+

Powering the API world

Leveraging GraphQL for Next-Generation API Platforms

An Integrated Blueprint from Kong and Apollo

KongHQ.com

Content

Introduction

Gartner projects over 200 billion APIs in use by the end of 2023, reflecting their central role in the creation, operation, and utilization of innovative and secure digital experiences. As digital economies grow, APIs have become critical to every modern application, underpinning communication, integration, and reuse of microservices and data.

Understanding and navigating this complex API landscape is not a trivial task as the pace of technological advancement continues to accelerate. New digital initiatives such as personalization, connected devices, and AI fuel this trend even further. Organizations pursue these initiatives to drive revenue and reduce costs, but execution is paramount. In order to deliver on these initiatives, technology leaders must consider the following:

1. Data that exists across hundreds or thousands of APIs must be delivered across any number of user interfaces across an organization.
2. API teams should be able to design, construct, test, deploy, and iterate APIs efficiently as demand requires.
3. Client teams must be able to consume these APIs in a self-service fashion.

The API ecosystem often positions REST and GraphQL as competing technologies, but they are highly complementary and are instrumental in developing a modern API platform that accomplishes the goals above.

In this paper, we will offer an outline of the entire process — from streamlining the developer's experience in building APIs to ensuring secure, efficient runtime connectivity to implementing robust governance mechanisms.

Through a balanced exploration of Kong's perspective on flexible, heterogeneous API ecosystems and Apollo's expertise in GraphQL, this paper offers a broad view of modern API platforms. Illustrated with real-world architectures and practical insights, you will gain insights into API management in this digital era.

API Management for Modern Microservices

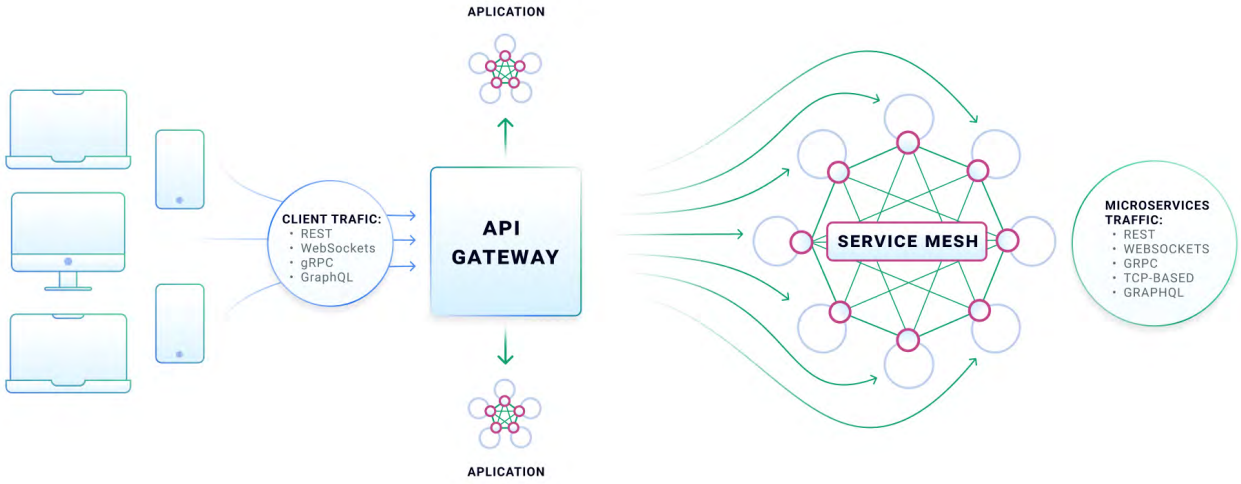
The entire application stack is experiencing fundamental changes, particularly due to the advent of containerization. This shift requires efficient management of microservices and promotes a fully automated DevOps model. Consequently, organizations are better positioned to not only develop, run, and govern new applications and services at scale but also transition legacy applications to modern microservices-based architectures.

APIs are a critical component of these architectures. Recognizing the criticality of APIs, organizations are mobilizing to support their DevOps teams more effectively, giving rise to a new generation of API platform teams. These teams focus on enabling the efficient design, construction, testing, deployment, and operation of APIs in a modern, containerized world.

However, these advances come with challenges. Approximately 50% of enterprise APIs remain unmanaged, posing security threats and decreasing developer efficiency. APIs, available in a variety of technologies and operating across different environments, add to operational and governance complexity. Each API represents a potential attack vector, making API security management a critical requirement for any organization.

The crucial role of APIs in business operations and mission-critical services further underscores the need for performance, scale, and availability. According to a recent Gartner study, 81% of organizations stated that an hour of downtime costs over $300,000, highlighting the significance of API availability.

Addressing these challenges calls for a comprehensive API platform strategy that balances two sets of requirements that are seemingly at odds: security and governance and flexibility and developer productivity. This requires a frictionless, automated approach to building and running APIs that ensures security, discoverability, and availability. The strategy should support multiple points of API connectivity — at the edge and within application boundaries — securely and robustly. Finally, it necessitates an effective governance model that works in tandem with the platform's build-and-run capabilities, reducing developer friction and maximizing API delivery capabilities.



