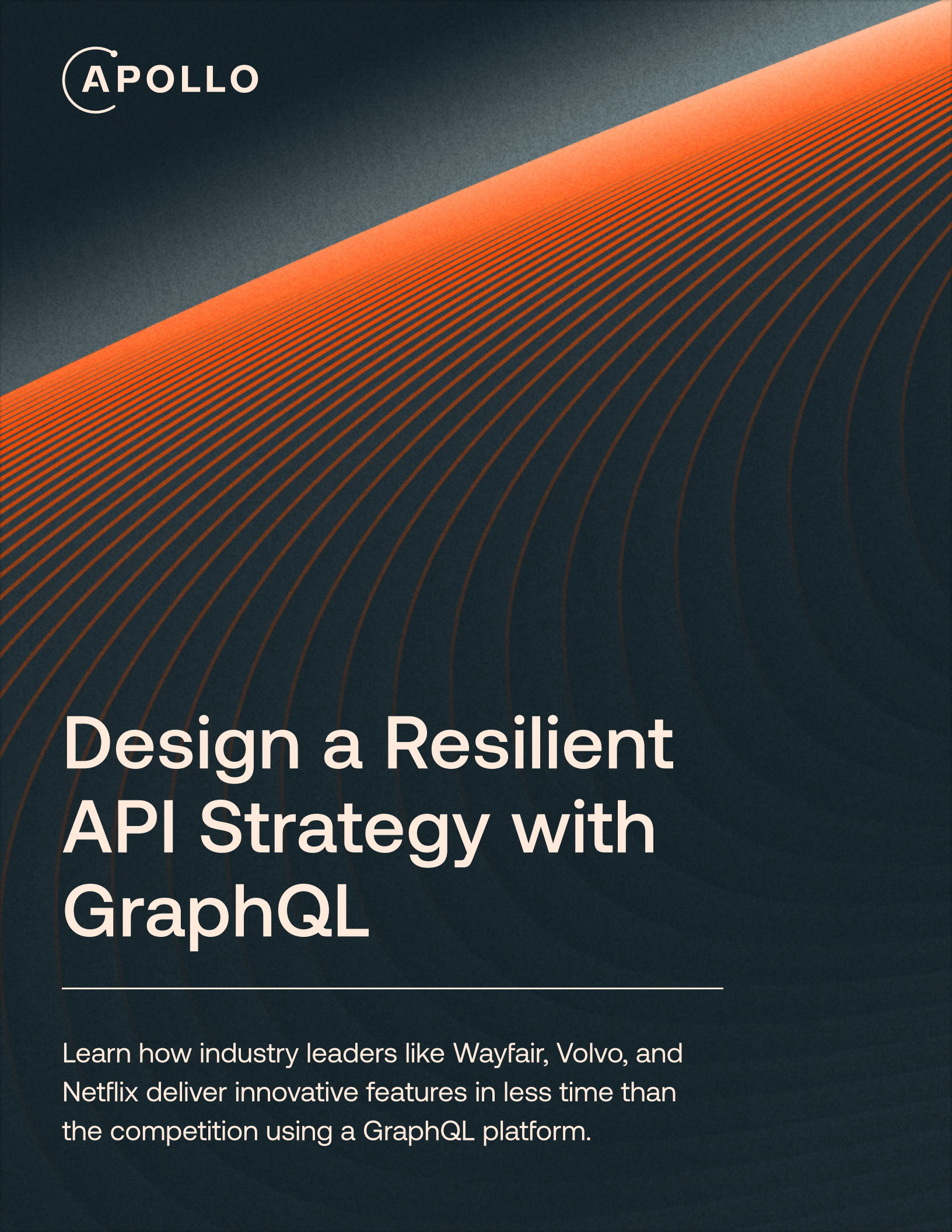


Design a Resilient API Strategy with GraphQL

Learn how industry leaders like Wayfair, Volvo, and Netflix deliver innovative features in less time than the competition using a GraphQL platform.

APOLLO

Table of contents

About the Author

Andrew I. Carlson is the Principal Field Architect at Apollo GraphQL. As a former Enterprise Architect and Group Director of Technology at VML, Andrew provided organizational leadership and architectural guidance for clients in the Fortune 50, driving successful modernization and digital transformation efforts spanning platform requirements, front-end modernization, and designing and migrating systems of record

Introduction

Resilient architectures grant us the freedom and flexibility to grow and change our systems with the needs of the business. This requirement presents challenges for Platform Architects at the API layer, who must also ensure stability at all times. Providing this API resiliency alongside stability isn't just a nice-to-have. In an increasingly digital world, it's a competitive advantage. API platform resiliency enables your organization to deliver better features and more offerings across any interface customers desire.

This white paper investigates the escalating complexity between the Presentation and Application tiers. It outlines how GraphQL enables technology leaders and platform teams to create a resilient strategy for their API platform. You'll learn:

01 What are the requirements and considerations for creating a **resilient API strategy** that helps teams confidently ship?

02 How can organizations **use GraphQL as a composable abstraction layer** that complements API technologies like RESTful APIs?

