https://doi.org/10.31891/2307-5732-2025-355-79
УДК 637.5.02

SIKORA ROSTYSLAV
State University of Trade and Economics
https://orcid.org/0009-0002-7570-9826
e-mail: r.sikora@knute.edu.ua
POSTOLIUK ANDRII
Ternopil Ivan Puluj National Technical University
https://orcid.org/0009-0004-0169-3379
e-mail: mrapostoliuk@gmail.com

COMPARATIVE ANALYSIS OF THE EFFECTIVENESS OF ARCHITECTURAL STYLES REST, GRAPHQL, AND GRPC FOR SCALABLE MICROSERVICE SYSTEMS

The microservice architectural pattern has rapidly emerged as a dominant and highly effective approach for building complex, distributed, and scalable applications in modern software engineering. By judiciously decomposing monolithic applications into a collection of small, independent, and loosely coupled services, organizations are increasingly able to achieve significant gains in agility, resilience, and maintainability. This decomposition facilitates independent development, deployment, and scaling of individual components, which is crucial for handling the dynamic demands of contemporary systems. As the adoption of microservices continues to grow and mature across various industries, the selection of appropriate communication protocols and architectural styles to govern interactions between these discrete services has become an increasingly critical and strategic decision. The chosen architectural style profoundly impacts various non-functional requirements and overall system effectiveness, including but not limited to scalability, performance, ease of development, integration complexity, and operational efficiency. This paper undertakes a comprehensive comparative analysis of three prominent and widely adopted architectural styles frequently employed in modern microservice architectures: Representational State Transfer (REST), GraphQL, and gRPC (Google Remote Procedure Call). While each of these styles inherently offers distinct advantages and disadvantages, their ultimate suitability is often highly contingent upon the specific requirements, constraints, and operational context of the target application. Our research delves deeply into the core principles, underlying technologies, and defining characteristics of each of these architectural styles, providing a robust theoretical understanding that underpins the subsequent comparative evaluation. The comparative evaluation will assess the effectiveness of these styles based on a meticulously defined set of key criteria, encompassing critical aspects of distributed system design: scalability, performance, development and integration simplicity, support for various interaction types (e.g., streaming), observability, and security. For instance, we will analyze how REST's statelessness and caching capabilities contribute to scalability, while also examining its potential for over-fetching and under-fetching, which can impact performance. For GraphQL, we will highlight its client-driven data fetching capabilities, which precisely mitigate these issues, but also consider the increased server-side complexity and challenges in caching. In the case of gRPC, the analysis will focus on its high performance due to Protocol Buffers and HTTP/2, as well as its strengths in efficient streaming, while acknowledging potential complexities in browser integration and debugging. Ultimately, this research aims to offer nuanced insights into when each architectural style might represent the most appropriate and effective choice for building highly scalable and resilient microservice systems. By examining the theoretical foundations and practical implications, alongside real-world case studies and industry experiences, this paper seeks to provide a comprehensive and practical guide for architects and developers who are navigating the inherent complexities of modern distributed system design and striving to make informed decisions regarding inter-service communication paradigms. The structured approach of this article will first lay the theoretical groundwork for each architectural style, then establish the criteria for a comparative analysis, followed by a detailed comparison, discussion of potential case studies and real-world implementations, an exploration of existing challenges and promising future research directions, and finally, a concise concluding summary of the findings. This structured analysis will empower practitioners to select the most suitable architectural style, thereby contributing to the development of more efficient, robust, and maintainable distributed applications.

Keywords: microservices, architectural styles, REST, GraphQL, gRPC, scalability, performance, distributed systems, API design.

СІКОРА РОСТИСЛАВ
Державний торговельно-економічний університет
ПОСТОЛЮК АНДРІЙ
Тернопільський національний технічний університет імені Івана Пулюя

ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ АРХІТЕКТУРНИХ СТИЛІВ REST, GRAPHQL ТА GRPC ДЛЯ МАСШТАБОВАНИХ МІКРОСЕРВІСНИХ СИСТЕМ

Мікросервісний архітектурний шаблон швидко став домінуючим та високоефективним підходом для побудови складних, розподілених та масштабованих додатків у сучасній розробці програмного забезпечення. Шляхом розумної декомпозиції монолітних додатків на сукупність малих, незалежних та слабо зв'язаних сервісів, організації все частіше можуть досягти значних успіхів у гнучкості, відмовостійкості та зручності обслуговування. Ця декомпозиція сприяє незалежній розробці, розгортанню та масштабуванню окремих компонентів, що є критично важливим для обробки динамічних вимог сучасних систем. Оскільки впровадження мікросервісів продовжує зростати та дозрівати в різних галузях, вибір відповідних протоколів зв'язку та архітектурних стилів для управління взаємодіями між цими дискретними сервісами став все більш критичним та стратегічним рішенням. Обраний архітектурний стиль суттєво впливає на різні нефункціональні вимоги та загальну ефективність системи, включаючи, але не обмежуючись масштабованістю, продуктивністю, легкістю розробки, складністю інтеграції та операційною ефективністю. Ця стаття проводить комплексний порівняльний аналіз трьох видатних та широко прийнятих архітектурних стилів, які часто використовуються в сучасних мікросервісних архітектурах: Representational State Transfer (REST), GraphQL та gRPC (Google Remote

