International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

(Research Article)

A Comparative Design Study of REST, GraphQL and gRPC APIs for Large-Scale Applications

Ramesh Tangudu *

Enterprise Architect and Application Development Lead, TX, USA.

International Journal of Science and Research Archive, 2026, 18(03), 102-111

Publication history: Received on 20 January 2026; revised on 25 February 2026; accepted on 28 February 2026

Article DOI: https://doi.org/10.30574/ijsra.2026.18.3.0416

Abstract

Aim. This study aims to comparatively analyze REST, GraphQL, and gRPC API paradigms to guide architectural decision-making in large-scale, distributed applications. It focuses on understanding how each paradigm addresses scalability, performance, and maintainability challenges. The goal is to provide practical insights for system architects. Emphasis is placed on real-world constraints such as latency, data consistency, and team productivity. The study targets enterprise and cloud-native systems. It seeks to bridge theory with applied design choices.

Method. A design-oriented comparative methodology is adopted, combining architectural analysis with qualitative and quantitative criteria. Core dimensions include communication models, data serialization, performance characteristics, and developer experience. Design patterns and protocol behaviors are examined under large-scale assumptions. Conceptual Figures are used to illustrate system interactions. Comparative tables summarize strengths and trade-offs. The approach emphasizes reproducibility and clarity.

Results. REST demonstrates simplicity and broad interoperability but shows limitations in over-fetching and versioning at scale. GraphQL provides flexible data querying and reduced payloads, improving client efficiency. gRPC achieves superior performance through binary serialization and HTTP/2, excelling in internal microservices. Each paradigm shows distinct advantages depending on workload patterns. No single approach dominates all dimensions. Hybrid adoption emerges as common in practice.

Conclusion. The study concludes that API selection should be context-driven rather than universal. REST remains suitable for public-facing services, GraphQL for complex client-driven data needs, and gRPC for high-performance internal communication. Large-scale systems benefit from combining paradigms strategically. Architectural clarity and governance are critical for success. Future systems will increasingly adopt polyglot API strategies.

Keywords: REST; GraphQL; gRPC; API Design; Large-Scale Systems; Distributed Architecture

1. Introduction

Application Programming Interfaces (APIs) have become a fundamental building block of modern software systems, enabling communication and integration among distributed components. In large-scale applications, APIs act as formal contracts that define how services interact, exchange data, and evolve over time. As systems grow in size and complexity, the design of APIs directly influences performance, scalability, security, and maintainability. Well-designed APIs reduce coupling between components, facilitate independent deployment, and support long-term system evolution, making API design a strategic architectural concern rather than a purely technical choice.

* Corresponding author: Ramesh Tangudu

The rapid adoption of cloud computing, microservices architectures, and mobile and web-based clients has significantly increased the demand for efficient and flexible APIs. Traditional monolithic systems have been replaced by distributed architectures where hundreds or even thousands of services interact continuously. In such environments, APIs must handle high request volumes, heterogeneous clients, and dynamic workloads. This shift has exposed limitations in earlier API approaches and motivated the development of alternative paradigms that better address data efficiency, latency, and scalability requirements.

Representational State Transfer (REST) has long been the dominant API design style due to its simplicity, statelessness, and reliance on standard HTTP mechanisms. Its widespread adoption has resulted in extensive tooling, documentation practices, and developer familiarity. However, as applications scale, REST-based systems often suffer from challenges such as over-fetching or under-fetching of data, complex versioning strategies, and inefficient client–server interactions. These challenges have prompted researchers and practitioners to explore more expressive and performance-oriented API models.

GraphQL and gRPC have emerged as prominent alternatives to REST, each addressing specific limitations observed in large-scale systems. GraphQL introduces a schema-driven, query-based approach that allows clients to request precisely the data they need, reducing network overhead and improving client flexibility. In contrast, gRPC adopts a remote procedure call (RPC) model built on HTTP/2 and binary serialization, prioritizing high performance, low latency, and strong typing. Both paradigms reflect a shift toward more specialized API designs tailored to modern distributed architectures.

Despite the growing adoption of GraphQL and gRPC, there is no universal consensus on which API paradigm is best suited for large-scale applications. Each approach involves trade-offs related to architectural complexity, learning curve, tooling maturity, and operational overhead. In practice, organizations often struggle to select an API style that aligns with their performance goals, team expertise, and long-term maintenance strategies. A systematic and comparative understanding of these paradigms is therefore essential to support informed decision-making.