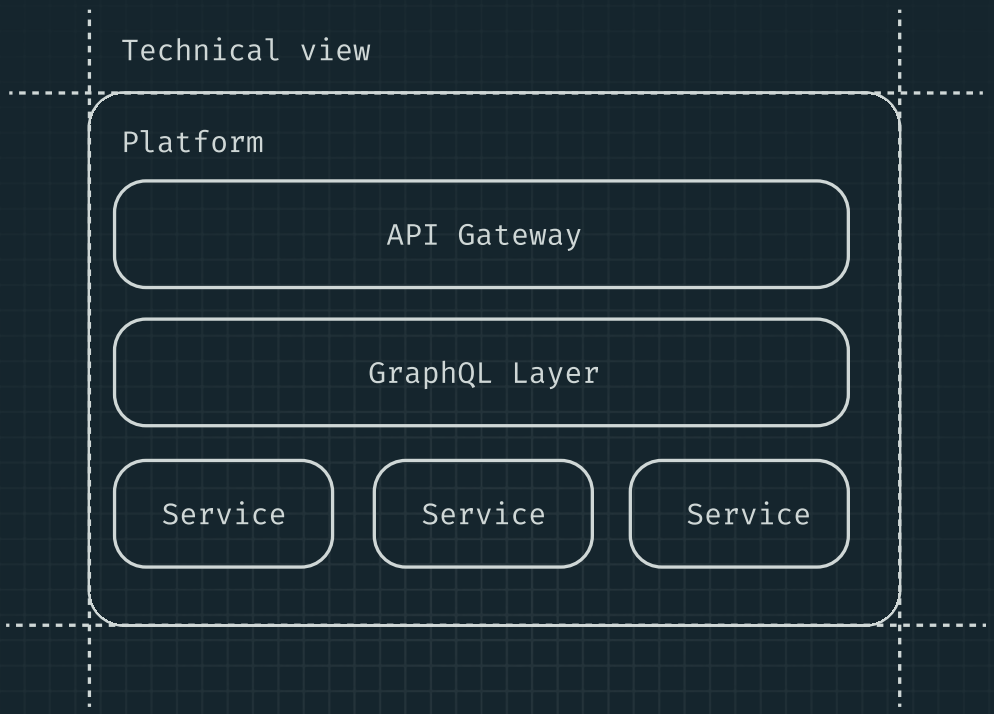
03 How do architects from Fortune 500 enterprises like **Wayfair, Volvo, and Netflix use GraphQL in their API strategy**?

Chapter 1

An Evolving Landscape

The internet is rife with debates about the best way to design, develop, and deliver APIs. When the discussion turns to GraphQL, it often pits GraphQL against REST or other architectural styles. GraphQL is flexible enough to serve different purposes. However, it is most effective when organizations use it as a complementary platform to the services and APIs they have invested in and continue to maintain rather than as a replacement.

At it's simplest, a platform could be viewed as:



Organizations are not replacing REST, although GraphQL usage is expanding. In many cases, they aren't even replacing SOAP. In the 2023 State of API[1] report published by Postman, 86% of respondents reported using REST, 26% reported using SOAP APIs, and 4% reported using EDI. This reliance on established technology is often an indicator of operational stability, not a sign that an enterprise is entrenched in old technology or is unwilling to change.

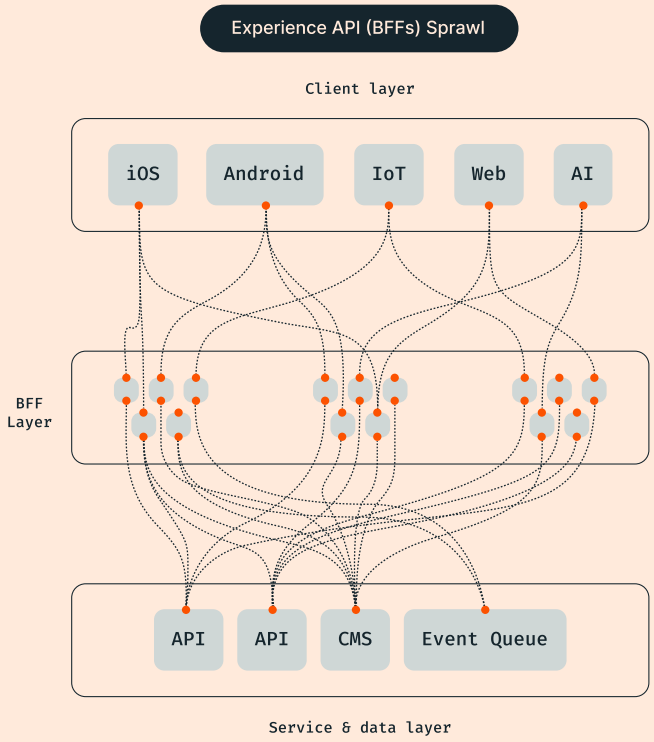
From customer information to product data and beyond, these APIs are valuable. One of the most demanding challenges architects face is building a system resilient enough to maintain stability while keeping an eye on the future. Replacing the APIs powering business-critical systems is efficient once something changes upstream. Still, we know new requirements, acquisitions, or presentation layers are coming.

GraphQL magnifies an organization's API investments when used as a complementary middle layer within an API platform. Using GraphQL to build an API composition layer increases the value of APIs, provides a unified governance boundary, and contributes a resilient strategy to an ever-evolving technical landscape.

So Many Endpoints, So Little Time

Our customers expect our products to work in an omni-channel world. Digital teams are expected to meet customers where they are, whether using a web browser, a mobile app, a voice assistant, or maybe even all three, with the need to transfer context between them. This presents challenges to enterprise architects, who must prepare for constant changes to the Presentation or Client tier A request for one landing page today can quickly turn into a request for ten by next week. Too many of us know the sinking feeling when someone drops by on a Friday afternoon and asks: "So how quickly could the team get a chatbot proof of concept put together?"