Procedure Call). Хоча кожен з цих стилів за своєю суттю пропонує окремі переваги та недоліки, їхня кінцева придатність часто значною мірою залежить від конкретних вимог, обмежень та операційного контексту цільового застосування. Наше дослідження глибоко занурюється в основні принципи, базові технології та визначальні характеристики кожного з цих архітектурних стилів, надаючи надійне теоретичне розуміння, яке лежить в основі подальшої порівняльної оцінки. Порівняльна оцінка буде оцінювати ефективність цих стилів на основі ретельно визначеного набору ключових критеріїв, що охоплюють критичні аспекти проектування розподілених систем: масштабованість, продуктивність, простота розробки та інтеграції, підтримка різних типів взаємодії (наприклад, потокова передача), спостережуваність та безпека. Наприклад, ми проаналізуємо, як безстатевість REST та можливості кешування сприяють масштабованості, одночасно досліджуючи його потенціал для надмірного або недостатнього отримання даних, що може вплинути на продуктивність. Для GraphQL ми підкреслимо його можливості отримання даних, орієнтовані на клієнта, які точно пом'якшують ці проблеми, але також розглянемо підвищену складність на стороні сервера та проблеми з кешуванням. У випадку gRPC, аналіз зосередиться на його високій продуктивності завдяки Protocol Buffers та HTTP/2, а також його сильних сторонах в ефективній потоковій передачі, одночасно визнаючи потенційні складнощі в інтеграції з браузером та налагодженні. Зрештою, це дослідження має на меті запропонувати тонкі відомості про те, коли кожен архітектурний стиль може бути найбільш відповідним та ефективним вибором для побудови високомасштабованих та відмовостійких мікросервісних систем. Розглядаючи теоретичні основи та практичні наслідки, поряд з реальними тематичними дослідженнями та досвідом галузі, ця стаття прагне надати комплексний та практичний посібник для архітекторів та розробників, які орієнтуються у притаманних складнощах сучасного дизайну розподілених систем та прагнуть приймати обґрунтовані рішення щодо парадигм міжсервісної комунікації. Структурований підхід цієї статті спочатку закладе теоретичну основу для кожного архітектурного стилю, потім встановить критерії для порівняльного аналізу, після чого буде проведено детальний порівняння, обговорення потенційних тематичних досліджень та реальних впроваджень, дослідження існуючих проблем та перспективних напрямків майбутніх досліджень, і, нарешті, коротке підсумкове резюме висновків. Цей структурований аналіз дозволить практикам вибрати найбільш підходящий архітектурний стиль, тим самим сприяючи розробці більш ефективних, надійних та легких у підтримці розподілених додатків.

Ключові слова: мікросервіси, архітектурні стилі, REST, GraphQL, gRPC, масштабованість, продуктивність, розподілені системи, дизайн API.

Стаття надійшла до редакції / Received 28.05.2025
Прийнята до друку / Accepted 26.06.2025

Theoretical Foundations of Architectural Styles

In the realm of microservices, the way services communicate with each other is paramount to the overall system's behavior and characteristics. Different architectural styles dictate the rules and conventions for this inter-service communication. This section will delve into the fundamental principles and characteristics of REST, GraphQL, and gRPC, providing a solid theoretical understanding for the subsequent comparative analysis.

REST (Representational State Transfer)

Representational State Transfer (REST) is an architectural style that has gained widespread adoption for building web services and, subsequently, microservices. At its core, REST leverages the well-established HTTP protocol as its communication backbone. A fundamental principle of REST is the stateless nature of client-server interactions (see Figure 1). This means that each request transmitted from a client to a server must encapsulate all the necessary information for the server to understand and process it, without the server relying on any previously stored client context or session state. This statelessness simplifies server-side design, enhances the scalability of the system by allowing requests to be handled by any available server instance, and contributes to the overall robustness of the architecture.

Another defining characteristic of REST is its emphasis on a uniform interface. This interface is built upon a consistent and predefined set of methods provided by the HTTP protocol, such as GET for retrieving resources, POST for creating new resources, PUT for updating existing ones, and DELETE for removing resources. Resources themselves are uniquely identified through Uniform Resource Locators (URLs), providing a standardized way to address and interact with the different entities within the system. The transfer of state between the client and the server occurs through representations of these resources, commonly using formats like JSON. Furthermore, REST architectures can benefit from explicit caching directives embedded in the responses, allowing intermediary components or clients to store and reuse responses, thereby improving efficiency and reducing the load on the server infrastructure. The layered system aspect of REST allows for the introduction of intermediary components like proxies and gateways, which can enhance scalability, security, and other qualities without requiring the client or server to possess explicit knowledge of these layers. While less frequently utilized in typical microservice scenarios, the optional "code on demand" principle allows servers to extend client functionality by providing executable code.

The widespread adoption of REST has led to a vast ecosystem of tools, libraries, and frameworks, simplifying the development and integration of services. This broad familiarity and extensive tooling offer a relatively low barrier to entry for developers. For basic Create, Read, Update, and Delete (CRUD) operations, REST provides a straightforward and intuitive approach to service design. Its reliance on standard HTTP and well-established conventions makes it advantageous for public-facing APIs where broad client support and discoverability are paramount. Systems where data requirements are relatively predictable and the overhead of potential over-fetching is acceptable can also find REST to be an effective solution. The maturity of HTTP caching mechanisms is a significant benefit for public APIs that serve frequently accessed, relatively static data.

In contrast to REST's resource-centric approach, GraphQL presents itself as a query language specifically designed for APIs, accompanied by a server-side runtime to execute these queries (see Figure 2). Developed internally at Facebook to address the inefficiencies often encountered with traditional REST APIs, GraphQL empowers clients to precisely specify their data requirements. Instead of receiving fixed data structures defined by the server, clients can construct queries that request only the specific fields they need, thus mitigating the problems of over-fetching

(receiving unnecessary data) and under-fetching (needing to make multiple requests to gather all required information). This client-driven approach to data fetching fundamentally alters how clients interact with APIs, leading to significant reductions in bandwidth consumption and improved performance on the client side. For data-hungry applications, GraphQL can drastically reduce network calls.