2. Background and API Paradigms

Application Programming Interfaces are structured mechanisms that enable software components to communicate across process, platform, and network boundaries. In large-scale systems, APIs are not merely technical interfaces but architectural abstractions that encode design decisions related to data representation, communication style, and system evolution. The background of API paradigms is closely tied to the broader evolution of distributed systems, where the need for interoperability, scalability, and loose coupling has continuously shaped how APIs are designed and consumed. Early distributed applications relied heavily on tightly coupled communication mechanisms such as remote procedure calls and proprietary middleware. While these approaches offered efficiency, they often suffered from poor interoperability and limited scalability. With the rise of the World Wide Web, HTTP emerged as a universal communication protocol, enabling a new generation of web-based APIs. This shift emphasized simplicity, statelessness, and standardization, laying the groundwork for modern API paradigms that prioritize openness and ease of integration.

Representational State Transfer (REST) emerged as a dominant architectural style in this context, promoting a resource-oriented view of systems. In REST, all interactions revolve around resources identified by uniform resource identifiers (URIs), and standard HTTP methods are used to manipulate these resources. This approach aligns naturally with the web's architecture and enables features such as caching, layered systems, and stateless interactions. As a result, REST became widely adopted for public and enterprise APIs alike.

Despite its advantages, REST introduces structural constraints that become more pronounced in large-scale applications. The rigid coupling between endpoints and resource representations can lead to inefficiencies when clients require flexible or aggregated data. Additionally, managing API evolution through versioning becomes increasingly complex as the number of consumers grows. These challenges reveal a gap between REST's original design assumptions and the dynamic data requirements of modern, client-diverse systems. GraphQL was introduced to address many of these limitations by shifting control over data retrieval from the server to the client. Instead of exposing multiple endpoints, GraphQL provides a single endpoint backed by a strongly typed schema. Clients specify exactly what data they need through declarative queries, reducing over-fetching and under-fetching problems. This paradigm is particularly well-suited for applications with complex user interfaces and multiple client platforms, where data requirements vary significantly.

In parallel, gRPC represents a return to procedure-oriented communication, but with modern enhancements. Built on HTTP/2 and using efficient binary serialization, gRPC emphasizes performance, low latency, and strong typing through explicit service contracts. It is especially attractive for internal service-to-service communication in microservices architectures, where efficiency and reliability are more critical than broad interoperability or human-readable payloads.

Table 1 Core Characteristics of API Paradigms

Aspect	REST	GraphQL	gRPC
Interaction Style	Resource-based	Query-based	RPC-based
Data Format	JSON/XML	JSON	Protobuf
Transport	HTTP/1.1	HTTP	HTTP/2
Typical Use	Public APIs	Client-driven apps	Internal services

3. Architectural Design Comparison

The architectural design of an API plays a critical role in shaping how large-scale applications are structured, deployed, and evolved. APIs influence service boundaries, communication paths, and dependency management across distributed components. In large-scale systems, architectural decisions must accommodate scalability, fault tolerance, and independent service evolution. REST, GraphQL, and gRPC embody distinct architectural philosophies that lead to different system structures and trade-offs when applied at scale. REST-based architectures typically emphasize a layered and resource-centric design. Services expose collections of resources through well-defined endpoints, and clients interact with these resources using standardized HTTP methods. This architectural approach encourages loose coupling between clients and servers, as long as the resource contracts remain stable. REST also aligns well with intermediary components such as proxies, caches, and gateways, which can be inserted transparently into the architecture to improve scalability and reliability.

In contrast, GraphQL introduces a more centralized architectural model built around a unified schema. Instead of exposing multiple resource-specific endpoints, GraphQL architectures often include an aggregation layer that sits between clients and backend services. This layer resolves client queries by orchestrating calls to multiple underlying services. While this design provides clients with significant flexibility, it also introduces architectural complexity, as the schema layer becomes a critical component that must be carefully governed and scaled.

gRPC-based architectures are typically designed around service-oriented or microservices principles, where each service exposes a set of strongly typed remote procedures. The architecture emphasizes explicit contracts defined using interface definition languages, which are shared between clients and servers. This approach results in tightly defined service boundaries and promotes consistency across teams. However, it can also increase coupling, as changes to service contracts often require coordinated updates across dependent services.

Coupling and cohesion differ significantly across these paradigms. REST promotes lower coupling through generic interfaces and uniform operations, but may sacrifice expressiveness. GraphQL increases semantic coupling through its centralized schema, while improving data cohesion from the client's perspective. gRPC achieves high cohesion within services through explicit contracts, but introduces stronger coupling between communicating components. These differences directly affect how easily systems can be refactored or extended over time. From an evolutionary perspective, architectural flexibility is a key concern in large-scale systems. REST architectures often rely on versioning strategies to manage change, which can lead to long-term maintenance overhead. GraphQL supports incremental evolution through schema extensions and deprecation, enabling smoother transitions. gRPC relies on careful contract evolution and backward compatibility rules, requiring disciplined governance but offering predictable behavior.