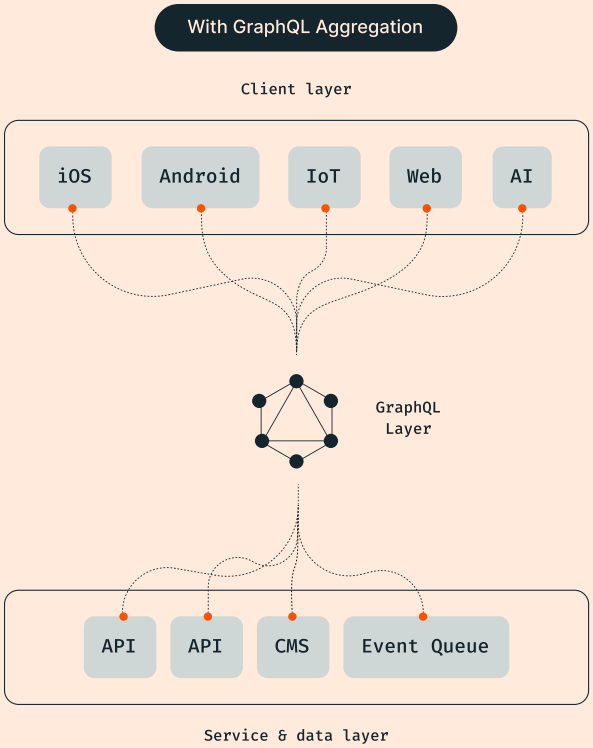
Layered architectures are a standard way of introducing the flexibility to rapidly respond to those one-off requests. One tenet of a well-layered architecture is that a given layer should be agnostic of its consumer[2]. For example, the Application tier shouldn't be aware of the Presentation. However, in practice, the Presentation and Application layers can inadvertently become tightly intertwined over years of developing new features, acquiring and merging with different companies, and digital transformations. This technology accretion sometimes leads to patterns like a proliferation of experience APIs or Backend-for-Frontends[3], resulting in dozens or hundreds of bespoke endpoints.



These endpoints don't exist in a vacuum, and they aren't a "build it and forget it" solution. They need the same care, tending, and maintenance that any other service requires. Security patches, version upgrades, and technologies are released daily with no sign of slowing down, with the Enterprise Software market expected to grow to $625 billion by 2030[4]. Even in organizations with strong SDLC practices, every service deployment introduces risks to the downstream consumers.

To best prepare for the unknown future of new requirements, technologies, and presentation clients, we need resilient API strategies that flex and evolve as our organizations grow while providing enforceable governance, security, and observability boundaries.

One way to achieve this is by inserting a loose-contract layer[5] after our API Gateway but before our application services. This loose-contract composition layer allows broad bidirectional expansion and evolution in the presentation and application tiers.



The high rate of change in the Presentation tier, paired with the system quality attributes that we're constantly improving in the Application tier, makes it the best place to inject resiliency into an existing architecture, whether it's a monolith or distributed system, and enforce layer agnosticism.

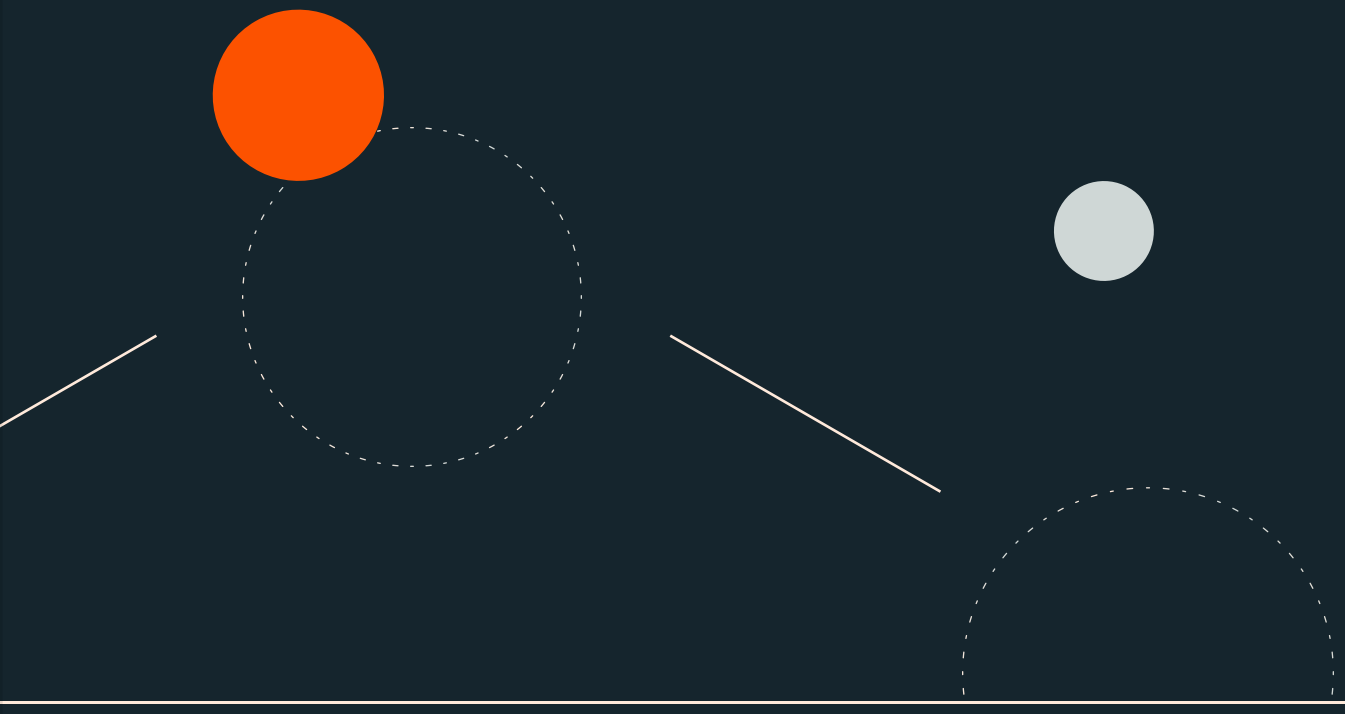
APOLLO

Chapter 2

Designing for Resiliency

C.S. Holling, the pioneer of resilience theory in ecology, identified two distinct types of resilience: Engineered and Ecological. Engineered Resilience focuses on stability near a state of equilibrium where successful resilience is measured by resistance to disturbance and the speed of return to equilibrium once broken. The second definition, Ecological Resilience, acknowledges that instabilities in a system can extend so far as to push a system into an entirely new state of equilibrium[6].

Like Holling's Ecological Resilience, truly resilient architectures are flexible and open to evolution. Designing with resiliency means acknowledging and accounting for change and building mechanisms for expansion and contraction – not waiting to ship until we've invented a perfect architecture.