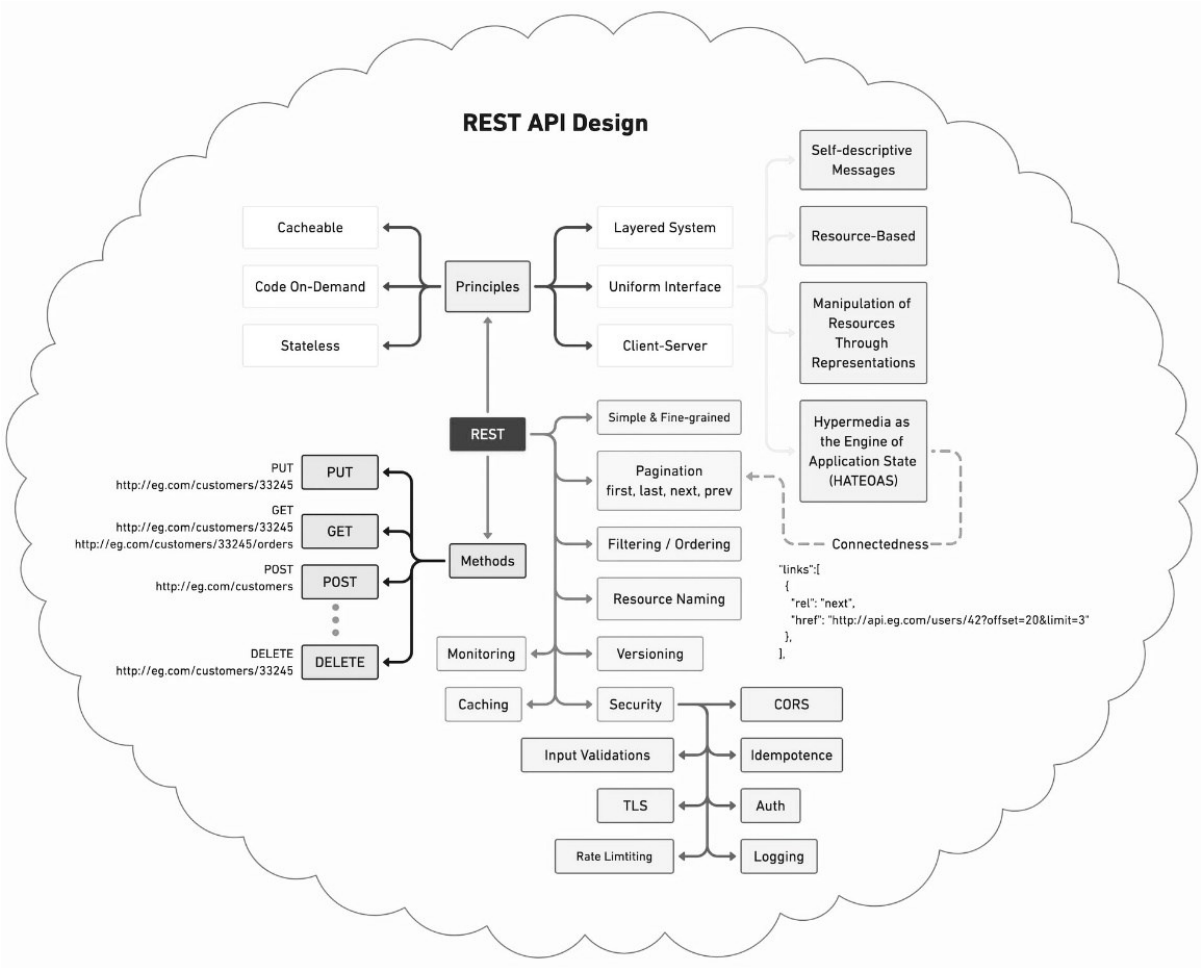


Fig.1. REST API Design (Source: blog.bytebytego.com)

GraphQL

The foundation of a GraphQL API lies in its strongly-typed schema. This schema meticulously defines all the data accessible through the API, outlining the various types of objects, their fields, and the relationships between them. This schema serves as a formal contract between the client and the server, ensuring a shared understanding of the available data landscape. The presence of a robust type system within GraphQL facilitates better communication between client and server development teams, enables earlier detection of errors during development, and supports the use of powerful tooling such as automatic code generation for client libraries and server-side resolvers. This also enhances the developer experience, as clients can tailor queries, simplifying front-end development by reducing data processing and transformation needs.

The foundation of a GraphQL API lies in its strongly-typed schema. This schema meticulously defines all the data accessible through the API, outlining the various types of objects, their fields, and the relationships between them. This schema serves as a formal contract between the client and the server, ensuring a shared understanding of the available data landscape. The presence of a robust type system within GraphQL facilitates better communication between client and server development teams, enables earlier detection of errors during development, and supports the use of powerful tooling such as automatic code generation for client libraries and server-side resolvers. Unlike the multiple endpoints often found in REST APIs, a typical GraphQL API exposes a single endpoint that acts as the entry point for all types of queries, regardless of the specific data being requested. Moreover, GraphQL extends beyond simple request-response interactions by natively supporting subscriptions, allowing clients to receive real-time updates when specific data on the server changes.