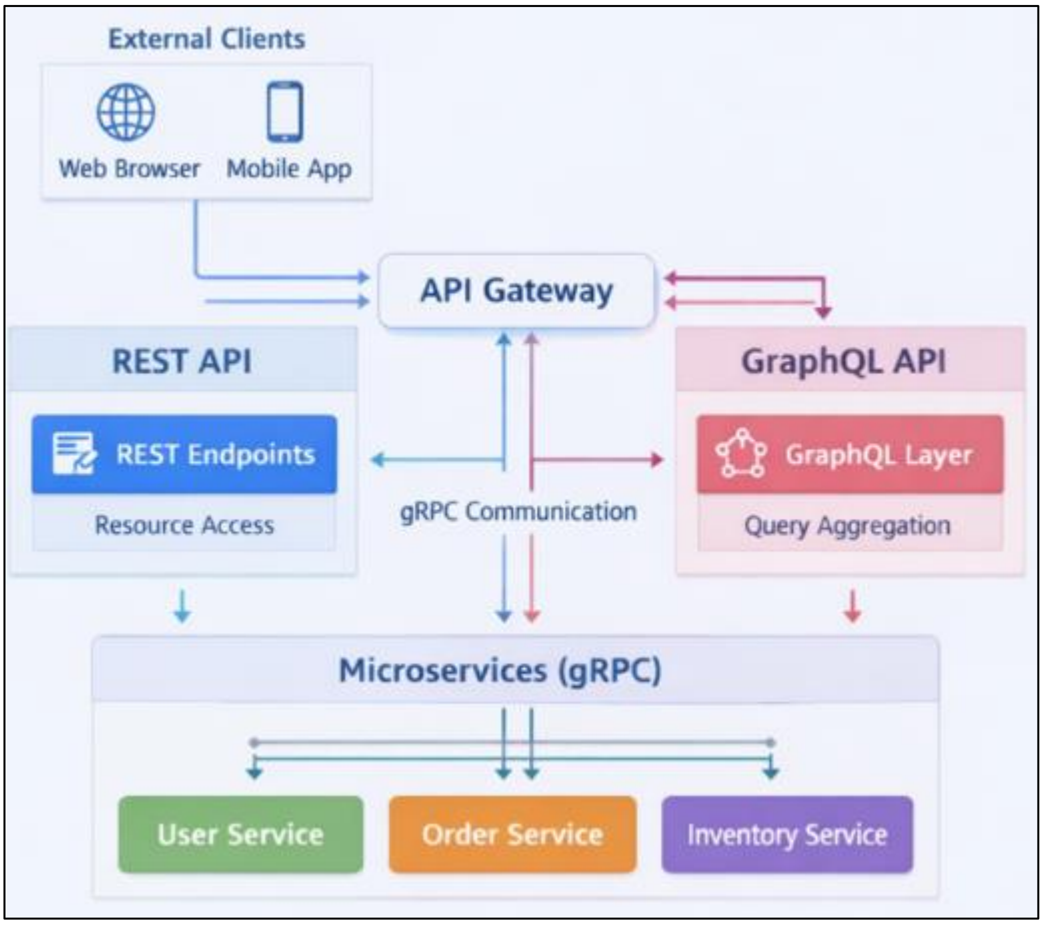

Figure 1 High-Level API Architecture Figure (REST, GraphQL, and gRPC Integration)

This Figure 1 shows a large-scale system architecture that integrates REST, GraphQL, and gRPC within a single application ecosystem. External clients such as web browsers and mobile applications interact with the system through REST and GraphQL APIs exposed via an API gateway. REST endpoints provide standardized access to resources, while the GraphQL layer acts as an aggregation and orchestration component, resolving client queries by communicating with multiple backend services. Internally, microservices communicate with one another using gRPC to achieve low-latency and high-throughput interactions. The Figure 1 shows how different API paradigms coexist, showing clear separation between external-facing interfaces and internal service-to-service communication, thereby demonstrating a hybrid architecture commonly adopted in large-scale applications.