We can't always design new systems, of course. More often than not, we have the more challenging task of maintaining an existing backend architecture while serving new clients or owning a legacy system and performing a progressive modernization on a three to ten-year time horizon. In application development, particularly new development, we talk about our work in sprints:

> "We can launch this new checkout flow in about one sprint." and, "It'll take about three sprints to ship that new homepage."

Managing an API platform, on the other hand, is a marathon – not a sprint – and for that marathon, we need resiliency. Our platforms must be strong enough to support the existing business, pliable enough to adapt to new technologies in both the Application or Presentation tiers without disrupting the other, and allow room for growth and expansion bi-directionally across tiers.

A resilient API strategy encompasses more than just the technical implementation. When considering the entire platform, it broadens to include solutions for the teams designing and developing the APIs, observability, governance, and the operations required to deploy and evolve the platform.

APOLLO

Resiliency Characteristics

When defining the characteristics of a resilient API strategy, we can approach the problem from the perspective of the pressure that we plan to put our platform under and then group them by similar attributes. For instance:

01 How might client-side teams be informed about changes to APIs?

02 How will the server-side teams collaborate with client-side teams?

03 How might we support and manage different types of APIs simultaneously?

04 How will we ensure we don't accidentally introduce breaking changes when a server-side dependency changes?

05 How might we add new presentation clients without deploying a new experience API?

06 How might we use our APIs more efficiently?

07 How might we fully understand the data in our system and who is using it?

08 How might we apply a principled approach to the services we deploy?

Taken collectively, we can group these attributes into four key characteristics that contribute to resiliency in our API platforms: **Rapid Self-Service, Insulating Layers, Magnification of Existing Investments, and Strong Governance.**

Rapid Self-Service

We build APIs for one purpose: to be used. When asked how they define the success of their platform,

58% of respondents to the 2023 State of APIs report[7] said that they measured success by **usage**.

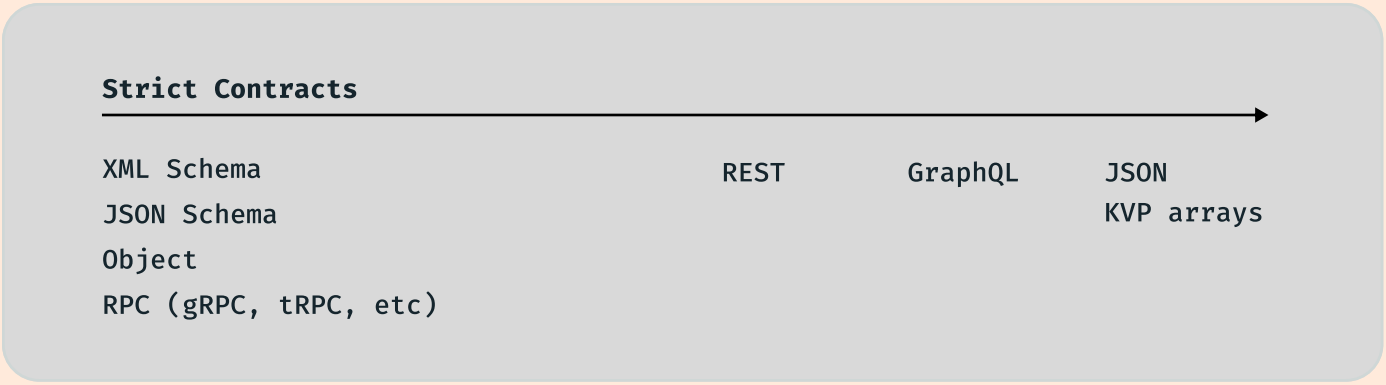
With the pressure to increase who is using our APIs at the center of our platform strategies, we need ways to make our APIs simple to both discover and implement.

The easier it is for frontend and product teams to discover available services on their own, the faster they can design and ship new experiences, and the better they can ask informed questions about potential gaps in functionality.

Insulating Layers

Nothing we ship is perfect the first time. Even if it's close, the technology, teams, and business surrounding it shift and evolve perpetually. Whether planning for a new presentation experience, charting a major version upgrade of an underlying system of record, or responding to a Common Vulnerabilities and Exposures (CVE) incident, we need a way to plan for turbulence in different layers of our stack and devise a strategy for accommodating change without disrupting other teams.

With updates to our services and presentation experiences imminent, we can create a strategy for mitigating the risks of those technical migrations by seeking an insulating layer that combines the benefits of loose coupling with loose contracts.



Magnify Existing Investments

APIs are an investment and a commitment. Beyond the initial cost to design and develop a service, there are years of maintenance, upgrades, and the internal lock-in of many teams having production contracts with any given service. We must find ways to make our existing investments in the services and APIs we already have even more valuable. Throwing away what we already have in favor of a rewrite just isn't practical.

Whether it's by finding new front-end experiences to implement the services we have today or identifying service-level areas for reuse, considering ways to magnify our existing investments is a crucial characteristic of a resilient API strategy.

Strong Governance

When considering governance for our API platforms, we can divide the discussion into two primary areas of concern: Data Governance and Service Governance.

Data Governance is concerned with answering questions like, "Who has access to PII?" and "What experiences are implementing the 'Users' service?." On the other hand, Service Governance answers questions such as, "What is our policy for controlling API sprawl?" and "What is our process for identifying and mitigating Zombie APIs?"

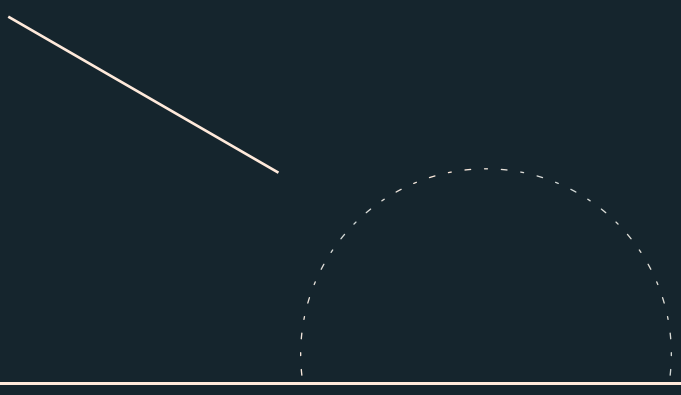
Any API strategy we implement must consider the data and service governance required to secure the future of our APIs without compromising availability, velocity, and reliability.