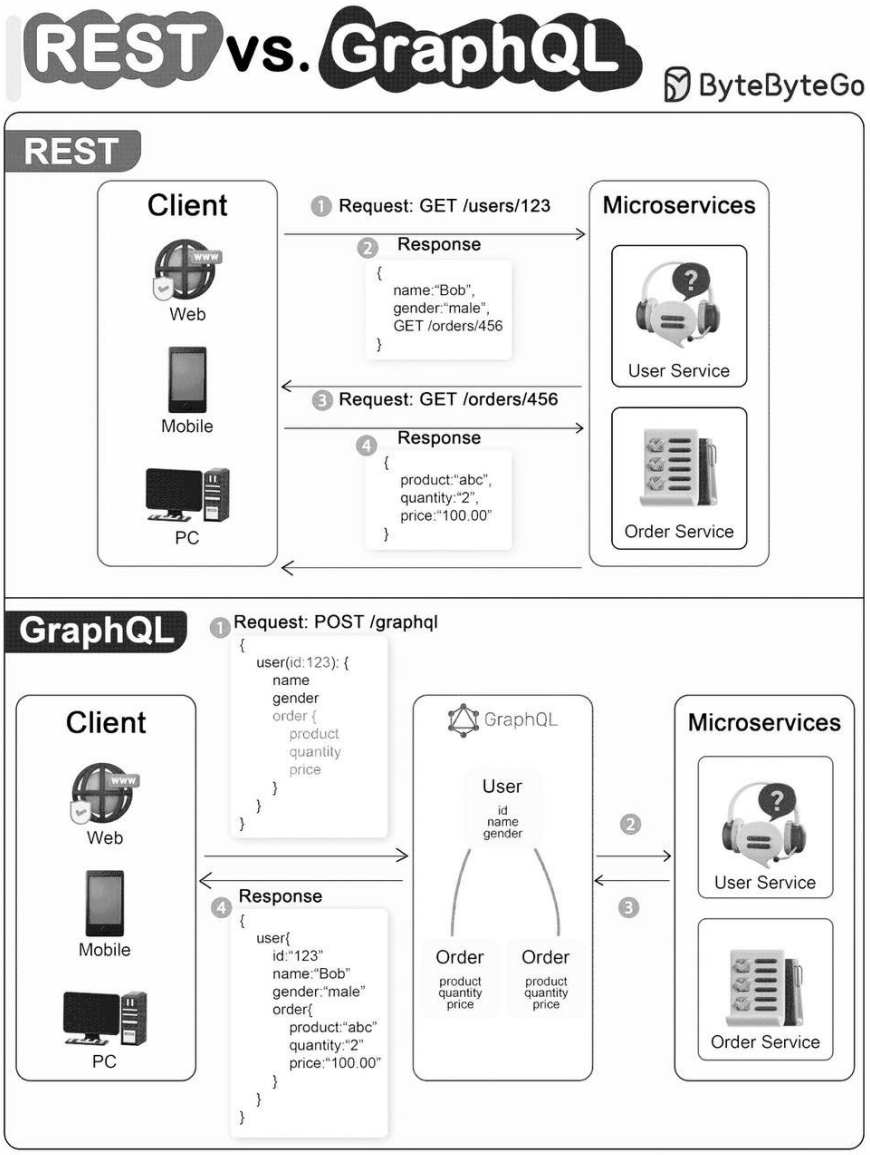


Fig.2. REST API Design (Source: blog.bytebytego.com)

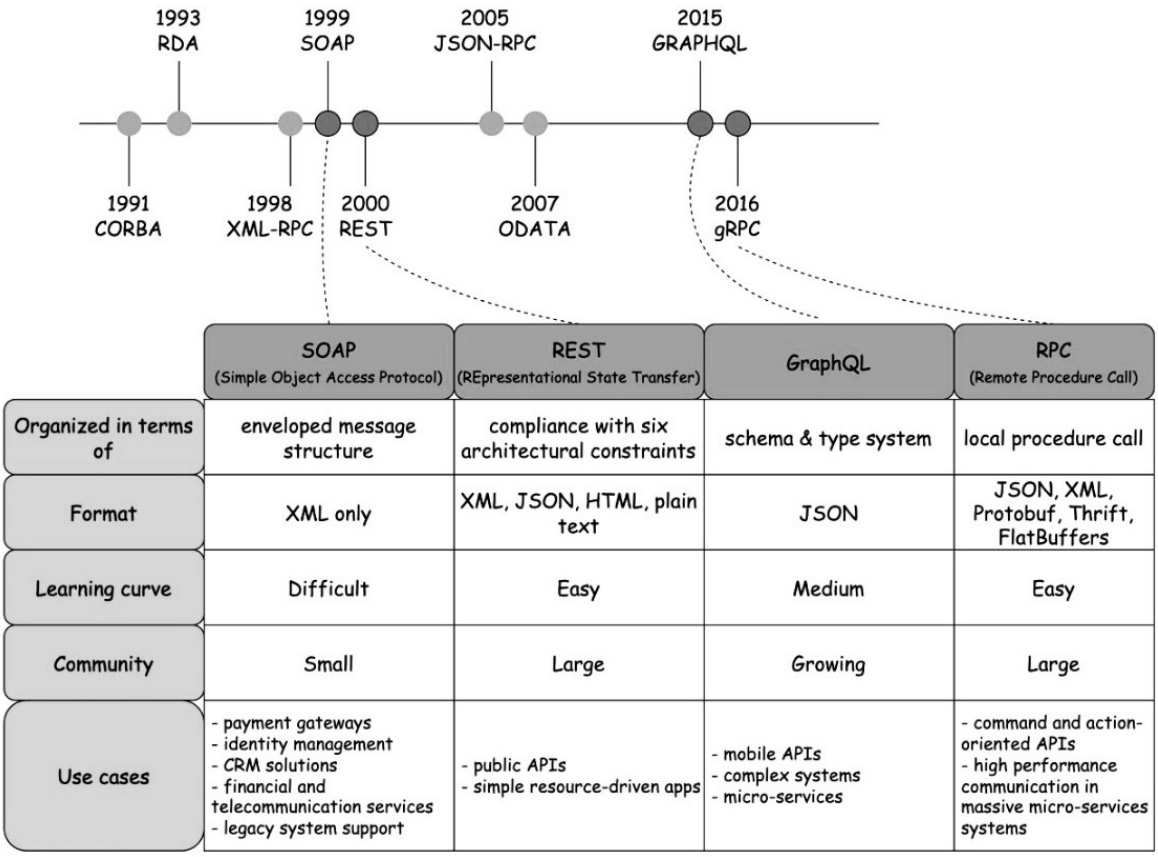
gRPC (Google Remote Procedure Call)

gRPC, developed by Google, offers a high-performance, language-agnostic framework for remote procedure calls. It is built upon modern technologies such as Protocol Buffers as its Interface Definition Language (IDL) and HTTP/2 as its transport protocol. Protocol Buffers provide an efficient and structured way to serialize and deserialize data, leading to smaller message sizes and faster transmission compared to text-based formats like JSON. HTTP/2, the next evolution of the HTTP protocol, offers significant performance improvements over HTTP/1.1, including multiplexing (allowing multiple requests and responses to be sent over a single connection), header compression, and server push capabilities.