Table 2 Architectural Implications

Criterion	**REST**	**GraphQL**	**gRPC**
Coupling	Low	Medium	High
Schema Centralization	No	Yes	Yes
Gateway Usage	Common	Essential	Optional
Versioning Strategy	URL/Header	Schema evolution	Contract-based

4. Communication Models and Data Handling

Communication models define how clients and services exchange information, and they are central to the efficiency and reliability of large-scale applications. In distributed systems, communication overhead, data serialization, and interaction patterns directly affect latency, throughput, and resource consumption. REST, GraphQL, and gRPC adopt fundamentally different communication models, each reflecting distinct assumptions about client needs, network conditions, and system boundaries. Understanding these models is essential for evaluating their suitability in large-scale environments.

REST primarily follows a stateless request–response communication model built on top of HTTP. Clients interact with servers by sending requests to specific resource endpoints, and servers respond with full representations of those resources. This simplicity makes REST easy to understand and widely interoperable. However, the stateless nature of REST means that each request must contain all necessary context, which can increase payload sizes and network overhead in data-intensive applications. A challenge in REST-based communication is data granularity. Since endpoints typically return fixed resource representations, clients may receive more data than required (over-fetching) or need to make multiple requests to gather related data (under-fetching). In large-scale systems with diverse clients, such inefficiencies can significantly impact performance and increase server load. Although techniques such as query parameters and custom endpoints can mitigate these issues, they often complicate API design and maintenance.

GraphQL introduces a fundamentally different communication model by allowing clients to specify their exact data requirements through declarative queries. Instead of multiple endpoints, GraphQL exposes a single endpoint that interprets queries against a strongly typed schema. The server resolves these queries by fetching and assembling data from underlying services or databases. This model reduces unnecessary data transfer and minimizes the number of network round trips, which is particularly beneficial for applications with complex and dynamic data needs.

The client-driven nature of GraphQL shifts responsibility for data shaping from the server to the client. While this improves flexibility and efficiency, it also introduces new challenges in query execution and performance optimization. Complex queries may result in expensive server-side operations, such as deep resolver chains or inefficient data fetching patterns. As a result, large-scale GraphQL systems require careful query validation, batching, and caching strategies to maintain predictable performance. gRPC adopts a remote procedure call communication model that emphasizes efficiency and strong typing. Clients invoke methods defined in service contracts, and data is exchanged using compact binary serialization formats. This approach minimizes payload size and leverages persistent connections provided by modern transport protocols. gRPC also supports advanced communication patterns, such as streaming, enabling continuous data exchange between services, which is particularly valuable in real-time or high-throughput scenarios.

