OFFICE OF THE CTO

The Role and Impact of GraphQL

By Rajesh Narayanan, Applications and Security

Reviewed by and Contributions from: OCTO Team and others.

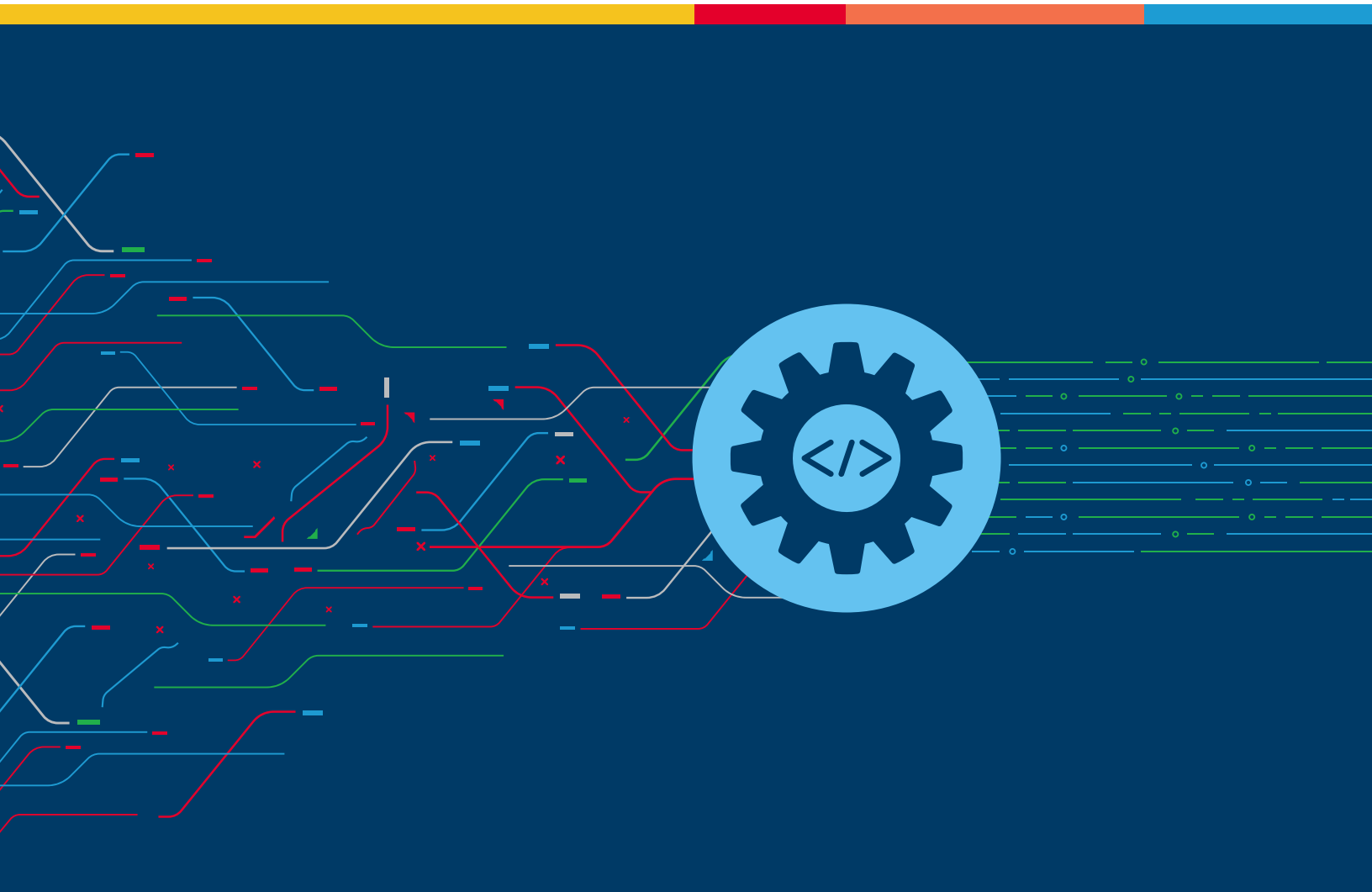

Table of Contents

F5 Office of the CTO Opinion

GraphQL has emerged as a modern and efficient approach to API development, surpassing the limitations of traditional REST APIs. While REST has been widely used since the early days of the web, GraphQL offers a fresh perspective and greater control for developers. With GraphQL, developers can define a strongly-typed schema that empowers clients to precisely request the data they need. By eliminating over-fetching and under-fetching of data, GraphQL optimizes performance and facilitates the creation of modern, interactive web applications which have complex data querying and data modeling requirements.

However, we acknowledge that GraphQL is not without its challenges. This paper provides insights into the security concerns and learning curve associated with GraphQL adoption. We explore real-world case studies describing the benefits netted by renowned companies that have successfully implemented GraphQL.

Furthermore, we present our own investigation into GraphQL, sharing our experiences and discoveries. We outline a short primer on GraphQL, compare it with REST, and delve into the challenges it addresses. Security considerations are also discussed to help organizations make informed decisions. Our research reveals the transformative potential of GraphQL. We showcase how we simplified a test management suite software architecture, resulting in a growth of over 300% in the amount of exposed data hidden within JSON objects through GraphQL.

In conclusion, we strongly recommend that enterprises grappling with scaling, optimizing, or operating a REST API infrastructure consider GraphQL as a viable solution. Our insights offer practical guidance to those embarking on the GraphQL journey, enabling them to harness its benefits and overcome challenges effectively.

GraphQL Primer

WHY GRAPHQL?

Limitations with REST APIs are driving demand for a new API technology approach.

The REST (Representational State Transfer)-based approach was originally proposed as a set of architectural principles for designing web APIs. REST evolved throughout the 2000s and 2010s with the emergence of Web 2.0 as a better means to implement service-oriented architecture (SOA) over other technologies like Common Object Request Broker Architecture (CORBA).

As the number of clients grew due to mobile adoption, the demand for precise data increased. However, REST-based APIs often resulted in over- or under-fetching of data, leading to inefficiencies. This approach, exemplified by apps like Facebook, often required numerous REST API calls for a single update, increasing network traffic and compromising performance and user experience.

GraphQL was specifically designed to address these limitations by offering a strongly-typed schema and a more efficient way to query data. This makes it well-suited for use cases where specific data is required to optimize network bandwidth. Additionally, GraphQL's ability to introspect the schema enables better documentation and tooling. While a more standardized implementation of REST could have provided some competition, GraphQL's unique features and benefits make it a compelling choice for modern application architectures that are distributed and data-intensive.