A key characteristic of gRPC is its reliance on a contract-first approach. Developers define the service interface using Protocol Buffers, specifying the data structures and the methods (procedures) that can be called remotely. From this definition, client and server stubs (code that facilitates communication) can be automatically generated in various programming languages, streamlining the development process and ensuring strong type safety across service boundaries. The use of HTTP/2 as the underlying transport enables advanced communication patterns, including efficient streaming of data in both directions (client-to-server, server-to-client, and bidirectional). While gRPC excels in performance and supports a wide range of languages, its integration with traditional browser-based clients can be more complex compared to REST or GraphQL [1], often requiring the use of intermediary proxies or other solutions.

Criteria for Comparative Effectiveness Analysis

To effectively compare the suitability of REST, GraphQL, and gRPC for building scalable microservice systems, it is essential to establish a set of relevant criteria (see Figure 2). These criteria will serve as the yardsticks against which each architectural style will be evaluated, allowing for a structured and comprehensive analysis of their strengths and weaknesses in the context of modern distributed applications [3][6]. The following subsections outline the key aspects that will be considered in this comparative assessment.



	SOAP (Simple Object Access Protocol)	REST (REpresentational State Transfer)	GraphQL	RPC (Remote Procedure Call)
Organized in terms of	enveloped message structure	compliance with six architectural constraints	schema & type system	local procedure call
Format	XML only	XML, JSON, HTML, plain text	JSON	JSON, XML, Protobuf, Thrift, FlatBuffers
Learning curve	Difficult	Easy	Medium	Easy
Community	Small	Large	Growing	Large
Use cases	- payment gateways - identity management - CRM solutions - financial and telecommunication services - legacy system support	- public APIs - simple resource-driven apps	- mobile APIs - complex systems - micro-services	- command and action-oriented APIs - high performance communication in massive micro-services systems

Fig.3. Comparison of API architectural styles (Source: altexsoft.com)

Scalability

Scalability refers to the ability of a system to handle an increasing amount of work by adding resources. In the context of microservices, this often involves horizontal scaling, where more instances of services are deployed to distribute the load. The architectural style adopted can significantly influence how easily a system can be scaled. Factors such as statelessness, connection management, and the overhead of communication protocols play a crucial role. We will analyze how REST, GraphQL, and gRPC facilitate or hinder the horizontal and vertical scaling of microservices and examine the common load balancing strategies employed with each style.

Performance

Performance is a critical aspect of any distributed system, encompassing factors such as latency (the time taken for a request to be completed), throughput (the amount of work a system can handle in a given time), and resource utilization (the consumption of CPU, memory, and network bandwidth). The choice of architectural style and its underlying communication protocols can have a profound impact on these metrics. We will compare the performance characteristics of REST (typically using JSON over HTTP/1.1 or HTTP/2), GraphQL (often using JSON over HTTP/1.1 or HTTP/2), and gRPC (using Protocol Buffers over HTTP/2), paying attention to data serialization/deserialization overhead and network efficiency [2][7].

Development and Integration Simplicity

The ease with which developers can build, understand, and maintain microservices, as well as the simplicity of integrating services built with different technologies, is an important consideration. This includes the availability of mature tools and libraries, the learning curve associated with each style, and the conventions and best practices that developers need to adhere to. We will assess the developer-friendliness of REST, GraphQL, and gRPC ecosystems and the challenges involved in integrating them with existing systems or third-party services.

Support for Different Interaction Types

Microservices often need to support various communication patterns beyond the simple request-response paradigm. These can include streaming of data (e.g., for real-time updates or large datasets) and bidirectional communication (where both the client and server can send and receive data streams concurrently). We will examine the native support for different interaction types offered by REST, GraphQL, and gRPC and their implications for building diverse functionalities within a microservice architecture.

Observability

In a distributed system composed of numerous independent services, the ability to monitor the health, performance, and behavior of the system is crucial. Observability encompasses logging, metrics, and tracing, allowing developers and operators to understand what is happening inside the system and diagnose issues effectively. We will

evaluate how well REST, GraphQL, and gRPC lend themselves to observability practices and the availability of tools for monitoring and tracing systems built with each style.

Security

Security is a paramount concern for any application, especially those handling sensitive data or financial transactions, as is common in e-commerce systems (as per your initial context). The architectural style can influence the security mechanisms that can be employed and the inherent security characteristics of the communication. We will discuss the built-in security features and common security considerations for REST, GraphQL, and gRPC, including aspects like authentication, authorization, and protection against common web vulnerabilities.

By analyzing REST, GraphQL, and gRPC against these six key criteria, we aim to provide a comprehensive understanding of their relative strengths and weaknesses in the context of building scalable and effective microservice systems.

Comparative Analysis of the Effectiveness of REST, GraphQL, and gRPC for Microservices

Having established the theoretical foundations and the criteria for comparison, this section will delve into a direct analysis of REST, GraphQL, and gRPC, evaluating their effectiveness across the defined metrics in the context of building scalable microservice systems [4].

Comparative Table Based on Defined Criteria

To provide a concise overview, table 1 summarizes the comparative analysis across the key criteria.