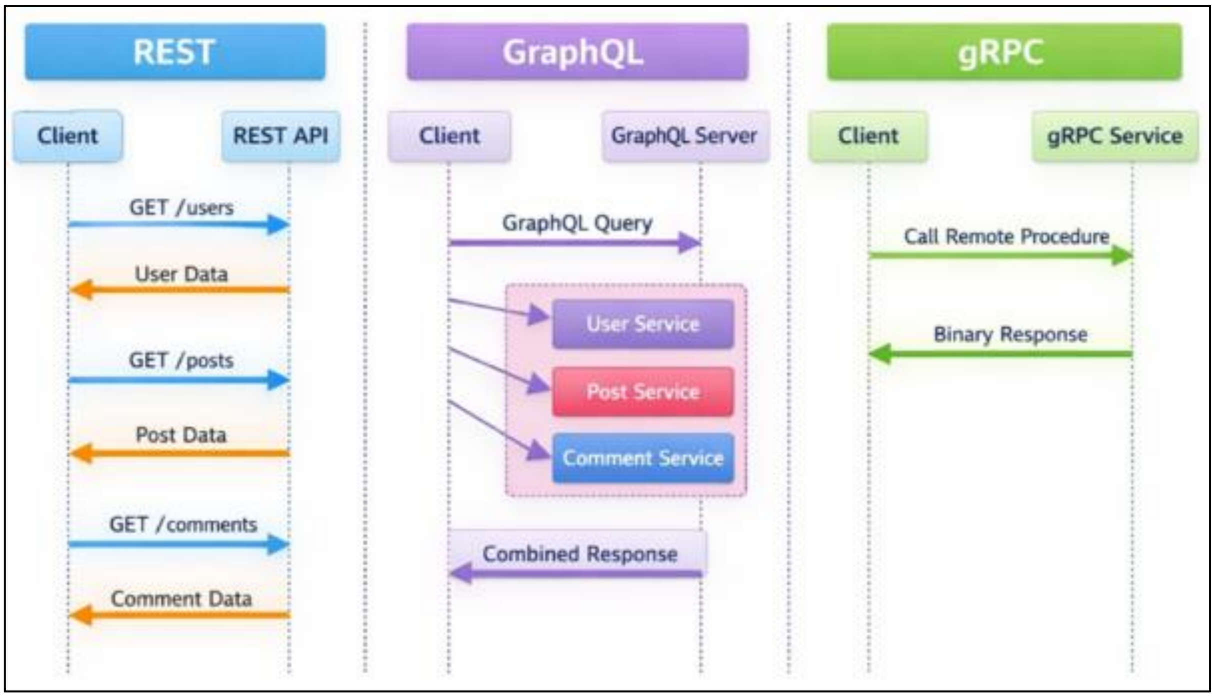

Figure 2 Request–Response Sequence Figure for Communication Models

This Figure 2 compares the communication flow of REST, GraphQL, and gRPC using a sequence-based representation. In the REST flow, the client issues multiple HTTP requests to different resource endpoints, each returning a fixed data structure, which may result in over-fetching or under-fetching of data. In the GraphQL flow, the client sends a single query to a unified endpoint, and the server resolves this query by fetching data from multiple services before returning a precisely shaped response. In the gRPC flow, the client directly invokes a remote procedure defined in a service contract, receiving a compact binary response over a persistent connection. The Figure 2 emphasizes differences in interaction patterns, number of network round trips, and data handling efficiency across the three paradigms.

5. Performance, Scalability, and Reliability

5.1. Performance Requirements in Large-Scale Applications

Large-scale applications operate under strict performance requirements, where APIs must efficiently handle millions of requests, diverse client types, and varying data volumes. Performance is commonly measured in terms of latency, throughput, and resource utilization. API design decisions directly affect these metrics, as communication overhead, serialization formats, and interaction patterns determine how quickly and efficiently data can be exchanged. As system scale increases, even minor inefficiencies in API design can lead to significant performance degradation.

5.2. REST: Performance Characteristics and Scalability

REST-based APIs benefit from their stateless design, which simplifies performance optimization and horizontal scaling. Since each request is independent, REST services can be replicated easily, allowing load balancers to distribute traffic across multiple instances. HTTP-level caching mechanisms, such as cache-control headers and content delivery networks, further improve performance by reducing redundant server processing. These features make REST particularly effective for large-scale, read-heavy workloads with predictable access patterns.

5.3. GraphQL: Performance Trade-offs and Optimization

GraphQL improves performance from the client’s perspective by allowing precise data retrieval, thereby reducing payload size and network round trips. This is especially advantageous for applications with complex user interfaces and multiple client platforms. However, the flexibility of GraphQL queries can lead to unpredictable server-side execution costs. Without proper controls, complex or deeply nested queries may negatively impact performance. Consequently, large-scale GraphQL systems rely on query complexity limits, caching strategies, and efficient resolver implementations to maintain stable performance.

5.4. gRPC: High-Performance Communication Model

gRPC is designed to maximize performance and efficiency, making it well-suited for large-scale, latency-sensitive environments. Its use of compact binary serialization significantly reduces message size compared to text-based formats. Persistent connections and support for multiplexing allow multiple requests to be processed concurrently, improving throughput. These characteristics make gRPC particularly effective for internal service-to-service communication, where performance and efficiency are prioritized over human readability.

5.5. Scalability Considerations Across Paradigms

Scalability in REST systems is largely achieved through replication, caching, and stateless service design. GraphQL scalability depends on the ability to manage query execution costs and distribute resolver workloads effectively. In gRPC-based systems, scalability is achieved through efficient connection management and lightweight communication, but often requires careful orchestration and service discovery. Each paradigm scales differently, and the choice depends on workload characteristics and architectural goals.

5.6. Reliability and Fault Tolerance

Reliability in large-scale systems requires APIs to handle failures gracefully and maintain consistent service availability. REST benefits from mature fault-tolerance patterns such as retries, circuit breakers, and timeouts. GraphQL introduces additional reliability considerations, as a single query may depend on multiple backend services, increasing the risk of cascading failures. gRPC supports robust error handling and deadlines, enabling precise control over failure behavior in distributed environments. Effective reliability strategies must be integrated alongside API design choices.

5.7. Comparative Perspective

From a comparative standpoint, REST provides stable and predictable performance with strong scalability support through standard web infrastructure. GraphQL offers improved data efficiency at the cost of increased operational complexity. gRPC delivers superior raw performance and reliability for internal communication but requires stricter governance and tooling. In large-scale applications, combining these paradigms strategically allows organizations to balance performance, scalability, and reliability across different system layers.