Figure 1 shows a topological difference between how REST and GraphQL are implemented. As stated in REST API vs GraphQL[1], "the key difference between GraphQL and REST APIs is that GraphQL is a query language, while REST is an architectural concept for network-based software."

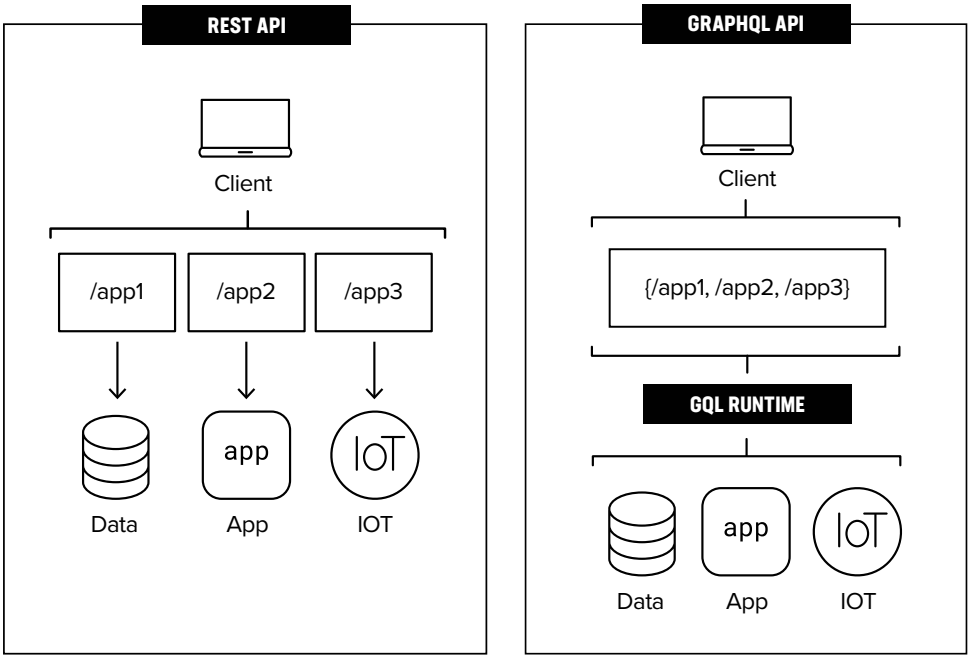

Figure 1: REST vs. GraphQL. Adapted from Rest API vs. GraphQL[1]

META'S MOTIVATION FOR GRAPHQL

Meta created GraphQL in 2012 (*open-sourced* 2015) to improve the performance and flexibility of their mobile apps. Prior to GraphQL, Meta's mobile apps were built using a combination of RESTful APIs and native code, making it difficult to handle the wide range of devices, screen sizes, and network conditions the apps needed to support.

One of the main challenges they faced was that RESTful APIs would often return the wrong amount of data—sometimes too much and sometimes too little. When the API returned a large amount of data the mobile apps didn't need it led to slow loading times and poor performance. When the API returned too little data, mobile apps needed to make multiple requests to different endpoints to fetch all the data they required, adding latency and complexity to the process.

Meta developed GraphQL so any app could request only the data it needed in a single request. This made it possible to optimize data transfer between the mobile apps and the backend services, leading to faster load times and better performance. Moreover, GraphQL's strong typing and self-documenting features made it easier for developers to understand and consume the API.

GRAPHQL ADVANTAGES

GraphQL offers powerful capabilities for data retrieval and manipulation, providing significant benefits over traditional API approaches.

Strongly-Typed Schemas

GraphQL features a strongly-typed schema that ensures clarity and accuracy in defining the structure and types of data that can be queried from an API. Let's say we have an API for a library that contains books, authors, and publishers.

a) **GraphQL Schema**: In GraphQL a strongly typed schema would look like Figure 2 below:

```
type Book {
  id: ID!
  title: String!
  author: Author!
  publisher: Publisher!
}

type Author {
  id: ID!
  name: String!
  books: [Book!]!
}

type Publisher {
  id: ID!
  name: String!
  books: [Book!]!
}
```

Figure 2: GraphQL schema example

```
type Query {
  books: [Book!]!
  book(id: ID!): Book
  authors: [Author!]!
  author(id: ID!): Author
  publishers: [Publisher!]!
  publisher(id: ID!): Publisher
}
```

Figure 2: GraphQL schema example (cont.)

Strongly-typed schemas in GraphQL offer security-related benefits by enabling input validation, preventing over-fetching and under-fetching of data, providing clear documentation-and-tooling support, facilitating version control, and aiding in authorization and access control. These features enhance API security by reducing the risk of common vulnerabilities and ensuring proper data handling and access management.

In the example shown (Figure 2), the schema defines the data types for books, authors, and publishers, and their relationships to one another. The schema is *strongly-typed*, which means every field has a specific data type, and clients can easily introspect the schema to discover available fields and their types.

b) **REST Schema**: In REST the schema definition would be loosely-typed as shown in Figure 3 below:

```
GET /books
GET /books/{id}
GET /authors
GET /authors/{id}
GET /publishers
GET /publishers/{id}
```

Figure 3: REST schema example

These endpoints return JSON objects representing the books, authors, and publishers, but the schema itself is not explicitly defined and is left to the skill and interpretation of the programmer. Clients will have to rely on documentation to understand the structure of the data.

Beyond REST

GraphQL has evolved beyond being a better alternative to REST and has the potential to be a preferred approach for enterprises considering a better data strategy. In addition to solving the limitations of REST, there are several other reasons why GraphQL has evolved into a new approach to API design. While the table (Figure 4) can show advantages of GraphQL over REST, GraphQL is best thought of as a response to the evolution of the internet and different applications rather than a response to the identification of problems with REST itself.