APOLLO

Chapter 03

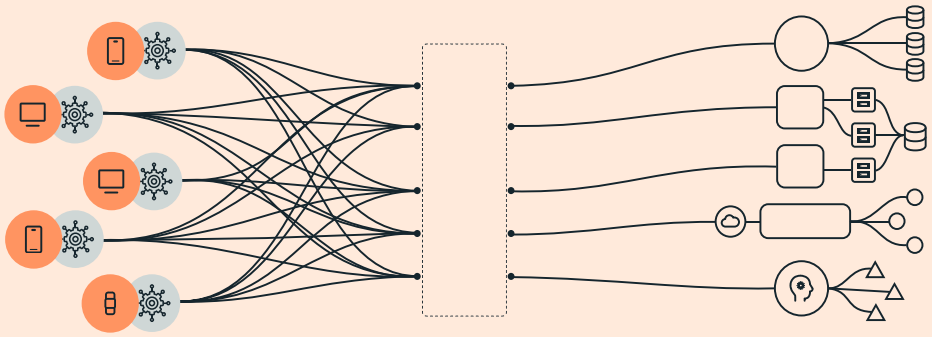
What We've Tried

The rise of Service Oriented Architecture[8] (SOA) came with an increased urgency for defining robust API strategies for our platforms. Before the SOA practices of distributing domains and distinct business capabilities across services, we had one primary approach for sending the correct data to our clients: Baking[9] it into the response sent to the client all at once. While this was an efficient reduction of complexity for monolithic applications, with the introduction of distributed architectures, two different patterns emerged for managing the complexity of the middle layer. Both answer the characteristics of a Resilient API Strategy to varying degrees: Client-side Management and BFFs or Experience APIs.



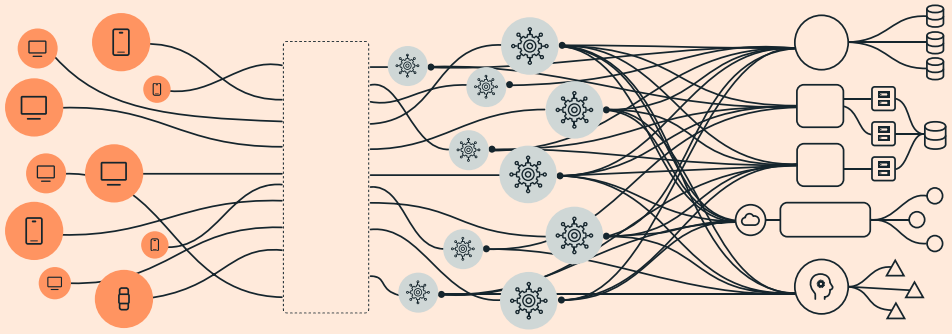
Client-side Management

The first approach for managing and connecting to the sprawl of services created in the SOA migration swung the complexity pendulum to the client. Even today, many API strategies start this way but quickly get out of control. In this pattern, every client is on their own for managing requests to each service required for hydrating a page, and this can quickly lead to inconsistencies and maintainability issues, not to mention issuing dozens of requests to load a single page.



Experience APIs or Backend For Frontends (BFFs)

Experience APIs were the next approach to the problem of resource aggregation as they allow data to be aggregated from many backing APIs, consolidating client-side requests. However, as teams add more and more clients, especially if our companies utilize micro frontends, this can quickly grow out of control, and before we know it, there's a BFF for every micro-front-end.



"While the BFF pattern might originally have meant dedicated backends for each frontend channel (web, mobile, etc), it can easily be extended to mean a backend for each micro frontend."

- Cam Jackson, Thoughtworks[10]

APOLLO

Scoring Client-Side Management and Experience APIs

Client-side Management

Rapid Self-Service

Each client-side team coordinates with each back-end service team, which quickly leads to an overwhelming exercise with people spending more hours on service discovery and meetings than building and shipping new experiences.

Insulating Layer

Shifts complexity from the server to the client. With the client managing many point-to-point connections, there is no insulation between the experience and potential changes in the underlying services.

Magnify Investments

One benefit is that each client-side team can work independently to implement any existing service, reusing them across experiences. The downside is that without an insulating layer, the client-side experiences have relatively raw access to the underlying services driving the business.

Invite Governance

Client-side management doesn't have a paved road for governing who is authorized to interact with given data, and back-end teams are left to manage who owns which service.

Experience APIs or BFFs

Rapid Self-Service

In many cases, Experience APIs were developed and deployed by full-stack teams, creating the front-end experiences they served. Each team could then self-serve their APIs that orchestrated connections to underlying services.

Insulating Layer

Experience APIs typically sit between the Presentation layers and domain-level services. This abstraction provides some level of insulation between the layers and has the benefit of each client connecting to a single endpoint.

Magnify Investments

The BFF pattern benefits from re-using any services that an organization has. Still, we only reap a linear benefit from these as they need to be re-implemented in every BFF.

Invite Governance

The success of a governance strategy depends on the operational maturity of the team designing and implementing the plan. Governance can be a mixed bag with BFFs. In many cases, we saw the sprawl of hundreds of small experience APIs being created and deployed without oversight.