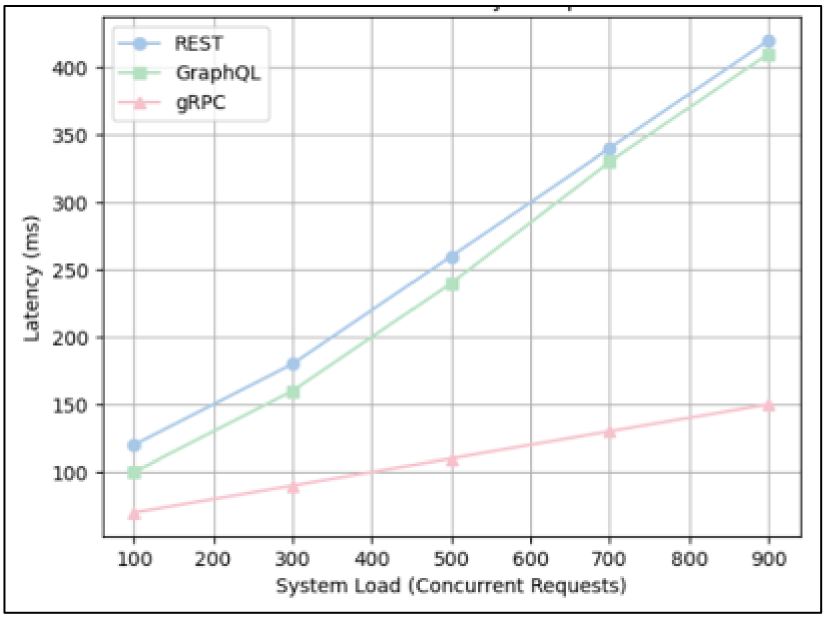

Figure 3 Performance and Scalability Comparison Chart

This Figure 3 presents a comparative visualization of performance metrics such as latency, throughput, and scalability for REST, GraphQL, and gRPC under increasing system load. REST shows moderate latency and stable scalability due to statelessness and caching, but performance declines as the number of client requests increases. GraphQL demonstrates improved client-side efficiency by reducing payload size, though server-side processing costs increase with query complexity. gRPC exhibits the lowest latency and highest throughput, maintaining consistent performance even under heavy load due to binary serialization and multiplexed connections. The Figure 3 effectively conveys how each API paradigm behaves under scale, supporting the conclusion that gRPC excels in performance-critical scenarios while REST and GraphQL balance scalability with flexibility.

Table 3 Performance Considerations

Metric	**REST**	**GraphQL**	**gRPC**
Latency	Medium	Medium	Low
Throughput	Medium	Medium	High
Caching Support	Strong	Limited	Limited
Streaming	Limited	No	Full

6. Security and Governance Considerations

Security and governance are essential concerns in large-scale applications, where APIs often expose critical business logic and sensitive data. As APIs become the primary access point for services, they also become attractive targets for attacks. Effective security design must therefore be integrated at both architectural and operational levels. Governance complements security by ensuring consistent policies, controlled evolution, and compliance across the API ecosystem.

Authentication and authorization form the foundation of API security. REST-based systems commonly rely on standardized mechanisms such as token-based authentication and delegated authorization models. These approaches integrate well with existing identity providers and are supported by a wide range of tools and frameworks. Their widespread adoption makes them suitable for public-facing APIs, but misconfiguration or inconsistent enforcement can introduce vulnerabilities at scale. GraphQL introduces unique security considerations due to its flexible query model. While it typically uses the same authentication and authorization mechanisms as REST, the ability for clients to construct complex queries increases the potential attack surface. Malicious or poorly designed queries can consume excessive server resources or expose unintended data paths. As a result, GraphQL systems require additional safeguards such as query complexity limits, depth restrictions, and fine-grained field-level authorization.

gRPC emphasizes secure, service-to-service communication and commonly relies on strong transport-level security. Mutual authentication mechanisms ensure that only trusted services can communicate with one another, making gRPC well-suited for internal system boundaries. Its strongly typed contracts also reduce the risk of malformed requests. However, the binary nature of gRPC traffic can make inspection and debugging more challenging, requiring specialized tooling for monitoring and security analysis.

Governance plays a critical role in maintaining consistency and control as API ecosystems grow. In REST-based systems, governance often involves versioning strategies, documentation standards, and deprecation policies. GraphQL governance centers on schema management, including controlled schema evolution and clear ownership of types and fields. gRPC governance relies on disciplined contract management and backward compatibility rules to prevent breaking changes across dependent services. At scale, operational security challenges such as monitoring, auditing, and compliance become increasingly complex. APIs must be observable to detect anomalies, enforce rate limits, and respond to incidents effectively. REST benefits from mature monitoring and gateway solutions, GraphQL requires specialized observability to track query behavior, and gRPC depends on protocol-aware tooling. A unified governance framework is often necessary to manage these differences across paradigms.