ATTRIBUTES	GRAPHQL	REST
FLEXIBLE DATA MODELING	GraphQL allows developers to easily define and evolve APIs to match changing requirements. Clients can precisely specify the data they need using the query language, allowing for a more flexible and efficient data retrieval process.	The server typically defines fixed endpoints that return predefined data structures. Clients have limited control over the shape and structure of the response, often resulting in over-fetching or under-fetching of data. REST lacks the fine-grained control over data retrieval and composition that GraphQL offers.
BATCHED QUERIES	GraphQL allows multiple queries to be combined into a single request, which can significantly reduce the number of round trips between the client and server and improve performance.	Rest has no built-in mechanism for batching multiple queries into a single request. Each REST request typically corresponds to a single resource or endpoint. Some REST frameworks or extensions may provide ways to bundle multiple requests together but are not native or standardized as in GraphQL.
TYPED QUERIES AND RESPONSES	Clients can specify the exact data they need, and servers can respond with only the requested data, reducing over-fetching and under-fetching. Additionally, GraphQL is strongly typed, which helps prevent errors and improves tooling support.	Typing is not inherently enforced in queries and responses. The structure and format of the data are typically predefined by the server, and clients must interpret and handle the data accordingly. This can lead to less type-safety.
INTEGRATION WITH FRONT-END FRAMEWORKS	GraphQL is designed to work well with front-end frameworks like React and Vue, making it easier to build modern, interactive web applications. GraphQL has dedicated libraries and tooling for seamless integration.	While REST can be used with front-end frameworks, the integration may require more manual effort and custom implementations. While there are third-party libraries, REST does not provide a standardized way of integrating with specific front-end frameworks like React or Vue.
GRAPH BASED QUERYING	GraphQL allows for complex queries that span multiple resources and relationships in a single request, making it easier to get the data needed to build complex user interfaces.	REST typically follows a resource-centric approach, where each endpoint represents a specific resource or entity. It does not inherently provide a mechanism for querying data across multiple resources or relationships in a single request.
PERFORMANCE	GraphQL's ability to precisely request only the required data can lead to more efficient data retrieval and improved performance.	REST APIs may suffer from over-fetching or under-fetching of data, as clients have limited control over the response structure. This can impact performance, particularly if the API returns excessive or unnecessary data.
DEVELOPER PRODUCTIVITY	GraphQL's self-documenting nature, with introspection capabilities, reduces the need for extensive documentation and fosters better understanding of the data model. Strongly-typed schema and query validation promote shared understanding and catch errors early. GraphQL has an intuitive query language and easier learning curve, facilitating better onboarding and knowledge transfer within dev teams.	Due to the lack of a standardized contract between the client and server, tribal knowledge suffers. REST APIs typically rely on informal documentation or conventions varying across different implementations. The lack of a shared understanding leads to inconsistencies and knowledge gaps within teams. This puts the onus on developers who have been on the team longer to spend precious cycles educating team members rather than focusing on the business problem. This reliance on individuals for documentation results in fragmented tribal knowledge, making it difficult for all team members to have a comprehensive understanding of the API's capabilities and data structures.

API VERSIONING	GraphQL's inherent advantage when it comes to versioning is in the ability to deprecate fields that are going to be removed, giving the developers on the client side time to adjust. Versioning similar to REST APIs is still possible.	Deprecating fields is something not inherently available in REST. It is left up to the client-side developers to ensure they are using the correct version of the API.

Figure 4: GraphQL advantages over REST

CHALLENGES WITH GRAPHQL

While GraphQL provides many advantages over traditional RESTful APIs, there are also some challenges to using it. Common challenges include:

1. **Learning curve:** Because most developers are more familiar with REST, organizations that plan to adopt GraphQL will need to budget time for their teams to learn how to use it effectively. GraphQL requires a different way of thinking about building and consuming APIs and adopting it could require changes to the underlying application architecture. Developers may need to learn new concepts such as schemas, resolvers, and types, as well as new tools and libraries. With GraphQL, clients have more control over the data they can access, which can make it more difficult to secure APIs. Techniques such as input validation, authentication, and authorization may need to be applied differently than with RESTful APIs.
2. **Caching:** Caching can be more complex with GraphQL, as clients can request different data with each query, making it harder to cache and reuse responses.
3. **Performance:** While GraphQL allows clients to request only the data they need, it can also enable more complex and resource-intensive queries. GraphQL APIs can have performance issues, particularly when querying large datasets or when a high number of concurrent requests are made. Developers need to implement strategies like limiting the query depth or ensuring clients are authorized to access only the specific data needed.
4. **Error Handling:** GraphQL can make error handling more complex as errors may be returned as part of the response rather than as a separate HTTP status code.
5. **Testing:** Testing with GraphQL presents challenges due to the complexity of queries, lack of standardized testing approaches, schema evolution, and query validation during runtime. Developers need to invest time in finding suitable testing frameworks and tools to address these challenges. Developers need to ensure comprehensive test coverage by selecting appropriate tools and considering schema evolution to effectively test GraphQL APIs and ensure their functionality and stability.

6. **Monitoring:** Monitoring GraphQL APIs can be challenging due to the complexity of queries, lack of standardized logging and metrics, and the potential for performance issues. The dynamic nature of GraphQL queries makes it harder to predict and monitor the structure and size of requested data. The absence of standardized monitoring tools specific to GraphQL APIs makes it difficult to gain insights into GraphQL query performance, error tracking, and API health. Developers need to adopt specialized monitoring tools and practices that can handle the unique characteristics of GraphQL, ensuring efficient performance and reliable operation.

GraphQL Usage Patterns

GraphQL is a powerful tool that can be used in a wide range of applications, from building APIs to powering mobile apps. Its flexibility and ability to unify data sources makes it well-suited for a variety of different usage patterns.

1. **Building efficient APIs:** One of the primary uses of GraphQL is to build APIs that can be consumed by web and mobile applications. GraphQL provides a flexible and powerful way to define and access data, making it well-suited for building APIs that need to support a wide range of clients and use cases.
2. **Data fetching and manipulation:** GraphQL can be used to fetch and manipulate data from a variety of sources, including databases, cloud services, and other APIs. By providing a unified way to access data, GraphQL can help to simplify the process of building and maintaining data-driven applications.
3. **Real-time Use Cases:** GraphQL is well-suited for real-time use cases, as clients can subscribe to updates to specific data and receive notifications when the data changes. This can be used in applications like chat, live streaming, real-time dashboards, and more.
4. **Microservices:** GraphQL can be used to build a flexible, loosely coupled architecture that allows different microservices to communicate with one another in a consistent and well-defined way.
5. **Back end to Front end:** GraphQL can be used to build a back-end-for-front-end (BFF) architecture, which allows different front-end clients to access data and services in a consistent and efficient way.
6. **Mobile Development:** GraphQL can be used to power mobile apps, by providing a way to access data and services in a consistent and efficient way, regardless of the platform or device.

Industry GraphQL Adoption Experience (Case Studies)

There are obvious motivations for why we need GraphQL, as seen in the following industry deployments:

NETFLIX

Netflix summarized their GraphQL journey in 2020 with a two-part blog series that is a must read for any GraphQL practitioner.[2] The case study they presented was the Studio Edge system taking their studio APIs and rearchitecting for GraphQL. Netflix's Studio API is used by their content teams to manage and monitor the production, post-production, and distribution of TV shows and movies. The API provides access to a vast amount of data, including metadata about titles, talent information, and more.

Initially, the Studio API was built using a traditional RESTful architecture, with endpoints that returned data in JSON format. However, as the API grew and the number of clients accessing it increased, it became clear a more efficient solution was needed.