APOLLO

Chapter 04

Building an API Composition Layer with GraphQL

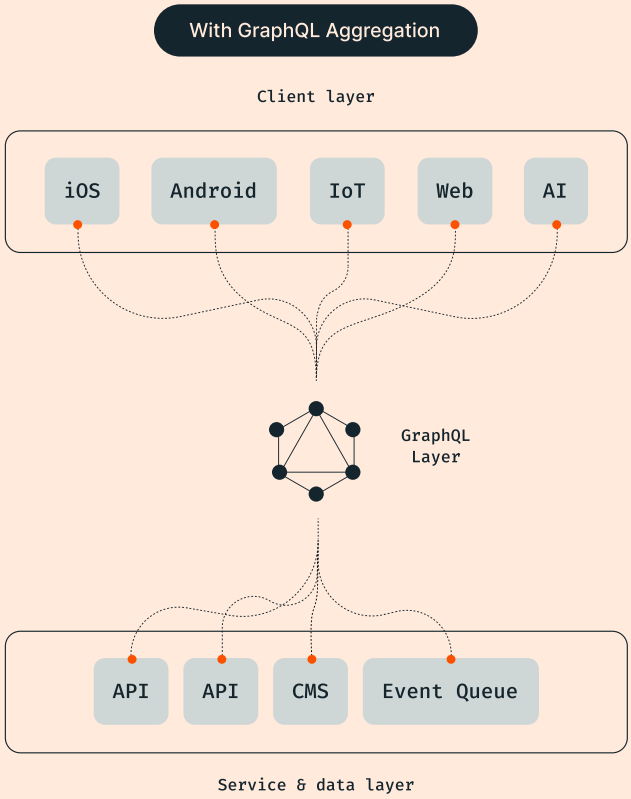
More than Just a Service: a Layer

In 2012, when Facebook began rebuilding its applications native mobile apps, GraphQL was built internally as a mechanism for rethinking their data from the client-side. This perspective applied a narrow lens through which the industry has viewed GraphQL ever since, squarely placing GraphQL as an alternative or replacement to REST – just another API.

In the intervening decade, however, the landscape and tooling evolved significantly with advances such as federated GraphQL architectures[11], a principled, open architecture for scaling GraphQL across any number of clients and services. With a federated architecture, GraphQL is an ideal distinct layer in an API Platform, not just another service.

In a federated GraphQL architecture, teams can maintain individual GraphQL APIs, which teams can write in various frameworks, allowing each team to use their language of choice. This architecture provides the simplicity of a GraphQL monolith for client teams but the modularity of a more decoupled approach for service teams.

These individual GraphQL APIs, or subgraphs, sit behind a single router that serves as an access point for clients. A composition process takes all subgraph schemas and intelligently combines them into a single schema, ensuring a consistent and performant runtime. This supergraph architecture — a graph of graphs — enables service teams to support more clients with greater consistency and less redundant work.



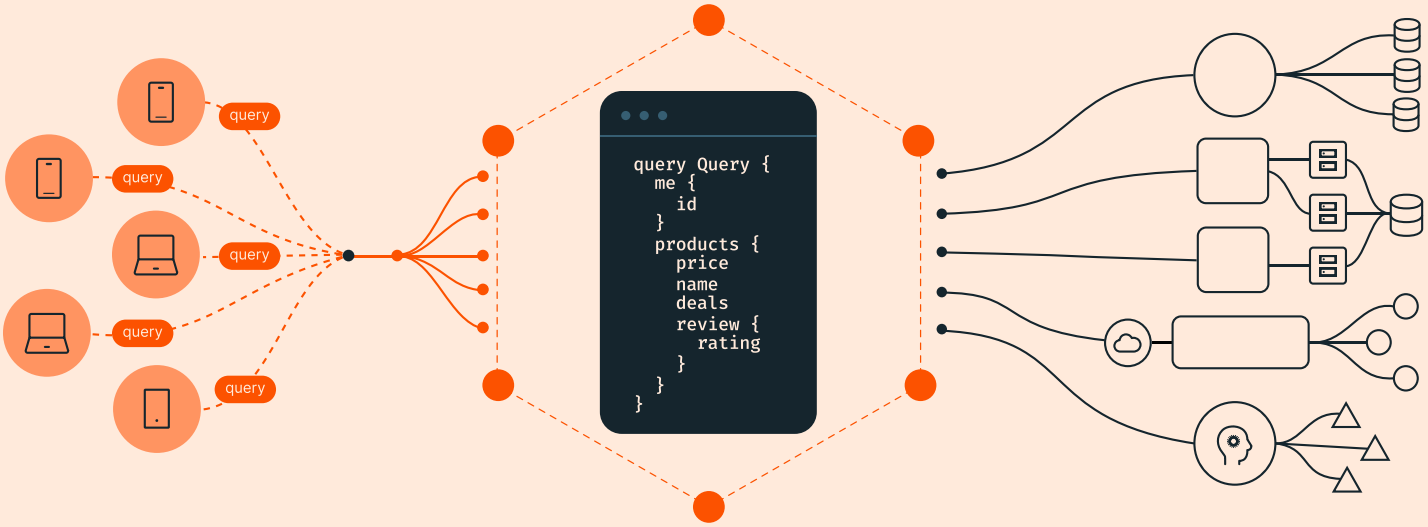
Designing for Resiliency

As our organizations evolve, we will juggle multiple modalities simultaneously. Legacy tech stacks sit adjacent to new-to-us architectures brought in by a merger and are alongside brand-new experiments. When GraphQL is federated, it expands its capabilities beyond just a simple API in a stack. It can provide ecological-style resiliency in an API strategy, accommodating shifts to new equilibriums.

Rapid Self-Service

The first characteristic of a Resilient API Strategy is the ability to provide rapid self-service velocity for feature teams. GraphQL accomplishes this with queries: on-demand definitions of the presentation view's needs, regardless of the data source. Each frontend use case, each micro-frontend, defines a declarative query to fetch precisely the data they need in a single request.