7. Developer Experience and Tooling

Developer experience is a critical factor in the success of APIs, particularly in large-scale applications where multiple teams collaborate over long periods of time. An API that is difficult to understand, test, or evolve can slow development, increase error rates, and raise maintenance costs. Tooling, documentation, and community support strongly influence how efficiently developers can design, consume, and operate APIs. As a result, API paradigms must be evaluated not only on technical performance but also on how they support developer productivity at scale.

REST offers a highly accessible developer experience due to its simplicity and long-standing adoption. Its reliance on standard HTTP methods and human-readable data formats makes it easy to learn and debug. A mature ecosystem of tools exists for REST, including API gateways, documentation generators, testing frameworks, and monitoring solutions. This maturity reduces onboarding time for new developers and allows teams to leverage well-established best practices when building and maintaining APIs. GraphQL enhances developer productivity through its schema-driven approach and strong introspection capabilities. The schema serves as a single source of truth, enabling automatic documentation and client-side tooling such as query validation and auto-completion. Front-end developers, in particular, benefit from the ability to iterate quickly without requiring frequent backend changes. However, the increased flexibility of GraphQL also demands a deeper understanding of schema design and resolver behavior, which can increase the learning curve for backend developers.

gRPC provides a different developer experience centered on strong typing and automated code generation. Service definitions are written once and used to generate client and server stubs in multiple programming languages, ensuring consistency across teams. This approach reduces runtime errors and improves maintainability in large systems. At the same time, the requirement to understand interface definition languages, binary protocols, and specialized tooling can make gRPC less approachable for teams unfamiliar with RPC-based systems.

Table 4 Developer Experience Comparison

Aspect	**REST**	**GraphQL**	**gRPC**
Learning Curve	Low	Medium	High
Tooling Maturity	Very High	High	Medium
Type Safety	Low	Medium	High
Client Generation	Manual	Automatic	Automatic

Debugging and testing practices vary significantly across API paradigms. REST APIs are easy to inspect using standard tools due to their text-based nature. GraphQL debugging often requires specialized tools to analyze query execution paths and resolver performance. gRPC debugging can be more complex because of binary payloads, necessitating protocol-aware logging and monitoring solutions. These differences influence how quickly developers can identify and resolve issues in production environments. From a collaboration perspective, API design directly affects how teams coordinate and evolve systems. REST supports decentralized ownership through loosely coupled endpoints, while GraphQL encourages centralized schema governance that requires cross-team coordination. gRPC promotes clear service contracts, which can improve collaboration but also require strict change management processes. Each paradigm shapes team workflows in distinct ways.

8. Use-Case–Driven Evaluation

The suitability of an API paradigm is highly dependent on the specific use case in which it is applied. Large-scale applications rarely operate under a single set of requirements; instead, they support diverse clients, workloads, and operational constraints. Evaluating REST, GraphQL, and gRPC through a use-case–driven lens provides practical insights into how these paradigms perform in real-world scenarios. Such an evaluation helps organizations align API choices with business goals and system requirements. Public-facing and external APIs often prioritize simplicity, interoperability, and broad accessibility. In these scenarios, REST remains a preferred choice due to its reliance on standard web technologies and its compatibility with a wide range of clients and platforms. Its human-readable data formats and well-understood semantics make it suitable for third-party developers and open ecosystems. Additionally, REST integrates seamlessly with web infrastructure such as caches and content delivery networks, which is advantageous for high-traffic external services.

Data-intensive client applications, such as single-page web applications and mobile apps, present different requirements. These clients often need to aggregate data from multiple sources and adapt to rapidly changing user interfaces. GraphQL is particularly well-suited for such use cases because it allows clients to request precisely the data they need in a single interaction. This reduces network overhead and improves responsiveness, especially in environments with limited bandwidth or high latency. Internal service-to-service communication within microservices architectures places a strong emphasis on performance, reliability, and efficiency. In these environments, gRPC is frequently favored due to its low-latency communication, compact message formats, and strong typing. The ability to define clear service contracts and leverage automated code generation simplifies coordination among development teams. These characteristics make gRPC a natural fit for backend systems that require high throughput and strict performance guarantees.