Netflix began exploring GraphQL as a potential solution for the Studio API. They started by building a curated graph for the Studio API. They identified several key benefits, including the ability to reduce network usage, simplify data access, and improve performance by allowing clients to request only the data they need.

Netflix also faced some unique challenges when adopting GraphQL for the Studio API, particularly around managing the complexity of the schema, and ensuring clients could only access the data they were authorized to see.

To address these challenges, Netflix developed a custom solution called "Netflix GraphQL Federation," which uses the Apollo Federation specification to split the Studio API schema into smaller, more manageable pieces. They also implemented a system for managing permissions and access control, which allows them to restrict client access to certain parts of the schema.

In the blog they also published the evolution of their API architecture, from a Monolith to Federated Gateway. Figure 5 is an adaptation of the picture from their blog.

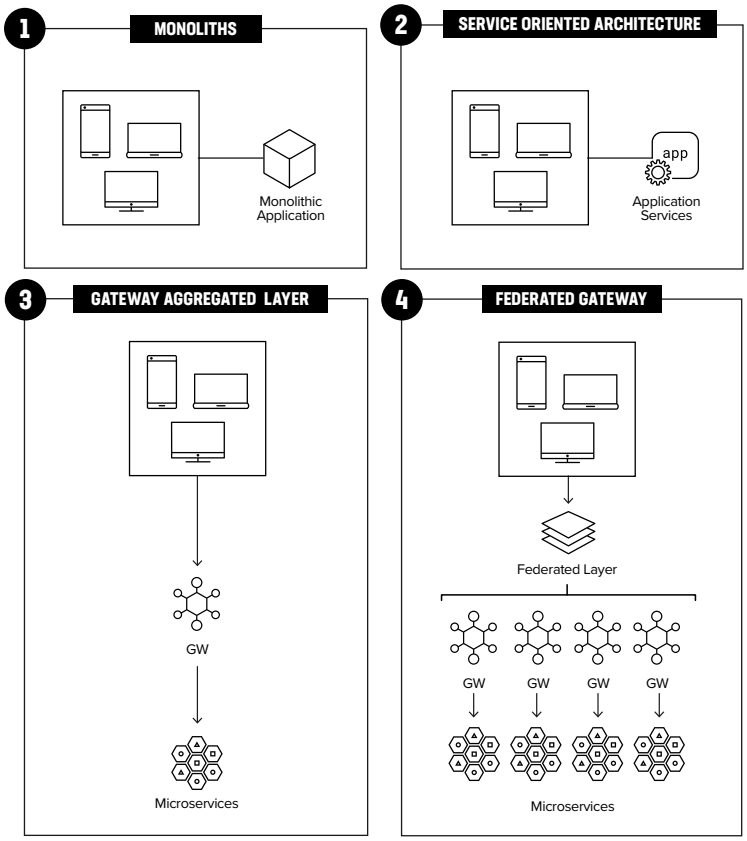

Figure 5: Evolution of API architecture (Adapted from Netflix)[2]

We believe there is another approach missing which may be prevalent in the industry but not formalized.

One could combine the gateway aggregated layer and federated gateway from Figure 5 as shown below:

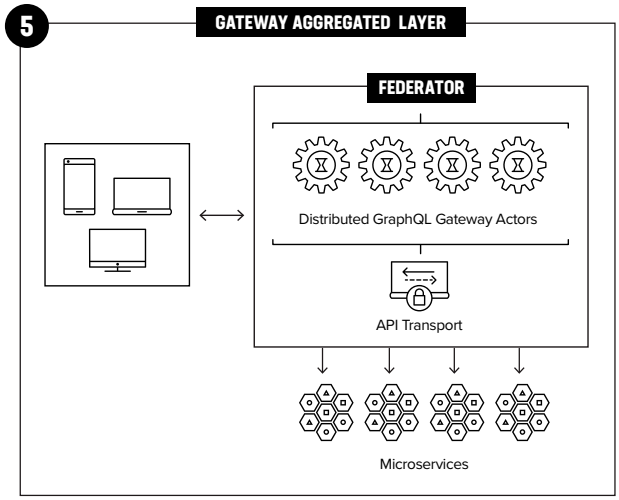

Figure 6: F5's evolved API architecture including ephemeral GraphQL gateway actors

Each gear icon essentially represents an ephemeral and federated GraphQL instance (a.k.a. gateway actor) that may be instantiated in real time (even on a per transaction basis) at the edge, i.e., topologically close to the clients and connect to the microservices over a secure API transport. This combination in essence replaces the gateway aggregation layer, or the federated layer, with distributed gateway actors which combine the best of both functionalities. We call this *distributed GraphQL gateway actors' architecture* as shown in Figure 7.