Table 1

Criterion	REST	GraphQL	gRPC
Scalability	Good (stateless, cacheable)	Good (efficient data fetching reduces load)	Excellent (HTTP/2, connection reuse)
Performance	Moderate (JSON overhead, potential over/under-fetching)	Generally good (avoids over/under-fetching), potential for complex queries	Excellent (Protocol Buffers, HTTP/2)
Dev & Integration Simplicity	High (familiar, broad tooling)	Moderate (learning curve, server complexity)	Moderate (contract-first, code generation)
Interaction Types	Primarily Request/Response	Request/Response, Subscriptions	Request/Response, Streaming (uni/bi)
Observability	Good (standard HTTP tooling)	Good (specific GraphQL tools emerging)	Good (standard HTTP/2 tooling, interceptors)
Security	Relies on HTTP security mechanisms	Relies on HTTP security, field-level auth	Relies on HTTP/2 & transport security

Detailed Analysis of Strengths and Weaknesses

Representational State Transfer (REST) has become a cornerstone of web service development, and its application in microservice architectures benefits significantly from its widespread familiarity among developers. The principles of REST, built upon the well-established HTTP protocol, offer a relatively low barrier to entry and are supported by an extensive ecosystem of tools, libraries, and frameworks, simplifying the development and integration of services. Particularly for basic Create, Read, Update, and Delete (CRUD) operations, REST provides a straightforward and intuitive approach to service design. However, a notable drawback of REST lies in its potential for inefficient data retrieval. Clients often face the dilemma of either receiving more data than their application requires (over-fetching) or needing to make multiple requests to different endpoints to gather all the necessary information (under-fetching). These inefficiencies can lead to increased network traffic and higher processing overhead. Furthermore, managing the evolution of RESTful APIs without disrupting existing clients can present considerable challenges, often requiring complex versioning strategies.

In contrast, GraphQL offers a client-centric approach to data fetching, fundamentally altering how clients interact with APIs. One of its key advantages is the ability for clients to precisely specify their data requirements in a query, thus eliminating the inefficiencies associated with over-fetching and under-fetching common in REST. This precise retrieval can lead to significant reductions in bandwidth consumption and improved performance on the client side. The presence of a strongly-typed schema in GraphQL APIs provides a clear and well-defined contract between the client and the server, fostering better communication between development teams and enabling the use of powerful tooling for code generation and validation. Moreover, GraphQL's inherent support for subscriptions makes it a compelling choice for building applications that require real-time data updates. However, the implementation of a GraphQL server typically involves greater complexity compared to setting up a basic REST API. Additionally, the flexibility afforded to clients in constructing queries introduces the potential for poorly optimized or deeply nested requests that can place a significant strain on server resources. Finally, leveraging the built-in caching mechanisms of HTTP is not as straightforward with GraphQL's single endpoint architecture, often necessitating the adoption of alternative caching strategies.

gRPC distinguishes itself in the realm of microservices through its emphasis on high performance and efficient communication. This efficiency is largely achieved through the use of Protocol Buffers as the data

serialization format, which offers significant advantages in terms of message size and processing speed compared to text-based formats like JSON. The underlying transport protocol, HTTP/2, further enhances performance with features such as multiplexing and header compression. A significant strength of gRPC is its contract-first development approach, facilitated by the use of Protocol Buffers as the Interface Definition Language (IDL). This allows for the automatic generation of strongly-typed client and server stubs in a wide range of programming languages, streamlining the development process and ensuring type safety across service boundaries. Furthermore, gRPC provides excellent built-in support for various streaming communication patterns, including unary, server-side, client-side, and bidirectional streaming. Nevertheless, gRPC also presents certain limitations. Direct integration with traditional browser-based clients is not as seamless as with REST or GraphQL, often requiring the implementation of intermediary proxy layers. Additionally, the reliance on Protocol Buffers, a binary format, can make debugging more challenging without specialized tooling, and the concepts and associated tooling of gRPC may present a steeper learning curve for developers primarily familiar with other architectural styles [5].

Discussion of Use Cases

The selection of an architectural style for a microservice system should not be a one-size-fits-all decision. The optimal choice is heavily influenced by the specific requirements, constraints, and context of the application being built. This subsection will explore typical use cases where REST, GraphQL, and gRPC might be the most appropriate architectural styles.

REST often emerges as a suitable choice for microservice systems that primarily involve simple Create, Read, Update, and Delete (CRUD) operations on resources. Its simplicity and the vast ecosystem of tools make it a pragmatic option for services that need to expose basic functionalities. Furthermore, for public-facing APIs where broad client support and discoverability are paramount, REST's reliance on standard HTTP and its well-established conventions are advantageous. Systems where the data requirements of clients are relatively predictable and where the overhead of potential over-fetching is acceptable might also find REST to be a straightforward and effective solution. The maturity of HTTP caching mechanisms can also be a significant benefit for public APIs that serve frequently accessed, relatively static data.

GraphQL, on the other hand, shines in scenarios characterized by complex and evolving data requirements on the client side. Applications such as mobile apps and single-page applications, which often need to fetch specific sets of data tailored to different views or user interactions, can greatly benefit from GraphQL's ability to eliminate over-fetching and under-fetching. In federated data scenarios, where data is aggregated from multiple underlying services, GraphQL's schema stitching capabilities provide a unified and efficient way for clients to query across these diverse sources. Moreover, for public APIs where providing clients with fine-grained control over the data they receive is a key consideration, GraphQL offers a powerful and flexible solution. The real-time capabilities offered by GraphQL subscriptions also make it a strong contender for applications that require live data updates, such as collaborative tools or notification systems.

gRPC typically proves to be most beneficial for high-performance internal communication between microservices where low latency and high throughput are critical. Its use of Protocol Buffers for efficient serialization and HTTP/2 as a high-performance transport protocol makes it well-suited for building backend systems that require rapid and efficient data exchange. In polyglot environments, where microservices are developed using a variety of programming languages, gRPC's support for code generation in numerous languages ensures seamless and strongly-typed communication across service boundaries. Furthermore, applications that require efficient streaming of data, such as media processing pipelines or real-time analytics platforms, can leverage gRPC's robust support for various streaming patterns. While browser integration might require additional effort, for backend-to-backend communication within a microservice architecture, gRPC often provides significant performance advantages.