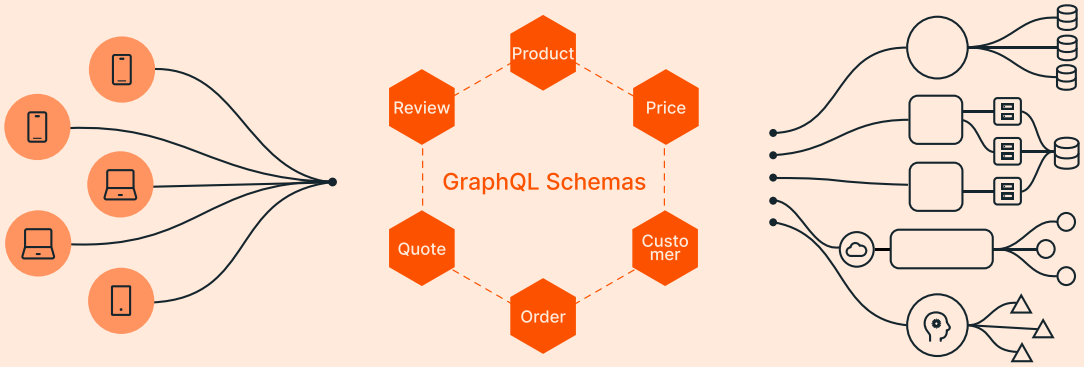
Each frontend use case, each micro-frontend, defines a declarative query to fetch precisely the data they need in a single request.



One key difference between a GraphQL Layer and exposing BFFs is that this is demand-driven API creation without endpoints. We don't need an endpoint for each query; they are all sent to the same single endpoint to be executed by a query planner and distributed efficiently to the underlying application services and data.

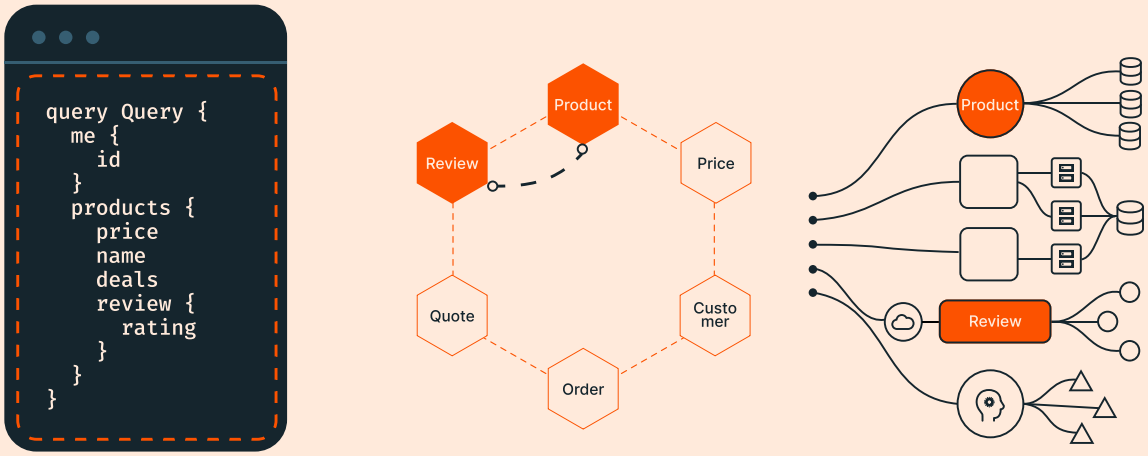
Insulating Layer

The most stable portion of any enterprise architecture are the fundamental nouns that drive the business, such as Customer, Order, or Product. They endure, even as the underlying data storage and service models that define them are re-built. A GraphQL Layer enables us to model these entities in a declarative layer that we haven't embedded in our frontend code or backend services that can survive and evolve.



Magnify Investments

With Federated GraphQL, an API can become more valuable than the sum of its parts. API clients can navigate effortlessly across the API boundary without awareness that they are retrieving data from two different teams, two different APIs that teams could write in different languages, with different API protocols.



Strong Governance

Bringing all these core domain entities into one layer where they can satisfy an unlimited number of tailored queries from our clients without creating a custom endpoint for each creates a single place where our API surface area is defined, rather than hundreds of endpoints and custom code.

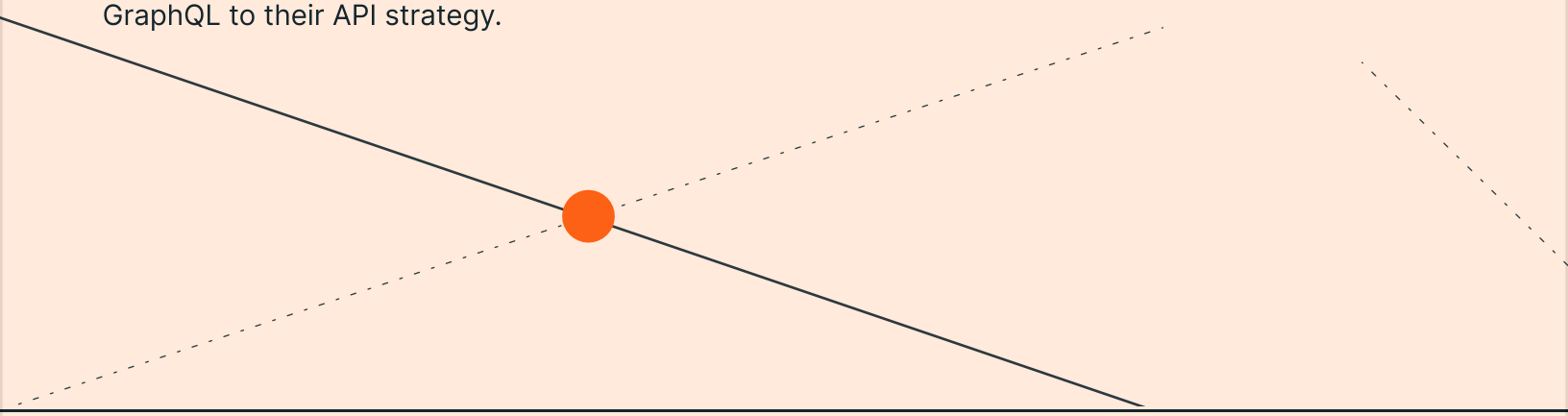
The rise of declarative security in tools like Snyk, HashiCorp Sentinel, Bridgecrew, and OPA has shown us the power of proactively applying policies as codeSecurity and compliance teams can proactively define policies as code to fix vulnerabilities and address compliance concerns across their stack automatically. They can rapidly deploy changes as new concerns emerge. A unified and federated GraphQL layer brings **data governance** (who has access to what) and **service governance** (what services can be deployed by whom and how my API evolves) to our architecture.