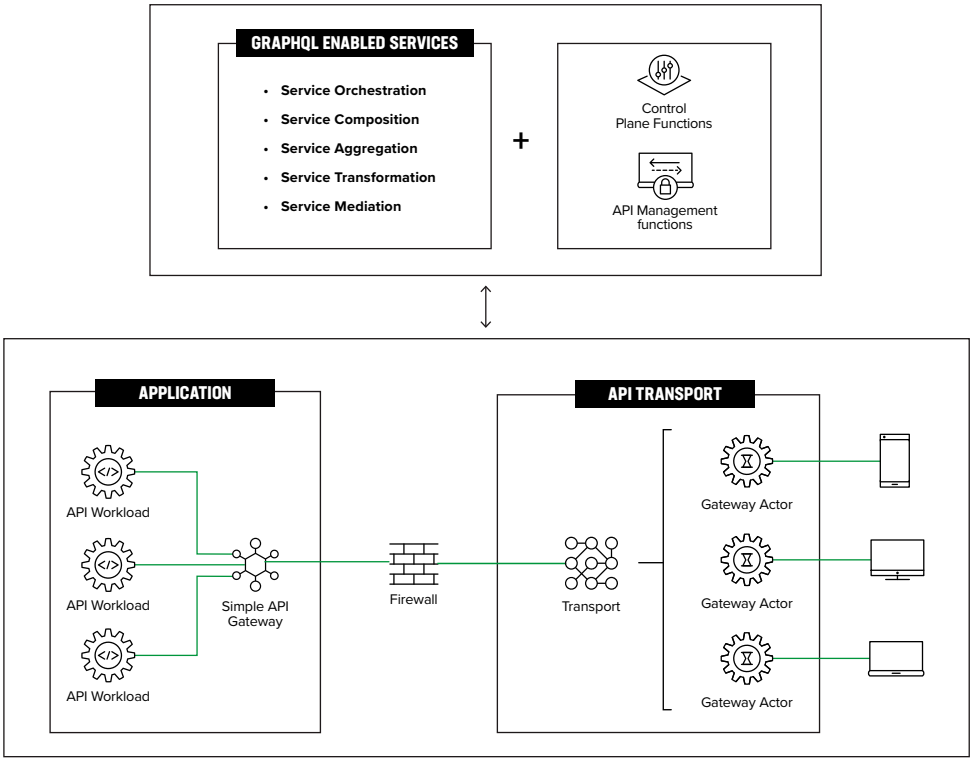

Figure 7: Distributed GraphQL gateway actors

OTHER EARLY ADOPTERS

PayPal

PayPal has been on the GraphQL journey since 2018 when they made it part of their Checkout app.[3] Their primary concern with REST, similar to Meta's, was the design principles are not optimized for web and mobile apps and their users. Client applications were making many round trips from the client to the server, taking approximately 700ms to fetch data. This resulted in slower rendering time, user frustration, and lower conversion rates. For the checkout app PayPal discovered that user interface (UI) developers were spending less than one third of their time building UI while the remaining time was spent figuring how to fetch and process data.

The other technologies the PayPal engineering team considered were orchestration APIs and Bulk REST. Building an orchestration API would lead to over fetching and clients being

coupled to the server. PayPal concluded that over time this approach can cause the API to become heavy, kludgy, and serve more than a single purpose. Their Bulk REST experiment also proved unsuccessful. While it freed the engineering team from having to tweak the orchestration APIs, it required their clients to have intimate knowledge of how the APIs worked.

PayPal's first experience using GraphQL for building a new product was a mobile SDK for integrating PayPal Checkout into apps. Within a week of evaluating GraphQL, the engineering team decided to use it for their new product. Despite the API not being ready, they were still able to complete the product ahead of schedule with almost no PayPal-specific knowledge necessary. Developers were able to quickly build an efficient and user-friendly app convincing the engineering team to fully adopt GraphQL in their technology stack.

Starbucks

Starbuck's employed a third-party to develop their Progressive Web App (PWA).[4]

The team was tasked with creating an ordering system that would efficiently and effectively accommodate complex business logic. As customers are able to personalize their orders, the development team had to ensure the system could accommodate multiple instances of unique business logic, sending the right data to the right place at the right time.

GraphQL made it possible for the team to make an efficient API with server-side caching and rendering to improve offline functionality. The team also used React to incorporate animation creating a dynamic and compelling user experience.

GraphQL at F5

The objective of the F5 Office of the CTO was to understand how we could use GraphQL within the F5 ecosystem. We have two projects underway exploring the use of GraphQL.

IMPROVING DATA MODELS AND VISIBILITY

The *test management suite* (TMS) offered by F5 provides customers with the ability to test endpoints or clients of their own systems and see if they are human or bots.

This was an internal facing project to help streamline the TMS software development and testing. The primary goal was to extract JSON data trapped in the existing SQL database, convert to graph data, and implement a GraphQL API for querying.

This was necessary because the current database poses challenges, including the "JSON-blob problem" which results from storing meta-data as JSON objects in a tabular database. Parsing and handling these objects is inherently expensive and inefficient. Moreover, the JSON-blobs, due to their graph nature, contain valuable data that can further improve F5's

products and security. By transitioning to a directed graph database, the JSON-blobs can be efficiently parsed and managed with directed relationships, optimizing data transfer and utilization.

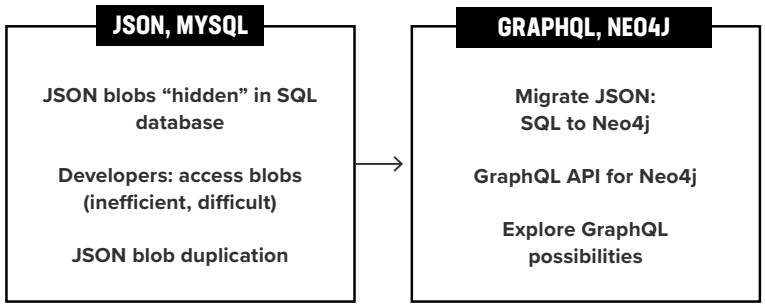

Figure 8: F5 TMS GraphQL Project Scope

The results are encouraging. By identifying which tables in the TMS relational database have JSON blobs, we determined that moving to a GraphDB and using GraphQL could increase our visibility into the system multi-fold.

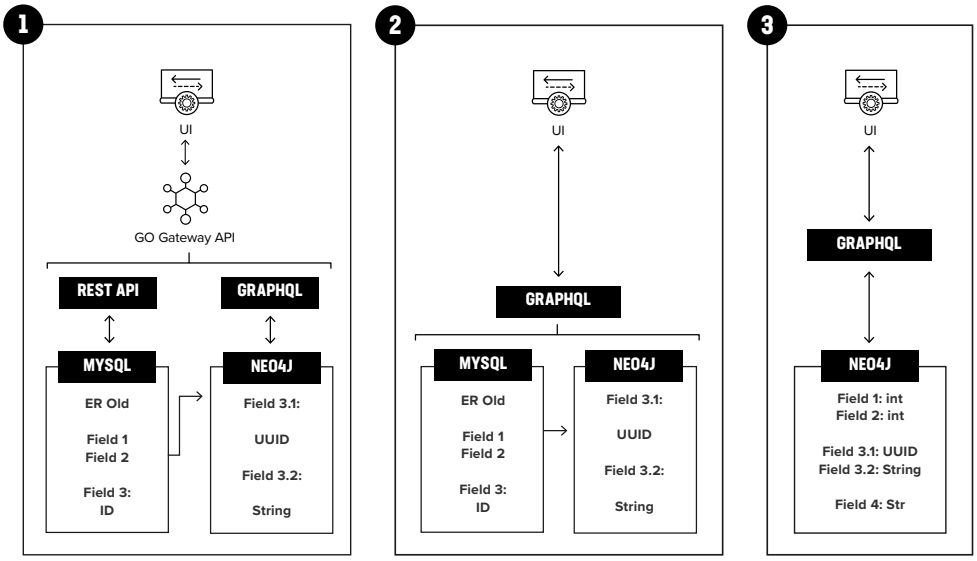

Figure 9: Implementation choices

Figure 9 shows the possible evolutions or implementation choices for this project. Each choice has its own implications.

Option 1 has the least impact on the UI as the client does not have to change. A hybrid mode, as in options 1 or 2, may work well depending on the situation, especially if there are a smaller number of columns with JSON objects. In addition to this, the derived schema for each JSON would also be small.