The choice ultimately depends on a careful evaluation of the specific needs of the system, considering factors such as the complexity of data requirements, performance sensitivity, development team expertise, and the need for real-time communication or broad client compatibility.

Analysis of Real-World Examples of Using REST, GraphQL, and gRPC in Large Microservice Systems

A notable example of modern microservice architecture in e-commerce is an open-source modular backend that integrates REST, GraphQL, and gRPC for different communication needs. In this system, core services such as account management, product catalog, and order processing are implemented as independent Go microservices, while a Python-based recommender service leverages gRPC for efficient, high-performance communication. The platform employs a unified GraphQL API gateway, providing clients with a flexible and efficient interface to query and mutate data across all backend services. Event streaming is handled through Kafka, enabling asynchronous updates and decoupled service interactions. This hybrid approach demonstrates how REST, GraphQL, and gRPC can be combined to optimize for developer experience, performance, and scalability in a real-world e-commerce platform.

Examination of Companies' Experiences Regarding the Selection and Implementation of Different Architectural Styles, Their Challenges, and Achievements

Industry case studies reveal that transitioning from monolithic to microservice architectures yields significant benefits, such as increased productivity, improved deployment frequency, and reduced downtime. For example, a leading e-commerce company reported a 30% reduction in downtime and faster feature rollouts after adopting microservices, enabled by independent service updates and robust orchestration tools. However, these gains come with challenges, including defining optimal service boundaries, managing distributed data consistency, and ensuring

effective monitoring. Companies often struggle with workforce reskilling and the complexity of testing in distributed environments. Successful organizations address these issues by adopting domain-driven design, leveraging cloud-native technologies, and investing in team collaboration and monitoring infrastructure. These strategies help balance the benefits of agility and scalability with the realities of integration and operational complexity.

Another prominent example is the experience of social networking platforms that have migrated to microservices to handle massive user loads and complex feature sets. For instance, companies like Facebook and Twitter initially relied on monolithic architectures but gradually transitioned to microservices to support rapid growth and evolving requirements. RESTful APIs were widely adopted for their simplicity and broad compatibility, especially for public-facing services and mobile applications. However, as the number of internal service-to-service calls increased, issues such as latency and payload overhead became more pronounced.

To address these challenges, some companies introduced gRPC for internal communications, benefiting from its high performance, efficient binary serialization, and support for streaming data. This shift enabled faster inter-service communication and reduced infrastructure costs. At the same time, GraphQL emerged as a popular choice for aggregating data from multiple microservices, providing clients with the ability to request exactly the data they needed and reducing over-fetching. For example, Shopify implemented GraphQL to streamline its storefront API, resulting in improved developer experience and more responsive applications.

Despite these achievements, organizations encountered several obstacles during implementation. These included the need to manage multiple API paradigms, ensure backward compatibility, and maintain comprehensive documentation. Additionally, operational challenges such as distributed tracing, monitoring, and securing service-to-service communication required significant investment in tooling and process improvements. Companies that successfully overcame these hurdles often did so by standardizing API gateways, adopting service meshes for observability and security, and fostering a culture of continuous learning and cross-team collaboration.

Overall, real-world case studies demonstrate that the selection and integration of REST, GraphQL, and gRPC in large-scale microservice systems is driven by specific business needs, technical constraints, and organizational maturity. The most successful implementations are those that carefully evaluate the trade-offs of each technology, invest in robust infrastructure, and prioritize clear communication and teamwork throughout the development lifecycle.

Challenges and Future Research Directions

While REST, GraphQL, and gRPC have proven to be effective architectural styles for building microservice systems, each also presents its own set of challenges and areas where further research and development are needed. This section will explore some of these existing problems and limitations, as well as potential future directions for research in the field of distributed system architecture, particularly concerning these communication styles.

One of the ongoing challenges with REST lies in effectively addressing the issues of over-fetching and under-fetching in more complex scenarios. While various strategies like HATEOAS and sparse fieldsets exist, they often add complexity to both the server and client implementations and are not universally adopted. Future research could explore more standardized and efficient ways to allow clients to specify their data requirements in RESTful APIs without significant overhead. Furthermore, the evolution and versioning of RESTful APIs remain a complex problem, and new approaches or tooling to manage these transitions more smoothly would be valuable.

GraphQL, despite its strengths in data fetching efficiency, faces challenges related to query optimization and security. As query complexity increases, ensuring efficient data retrieval on the server becomes crucial, and research into advanced query optimization techniques and cost analysis for GraphQL execution is an important area. Security considerations, particularly around preventing malicious or overly complex queries from impacting server performance, also warrant further investigation. Additionally, the development of more robust and standardized caching solutions for GraphQL APIs is an ongoing area of interest.

gRPC, while excelling in performance for internal microservice communication, still faces challenges in terms of broader client compatibility, especially with web browsers. While solutions like gRPC-Web exist, they often introduce additional layers and potential performance trade-offs. Future research could focus on improving the interoperability of gRPC with web-based clients and exploring more efficient ways to bridge the gap between the binary nature of Protocol Buffers and the text-based expectations of web browsers. Furthermore, enhancing the debugging and observability tooling for gRPC in complex distributed environments remains an important area.

More broadly, the increasing adoption of hybrid architectural approaches, where microservices are combined with other paradigms like serverless computing or event-driven architectures, presents new challenges for inter-service communication. Research into how REST, GraphQL, and gRPC can be effectively integrated with these emerging patterns, and what new communication styles might be necessary, is crucial. Additionally, the application of artificial intelligence and machine learning techniques to optimize communication patterns, predict potential performance bottlenecks, or enhance the security of microservice interactions represents a promising avenue for future research. Finally, the development of more comprehensive frameworks and best practices for choosing the most appropriate architectural style based on specific application requirements and context remains an important area for the community.