Many large-scale systems adopt hybrid or polyglot API architectures that combine multiple paradigms. For example, a system may expose REST or GraphQL APIs to external clients while using gRPC for internal communication between microservices. This approach allows organizations to leverage the strengths of each paradigm while mitigating their weaknesses. However, it also introduces additional complexity in terms of governance, monitoring, and operational consistency. Organizational factors, such as team expertise, development workflows, and existing infrastructure, also influence use-case–driven API selection. Teams with strong web development backgrounds may gravitate toward REST or GraphQL, while those with experience in distributed systems may prefer gRPC. Tooling availability, operational maturity, and long-term maintenance considerations further shape these decisions. As a result, technical suitability must be evaluated alongside organizational readiness.

9. Conclusion

This comparative design study has examined REST, GraphQL, and gRPC as three prominent API paradigms used in large-scale applications, highlighting their architectural principles, communication models, performance characteristics, security considerations, and developer experience. The analysis demonstrates that each paradigm addresses different system needs: REST provides simplicity and broad interoperability, GraphQL offers flexible and efficient data access for diverse clients, and gRPC delivers high-performance, strongly typed communication for internal services. Rather than identifying a single superior approach, the study emphasizes that API effectiveness is highly context-dependent. Large-scale systems benefit most when API design choices are aligned with workload characteristics, organizational capabilities, and long-term maintenance strategies. As a result, hybrid and polyglot API architectures have emerged as a practical and increasingly common solution.

Future Directions

Future developments in API design are likely to focus on improving automation, adaptability, and governance across heterogeneous systems. Emerging trends include smarter API gateways capable of dynamically routing and transforming requests between REST, GraphQL, and gRPC, as well as enhanced observability tools that provide unified monitoring across paradigms. Research opportunities also exist in automated schema and contract evolution, adaptive query optimization, and AI-assisted API design and security analysis. As large-scale applications continue to grow in complexity, future API ecosystems will increasingly prioritize flexibility, performance, and resilience while minimizing operational overhead, reinforcing the need for continued comparative research and innovation in API technologies.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Fielding, R.T. (2000) Architectural Styles and the Design of Network-Based Software Architectures. University of California.

[2] Fielding, Roy T. and Richard N. Taylor. "Principled design of the modern Web architecture.", ACM Transactions on Internet Technology, Vol. 2, No. 2, May 2002, Pages 115–150.

[3] C. Pautasso, O. Zimmermann, F. Leymann, RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision,Proc. of the 17th International World Wide Web Conference (WWW2008), Bejing, China, April 2008.

[4] Richardson, L., & Ruby, S. (2007). RESTful Web Services. pages. 446, O'Reilly Media.

[5] G. Brito, T. Mombach and M. T. Valente, "Migrating to GraphQL: A Practical Assessment," 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019, pp. 140-150, doi: 10.1109/SANER.2019.8667986.

[6] Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference (WWW '18). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1155–1164. https://doi.org/10.1145/3178876.3186014

[7] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (February 1984), 39–59. https://doi.org/10.1145/2080.357392

[8] Google. (2016). gRPC: A High Performance, Open-Source Universal RPC Framework.

[9] Erik Wittern, Alan Cha, and Jim A. Laredo. 2018. Generating GraphQL-Wrappers for REST(-like) APIs. In Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings. Springer-Verlag, Berlin, Heidelberg, 65–83. https://doi.org/10.1007/978-3-319-91662-0_5

[10] Lawi, A.; Panggabean, B.L.E.; Yoshida, T. Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. Computers 2021, 10, 138. https://doi.org/10.3390/computers10110138

[11] Espinha, T., Zaidman, A., Gross, H.-G., 2014b. Web API growing pains: stories from client developers and their code. In: Proceedings of the Conference Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), IEEE CS, pp. 84-93.

[12] Steve Vinoski. 2005. RPC under fire. IEEE Internet Computing 9, 5 (2005), 93--95.

[13] Zimmermann, O.: Microservices Tenets: Agile Approach to Service Development and Deployment. In: Computer Science — Research and Development, Springer 2016. https://www.ost.ch/fileadmin/dateiliste/3_forschung_dienstleistung/institute/ifs/cloud-application-lab/msa-pospaperzio4summersoc2016v15nc.pdf

[14] Jamshidi, Pooyan et al. "Microservices: The Journey So Far and Challenges Ahead." IEEE Softw. 35 (2018): 24-35.

[15] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. ACM Comput. Surv. 37, 4 (December 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[16] Brito, Gleison and Marco Túlio Valente. "REST vs GraphQL: A Controlled Experiment." 2020 IEEE International Conference on Software Architecture (ICSA) (2020): 81-91.

[17] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

[18] C. Pautasso, E. Wilde, RESTful web services: principles, patterns, emerging technologies, in: Proceedings of the 19th International Conference on World Wide Web, 2010, pp. 1359–1360.