But when planning this we realized the JSON objects required context which was stored in other columns in the SQL database. The size of the schema stored in each JSON object was also quite large. This would have created more work maintaining the codebase. So, after careful consideration, we decided to rearchitect the application to support GraphQL and Neo4J as shown in option 3.

CLOUDGRAPH INTEGRATION

CloudGraph is a free open-source universal GraphQL API and Cloud Security Posture Management (CSPM) tool for AWS, Azure, GCP, and Kubernetes (K8s). The objective of the project is to use CloudGraph to build a 'CloudGraph plugin' (Figure 10) for the F5 Distributed Cloud (F5XC) data to appear on it, allowing for better visibility to cloud resources connected using F5 Distributed Cloud.

Our goal is to integrate with CloudGraph to gain a deep understanding of the technology and to build future CloudGraph integrations, insights, and GraphQL APIs for F5XC data.

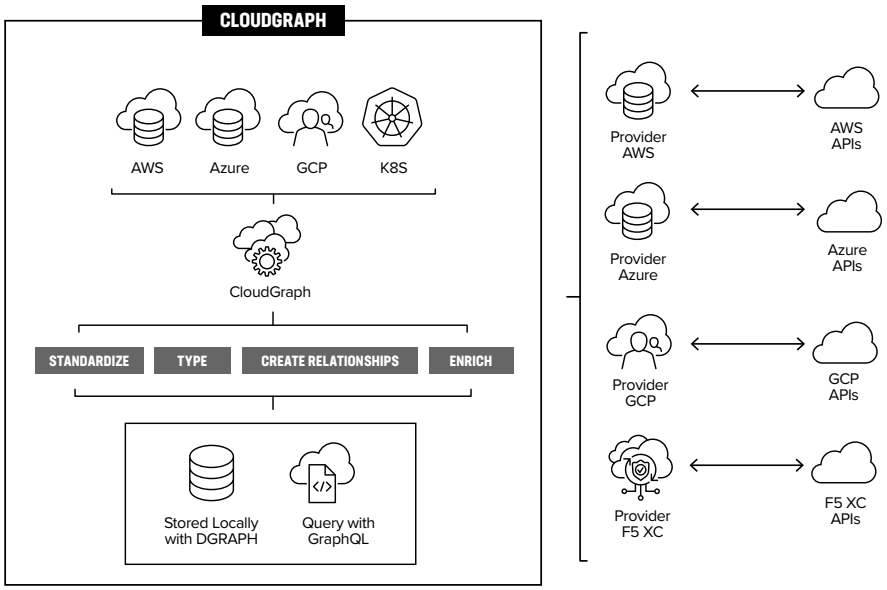

Figure 10: CloudGraph integration with F5 Distributed Cloud

After integrating the custom F5XC provider into the CloudGraph platform and using the create relationships capability of the platform we will have better visibility of cloud resources connected using F5XC and their relationships across other clouds. We can then generate complex queries on resources across multiple clouds for the same tenant. This will enable us to provide stakeholders with a comprehensive F5XC overview and explore a deeper integration of CloudGraph for additional GraphQL plugins and custom insights for cloud operations relevant to F5 and its customers.

Based on the above initiatives, our initial exploration has been in line with the experiences outlined by the case studies presented earlier.

GraphQL Security Concerns

Initial perceptions regarding GraphQL's superiority over REST in terms of security have been debunked, as GraphQL, like any other technology, carries its own risks if not implemented correctly. It is important to recognize GraphQL is not inherently more secure than other technologies. Proper implementation and adherence to best practices are crucial to ensure the security of GraphQL APIs. By dispelling this myth, we can approach GraphQL with a realistic understanding of its security considerations and take appropriate measures to mitigate any potential risks.

INTROSPECTION ATTACK

GraphQL provides developers with a powerful tool called introspection, which allows them to request information about the schema used by a GraphQL service. This tool enables developers to learn about the service and its structure. However, there are risks associated with introspection. Enabling introspection in a production GraphQL service means sensitive information within the schema can also be accessed. This poses a significant risk, as malicious users can exploit introspection to obtain sensitive data and potentially cause harm. Furthermore, introspection grants these users the ability to easily identify potentially harmful operations, as they can view the entire schema and determine which queries to execute. The consequences of such a leak can be disastrous, depending on the nature of the schema and the data it contains.

Mitigation: A solution to mitigate the risks of introspection is to disable it in production APIs using various frameworks. By disabling introspection, developers lose some of the convenient functionality it offers. However, to regain its usability, developers can register the GraphQL schema of the production API with tools like GraphOS. This allows them to retain controlled access to the schema and its information.

SQL INJECTION

SQL injection is a widely recognized and prevalent attack that poses risks ranging from data manipulation to complete database deletion. This straightforward attack takes advantage of string concatenation, allowing an attacker to insert executable code within input, granting unauthorized access to the database and enabling malicious alterations. Interestingly, this attack vector is not limited to SQL databases alone but can also affect graphical databases like Neo4j. Neo4j employs the Cypher query language, which resembles SQL and inherits its vulnerabilities. This type of attack, known as Cypher injection, exploits the similarities but also benefits from existing solutions without the need for new inventions.

Mitigation: Countermeasures include sanitizing user input to detect and prevent exploitation and using parameterization to abstract user input from direct query creation. These measures effectively mitigate the risk of injection attacks. It is crucial to address this security issue in GraphQL implementations, but fortunately, well-known and easily implementable solutions are available.

REST PROXY API ATTACK

Layering a GraphQL API on top of a REST API can lead to server-side request forgery (SSRF) vulnerabilities. Attackers can exploit this by modifying parameters sent to the REST API through the GraphQL proxy. If neither API validates parameters for specific calls, attackers gain control over the backend system. For example, appending "/delete" to a user ID in a GraphQL query can result in unintended deletion of data when passed to the REST API.

Mitigation: While this vulnerability is a valid concern, it can be effectively addressed by defining types in the GraphQL schema and thoroughly validating parameters before sending them to external APIs or services. It's important to note that this vulnerability, along with others associated with GraphQL, stems from implementation issues rather than being inherent problems with GraphQL itself. Such problems arise when GraphQL is misused due to a short-term mentality, despite the technology holding great promise.

BATCHING ATTACKS

Authentication vulnerabilities are a prevalent attack vector across various systems, including GraphQL. GraphQL's ability to send multiple queries or mutations in a single request exposes it to batching attacks, which involve brute force methods.

The first type of attack involves brute forcing login passwords. Attackers send a request with numerous mutations containing login credential pairs. Since GraphQL allows many mutations in one request, this attack bypasses rate limiting checks and enables the creation of a session by brute-forcing passwords.