GraphQL and Resiliency

GraphQL enables rapid self-service for our presentation clients by replacing one-off handwritten BFF APIs with declarative queries. It provides an insulating contract between our Presentation and Application layers that is flexible enough to manage the constant change while stable enough to prevent breaking changes. By federating GraphQL, this layer goes beyond preserving the value of our existing APIs to increasing and magnifying the value through its ability to connect and compose independent APIs. GraphQL brings a declarative layer and a control plane to the middle, enabling us to apply declarative governance in this crucial layer.

GraphQL in Production

Companies worldwide, including 50% of the Fortune 100, have found runaway success by adding GraphQL to their API strategy.



wayfair

Wayfair, one of the world's leading home goods brands, has reaped the benefits of adding a federated GraphQL layer to its API platform, powering every product search while increasing developer velocity. But running billions of GraphQL queries through any layer poses enormous infrastructure challenges, especially with all eyes on efficiency in 2023. The Wayfair team turned to Apollo Router, Apollo's new Federated GraphQL runtime, in search of a win-win-win. They adopted the Apollo Router just in time for peak holiday traffic, cut cloud costs by 93%, shaved 50% off request latency, and simplified their infrastructure along the way.[12]

VOLVO

Volvo Cars aimed for 50% of their sales online by 2025. They faced significant challenges due to a complex system landscape, leading to development delays, site abandonment, and inaccurate pricing and graphics. Transitioning from a GraphQL monolith to a federated GraphQL architecture using Apollo GraphOS enhanced online sales and injected resilience by eliminating unplanned breaking changes and inspiring developers to innovate with accessible data. Adding a federated GraphQL layer enabled rapid prototyping and implementation of new features like company car policies, proving essential to Volvo's online sales strategy and offering a consistent user journey on their website.[13]

NETFLIX

Netflix, renowned for its microservice architecture, faced challenges with its UI developers managing hundreds of microservices and backend developers grappling with complexity and resilience issues. They introduced a unified API aggregation layer to address this, but as their business and developer count grew, this layer became increasingly challenging to maintain. Netflix adopted a federated GraphQL platform to resolve this, enhancing consistency and development velocity while maintaining scalability and operability.

By adopting federated GraphQL, Netflix successfully addressed bottlenecks in their system and injected resiliency into their API strategy for UI Developers and backend developers. This move to a federated GraphQL architecture, including migrating 350 subgraphs to Apollo Router, highlights their commitment to a unified yet flexible and resilient API strategy, encouraging innovation and collaboration in their organization.[14]

Conclusion

One of the most pressing challenges in platform architecture is balancing sustaining existing, stable systems and preparing for future developments. As businesses face a constant influx of new requirements, acquisitions, and the need to enhance customer experiences, the viability of replacing existing APIs and services hinges on upstream changes. This situation is further complicated by technology teams striving for greater autonomy and speed, highlighting a critical but often overlooked component: the middle layer that bridges the Presentation and Application tiers.

GraphQL is more than an API. With advancements in GraphQL architectures, including federated GraphQL, it's an ideal middle layer capable of driving API resilience in a rapidly evolving technological landscape.

Further Reading

[1] "2023 State of the API Report: API Technologies." Postman API Platform, 2023, www.postman.com/state-of-api/api-technologies/#api-technologies.

[2] Fowler, Martin. "Bliki: Layeringprinciples." martinfowler.com, 2005, martinfowler.com/bliki/LayeringPrinciples.html.

[3] Newman, Sam. "Pattern: Backends for Frontends." Sam Newman & Associates, 18 Nov. 2018, samnewman.io/patterns/architectural/bff/.

[4] Dhapti, Aarti. "Enterprise Software Market Size, Trends - 2030." Enterprise Software Market Size, Trends - 2030, 2023, www.marketresearchfuture.com/reports/enterprise-software-market-2442.

[5] Schadler, Ted. "Mobile Needs a Four-Tier Engagement Platform." Forrester, 18 July 2017, www.forrester.com/blogs/13-11-20-mobile_needs_a_four_tier_engagement_platform/.

[6] Holling, C.S. "Engineering within Ecological Constraints." Engineering Resilience versus Ecological Resilience | Engineering Within Ecological Constraints, The National Academies Press, 1996, nap.nationalacademies.org/read/4919/chapter/4#33.

[7] "2023 State of the API Report: API Technologies." Postman API Platform, 2023, www.postman.com/state-of-api/api-first-strategies/#measuring-success-of-public-apis.

[8] "What Is Service Oriented Architecture." AWS, aws.amazon.com/what-is/service-oriented-architecture/. Accessed 14 Nov. 2023.

[9] Swartz, Aaron. "Bake, Don't Fry." Bake, Don't Fry, 9 July 2002, www.aaronsw.com/weblog/000404.

[10] Jackson, Cam. "Micro Frontends." Martinfowler.Com, 2019, martinfowler.com/articles/micro-frontends.html.

[11] "What Is Federated Architecture?" GraphQL.Com, graphql.com/learn/federated-architecture/. Accessed 15 Nov. 2023.

[12] https://www.apollographql.com/events/champions-corner/how-wayfair-slashed-costs-simplified-infra-and-cut-latency-in-half-with-apollo-router/

[13] https://www.apollographql.com/blog/community/graphql-champions/volvo-cars-drives-into-the-future-of-online-car-shopping-with-the-supergraph/

[14] https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2