Conclusion

The choice of architectural style is a critical decision in the design and implementation of scalable microservice systems. This paper has provided a comparative analysis of three prominent architectural styles: REST,

GraphQL, and gRPC, evaluating their effectiveness based on key criteria such as scalability, performance, development and integration simplicity, support for different interaction types, observability, and security.

Our analysis reveals that each architectural style offers distinct advantages and disadvantages, making their suitability highly dependent on the specific context and requirements of the application. REST, with its simplicity and broad ecosystem, remains a strong contender for basic CRUD operations and public-facing APIs where discoverability is key. However, its potential for over-fetching and under-fetching can lead to inefficiencies in more complex scenarios. GraphQL offers a client-driven approach to data fetching, providing significant benefits in terms of data retrieval efficiency and flexibility, particularly for applications with evolving data needs. However, it introduces increased server-side complexity and poses unique challenges for caching. gRPC stands out for its high performance and efficient communication, making it well-suited for internal microservice interactions where low latency and high throughput are paramount, although its browser integration can be complex.

Ultimately, the selection of the most appropriate architectural style requires a careful evaluation of the trade-offs between these factors. There is no universally superior style; rather, the optimal choice depends on a deep understanding of the application's specific needs, the expertise of the development team, and the overall architectural goals. As the landscape of distributed systems continues to evolve, future research should focus on addressing the existing challenges and exploring new approaches to inter-service communication, potentially leading to hybrid models and more context-aware architectural decisions.

This comparative analysis provides a foundation for architects and developers to make informed decisions when designing and building scalable microservice systems, ultimately contributing to more efficient, robust, and maintainable applications.

Література

1. Олівос І., Йоханссон М. Порівняльне дослідження REST та gRPC для мікросервісів у встановлених архітектурах програмного забезпечення / І. Олівос, М. Йоханссон // Університет Лінчепінгу, Швеція. – 2023. – 46 с. – Режим доступу: https://www.diva-portal.org/smash/get/diva2:1772587/FULLTEXT01.pdf

2. Нісвар М., Сафруддін Р. А., Бустамін А., Асвад І. Оцінка продуктивності мікросервісної комунікації за допомогою REST, GraphQL та gRPC / М. Нісвар, Р. А. Сафруддін, А. Бустамін, І. Асвад // International Journal of Electronics and Telecommunications. – 2024. – Vol. 70, No. 2. – С. 429–438. – doi: 10.24425/ijet.2024.149562

3. Порівняльний аналіз RESTful, GraphQL та gRPC API: Аналіз продуктивності за результатами навантажувального та стрес-тестування // Jurnal Sistem Informasi dan Komputer Akuntansi. – 2025. – Vol. 14, No. 1. – С. 1–10. – doi: 10.32736/sisfokom.v14i1.2375

4. REST, GraphQL, RPC, gRPC: Порівняльний аналіз / Rahul Vijayvergiya // Dev.to. – 2024. – Режим доступу: https://dev.to/rahulvijayvergiya/rest-graphql-rpc-grpc-3m09

5. Порівняння продуктивності REST проти GraphQL API / Проект бакалаврської дисертації. – Стокгольм: Королівський технологічний інститут KTH, 2023. – 40 с. – Режим доступу: https://www.diva-portal.org/smash/get/diva2:1768044/FULLTEXT01.pdf

6. Порівняння SOAP vs REST vs GraphQL vs RPC API / AltexSoft. – 2020. – Режим доступу: https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/

7. Оцінка продуктивності мікросервісів за допомогою REST, GraphQL та gRPC / Semantic Scholar. – 2024. – Режим доступу: https://www.semanticscholar.org/paper/Performance-evaluation-of-microservices-with-REST,-Niswar-Safruddin/8526bef65e8d241155be2ce807989afc146f0b18

References

1. Olivos I., Johansson M. Comparative Study of REST and gRPC for Microservices in Established Software Architectures / I. Olivos, M. Johansson // Linköping University, Sweden. – 2023. – 46 c. – Access mode: https://www.diva-portal.org/smash/get/diva2:1772587/FULLTEXT01.pdf

2. Niswar M., Safruddin R. A., Bustamin A., Aswad I. Performance evaluation of microservices communication with REST, GraphQL, and gRPC / M. Niswar, R. A. Safruddin, A. Bustamin, I. Aswad // International Journal of Electronics and Telecommunications. – 2024. – Vol. 70, No. 2. – С. 429–438. – doi: 10.24425/ijet.2024.149562

3. Comparative Analysis of RESTful, GraphQL, and gRPC APIs: Perfomance Insight from Load and Stress Testing // Jurnal Sistem Informasi dan Komputer Akuntansi. – 2025. – Vol. 14, No. 1. – С. 1–10. – doi: 10.32736/sisfokom.v14i1.2375

4. REST, GraphQL, RPC, gRPC: Comparative Analysis / Rahul Vijayvergiya // Dev.to. – 2024. – Access mode: https://dev.to/rahulvijayvergiya/rest-graphql-rpc-grpc-3m09

5. Performance comparison of REST vs GraphQL APIs / Bachelor Degree Project. – Stockholm: KTH Royal Institute of Technology, 2023. – 40 с. – Access mode: https://www.diva-portal.org/smash/get/diva2:1768044/FULLTEXT01.pdf

6. Comparing SOAP vs REST vs GraphQL vs RPC API / AltexSoft. – 2020. – Access mode: https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/

7. Performance evaluation of microservices with REST, GraphQL, and gRPC / Semantic Scholar. – 2024. – Access mode: https://www.semanticscholar.org/paper/Performance-evaluation-of-microservices-with-REST,-Niswar-Safruddin/8526bef65e8d241155be2ce807989afc146f0b18