The second type of attack operates similarly but targets a popular form of two-factor authentication (2FA) called one time password (OTP). A single request is sent with multiple mutations, each containing a valid login credential paired with a different OTP variation. If any of the mutations include the correct OTP, the request will be authenticated, and a session will be established.

Mitigation: Addressing these vulnerabilities requires a comprehensive approach, as there is no foolproof solution. Developers play a crucial role by adopting secure coding practices and emphasizing the perspective of business logic. On the web server side, it is essential to implement measures such as limiting login attempts and validating user input. Additionally, specific attention should be given to scanning requests for multiple mutations, as a legitimate login attempt typically does not involve more than one mutation.

DENIAL-OF-SERVICE ATTACK

These attacks involve overwhelming the GraphQL service with excessive traffic, rendering it incapable of handling requests from legitimate users and potentially causing server crashes. DoS attacks can be executed through recursive GraphQL queries or by sending large query requests.

In the recursive query scenario, attackers exploit the cyclical nature of GraphQL schema definitions. For instance, if the schema includes author and book types, an attacker can repeatedly query the author and book in a loop, overwhelming the server with recursive calls and disrupting its operation.

Alternatively, attackers can issue queries that request an extensive amount of data from the database. For example, a query may retrieve many authors and all their associated books. Such a massive data request can significantly slow down or crash the system.

Mitigation: There are several solutions to mitigate this problem. The first approach is to set a timeout on queries, preventing users from making requests that exceed a specified duration. However, this may not fully address the issue, as damage can still occur within the allotted time. Another solution is to enforce limits on query depth and complexity. Queries that surpass the designated depth or complexity, which deviates from the schema's definition, can be rejected. Lastly, implementing user throttling can be effective. If a user sends a large or consecutive set of requests, their access to the server can be temporarily denied until the server is prepared to handle additional requests.

GraphQL Deployment Patterns

There are several deployment patterns for GraphQL as shown, each with their own set of trade-offs and considerations.[5] Some of the most common patterns include:

MONOLITHIC DEPLOYMENT

This is the simplest and most common pattern for deploying a GraphQL API. In a monolithic deployment, the GraphQL server, the database, and any other necessary services are all deployed together as a single unit. This can make it easy to get started with GraphQL but can become difficult to scale and maintain as the application grows.

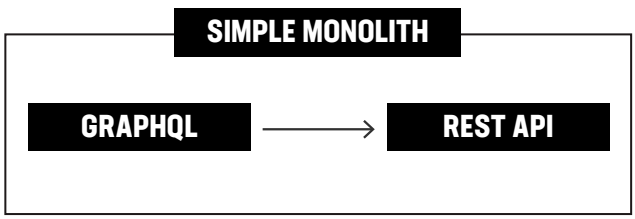

Figure 11: Simple Monolith

COMPOSED MONOLITHS

Composed monoliths refer to the architecture where a monolithic application is built using a combination of monolithic and microservices with GraphQL acting as the API layer (Figure 12). In this pattern, instead of breaking down the entire system into separate microservices, the application is initially developed as separate monoliths, which encapsulate all the business logic and data access layers.

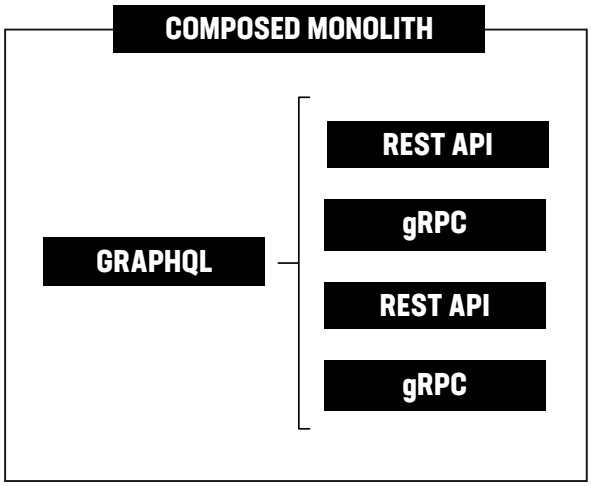

Figure 12: Composed Monolith

Within this monolithic structure, GraphQL is implemented as the API layer, allowing clients to request and retrieve data from different underlying microservices. This means while the application remains monolithic from a deployment perspective, it utilizes GraphQL as a means to compose data from various services and present it to clients in a flexible and efficient manner.

This pattern offers benefits such as simplified development and deployment, as well as the ability to leverage GraphQL's powerful querying and data-fetching capabilities. It allows developers to progressively adopt microservices architecture while still benefiting from the flexibility and ease of use provided by GraphQL.

FEDERATED SUB-GRAPHS

Federated graphs in GraphQL deployment patterns involve combining multiple GraphQL services, called subgraphs, into a single unified GraphQL schema. Each subgraph represents a separate domain or microservice with its own data and functionality. This architecture uses a central gateway that routes client queries to the appropriate subgraph based on the requested fields. By federating these subgraphs, developers can create a cohesive graph where clients can seamlessly query and traverse across different services.

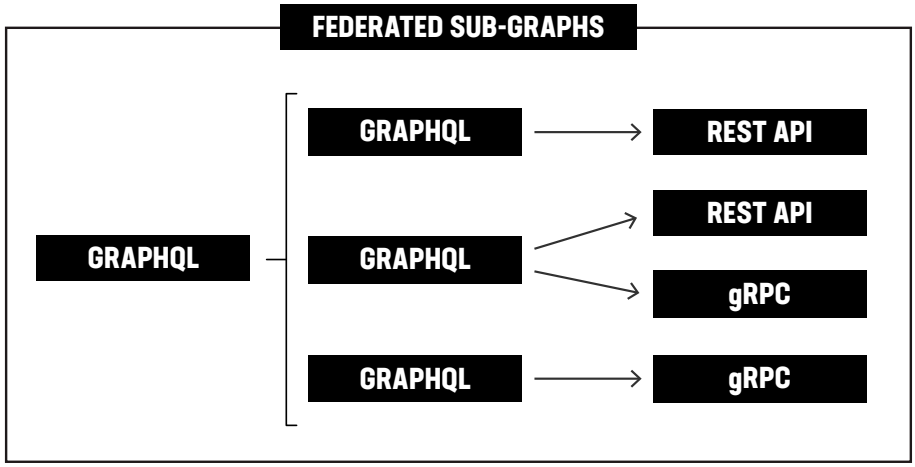

Figure 13: Federated sub-graphs

This approach promotes modularity, scalability, and independent development of services, resulting in improved performance and flexibility for building GraphQL APIs. Federated subgraphs (Figure 13) provide a powerful way to compose data from various services and create a distributed and scalable architecture.

