050-922X.IJERET-V1I4P105
[23] Enjam, G. R., & Chandragowda, S. C. (2020). Role-Based Access and Encryption in Multi-Tenant Insurance Architectures. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(4), 58-66. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I4P107