HYBRID GRAPHS

Hybrid graphs (Figure 14) in GraphQL deployment patterns combine federated subgraphs and non-federated schemas through stitching techniques. This approach enables organizations to integrate existing GraphQL APIs or legacy systems with federated microservices, resulting in a unified GraphQL API that benefits from both approaches. By merging schemas and resolving relationships between types and fields, the hybrid graph architecture offers flexibility, modularity, and scalability.

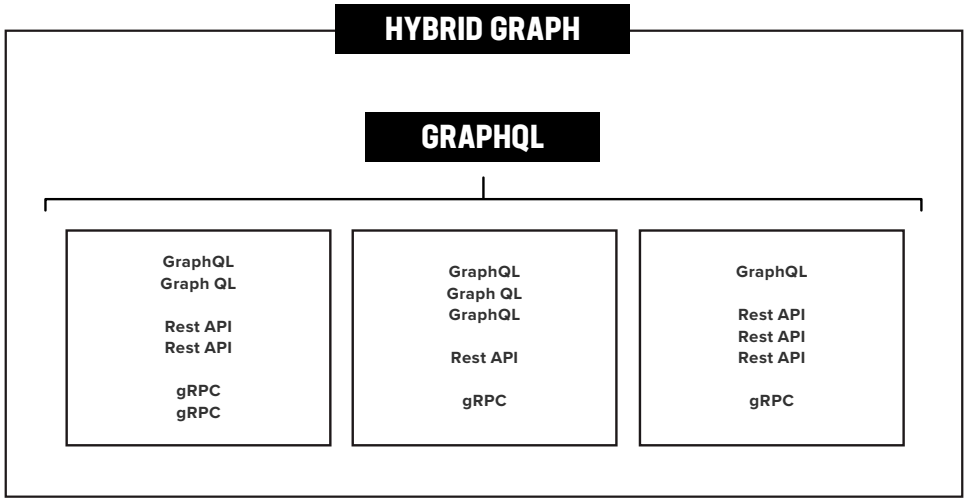

Figure 14: Hybrid graphs

The hybrid graph pattern provides organizations with the ability to gradually adopt federation while leveraging their existing resources. It allows for the seamless integration of federated

subgraphs and non-federated schemas, accommodating diverse requirements and promoting interoperability. This approach empowers organizations to build comprehensive GraphQL APIs that can scale and adapt to evolving needs. By combining the strengths of both federated and non-federated services, hybrid graphs offer a flexible and powerful solution for building GraphQL deployments.

Challenges with hybrid graphs in GraphQL deployments include managing the complexity of merged schemas, addressing potential performance overhead, ensuring data consistency and integrity, handling system evolution and versioning, and navigating the complexities of monitoring and debugging in a distributed architecture. These challenges require careful planning, optimization, and robust development practices to overcome and ensure the smooth operation of hybrid graph deployments.

SERVERLESS ROUTING

This pattern involves deploying the GraphQL API as a set of serverless functions, such as AWS Lambda or Google Cloud Functions. This can be a cost-effective and scalable way to deploy a GraphQL API but can also introduce additional latency and increase the complexity of the architecture.

There are many challenges with serverless routing. One issue is in seamlessly connecting distributed subgraphs as the application expands. Coordinating multiple teams and deployments can become complex, making it challenging to manage and scale the subgraphs effectively. Ensuring data consistency and synchronization between these subgraphs is another hurdle. Monitoring and debugging in a distributed serverless environment can be more difficult, requiring proper logging and error handling mechanisms. Additionally, managing access control and authorization across multiple serverless functions and subgraphs poses challenges. Addressing these challenges is essential for the smooth operation and scalability of a serverless-based GraphQL deployment.

EDGE-DEPLOYMENT

In an edge deployment pattern for GraphQL APIs, the API is placed closer to clients using a CDN service. This brings several benefits, including lower latency for faster responses, reduced load on the origin server, and protection against DDoS attacks.

By distributing the API infrastructure across edge servers worldwide, it can handle traffic more efficiently and provide a better user experience globally. The CDN's caching and filtering capabilities help improve scalability and ensure the API's availability, even in the face of heavy traffic or malicious attacks. Edge deployments have the potential to optimize an API's performance and reliability by leveraging CDNs' network of servers.

Conclusion

GraphQL technologies have matured enough to be embraced by traditional enterprise companies. Although Gartner's Hype Cycle for APIs 2022 indicates we are still several years away from widespread adoption, we have now reached a critical turning point where the necessary tools are already in place. While GraphQL is still relatively new, it has evolved to the point where it can meet the needs of established, large-scale enterprises.

The maturity of GraphAPI technologies is rooted in the growing number of enterprises embracing GraphQL and other graph database technologies. As more and more companies adopt these technologies, the tools and resources necessary to support them are becoming increasingly available making it easier for other enterprises to follow suit. Additionally, the benefits of GraphAPI technologies—such as improved data querying and reduced complexity—are becoming more widely recognized, further fueling their adoption among enterprise companies.

As an industry, we must recognize the importance of GraphAPI technologies such as GraphQL and their potential to overcome the limitations of REST. Enterprises must not overlook the urgent need to invest in and comprehend GraphQL, particularly in terms of data modeling, diversity, and opaqueness. While it's easy to become enamored with the features and potential of GraphQL, we must also acknowledge the potential trough of disillusionment after peaking at inflated expectations. Vendors must prioritize enhancing their products to fully support GraphQL, while enterprise IT—including vendors—should identify which data would benefit from GraphQL's exposure to improve productivity.

Customers must prioritize understanding their data and recognize an average enterprise operates differently than the hyperscalers that originally introduced GraphQL. Let us work together to embrace GraphQL as an effective software architecture pattern and drive the industry forward towards better productivity and innovation.

References
1. Rest API vs GraphQL, Bentil Shadrack, published November 28, 2022, https://dev.to/documatic/rest-api-vs-graphql-1a0n
2. How Netflix Scales its API with GraphQL Federation, published November 9, 2020, https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2
3.GraphQL, a Success Story for PayPal Checkout, Mark Stuart, published October 16, 2018, https://medium.com/paypal-tech/graphql-a-success-story-for-paypal-checkout-3482f724fb53
4.Case Study: Starbucks Progressive Web App, customer story retrieved from chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.mixedmediaventures.com/wp-content/uploads/2018/04/Starbucks.pdf on September 8, 2023
5. Announcing GraphQL for Gloo Gateway, Brian Gracely, September 15, 2022, https://www.solo.io/blog/announcing-graphql-for-gloo-gateway/

DC0623| REPORT-CORP-1110257067-distributed-gateway-actors-evolving-api-management