API platform teams must secure, iterate, and monitor APIs, but how can technical leaders unlock developer velocity for client teams by making APIs easier to consume? The next sections of this document outline how GraphQL unlocks this developer velocity, provides best practices for incorporating it into a modern API platform, and presents a reference architecture laying out a path to a secure and efficient modern API organization.

GraphQL Abstracts API Complexity for Client Teams

API lifecycle management tools ensure that backend services are secure, reliable, and can be effectively managed in one place. But how can technical leaders effectively ensure that client teams can efficiently consume data that lives across dozens of APIs in a self-service fashion? And how should API teams that practice domain-driven design (DDD) promote composability and reuse for entities that exist across domains?

Mitigating the complexity of REST APIs is a relatively straightforward task for smaller engineering teams, but it can stifle innovation at an enterprise scale. Engineering teams must ensure that services that live across hundreds of internal and external APIs can be easily understood and consumed by any number of web, mobile, and IoT clients.

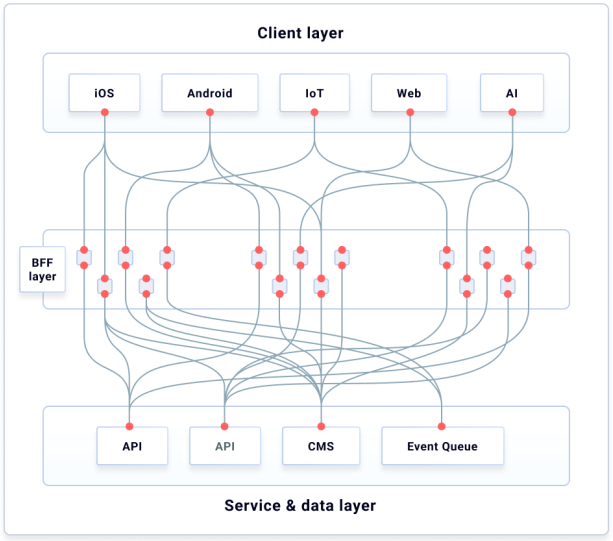
This is not just a technical problem — it is a people problem, as these teams often work in disparate business units, in different geographies, and at different cadences.

Point-to-point connections between clients and underlying APIs present the following challenges:

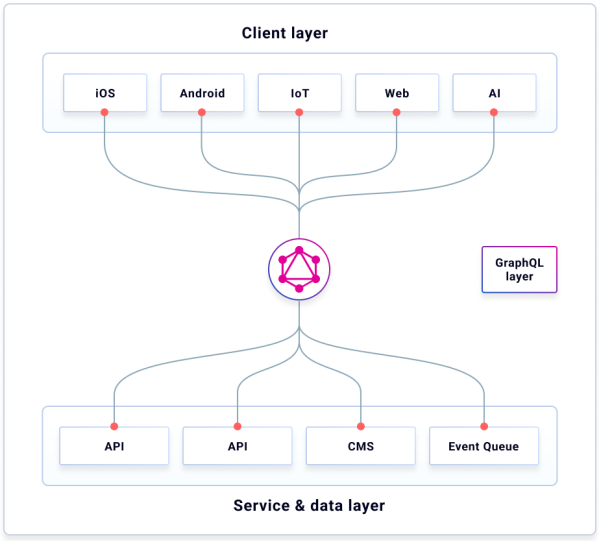
No single source of truth for the data to fetch	Relying on disparate client teams to consume and orchestrate across multiple REST interfaces (using frontend business logic) introduces inconsistencies across applications without significant coordination. Change management becomes unruly as more and more clients are introduced.
Decreased developer productivity	REST API endpoints are fixed, forcing client developers to alter requests to prevent breaking changes across applications. They cannot iterate on features without first understanding the correct paths to use for queries. It is not always clear what teams manage which APIs, and when they have breaking changes.
Performance issues due to over-fetching or under-fetching	When clients have to make multiple sequential network calls across multiple services, it introduces excessive latency. This is particularly painful for mobile apps on slower connections. The cost of maintaining a separate endpoint with exactly the right data for each component becomes untenable as the needs of clients change.

Rather than relying on point-to-point connections between frontends and backends, many organizations began to build backend-for-frontend (BFF) services to abstract this complexity for frontend teams. The BFF pattern simplifies data fetching for client teams, but it does not come without costs. With more and more decoupled architectures, many enterprises have hundreds of BFFs. Each backend-for-frontend presents redundant data fetching code and business logic. In this respect, BFFs have become a necessary evil for many — a new part of the stack that must be built, maintained, and staffed.

GraphQL presents a breakthrough for eliminating hand-crafted BFFs. It enables an API team to define a domain in a much more composable and reusable way. All of the data that defines a business domain is codified in a centralized GraphQL schema on a server. This acts as a "menu" of all of the data available to clients and describes the data's shape. Service developers can define different types of nodes and how they connect/relate to one another. GraphQL resolvers connect GraphQL fields, graph edges, queries, mutations, and subscriptions to their respective data sources and microservices. Client teams can then access this data using a declarative query language and a single GraphQL endpoint that sits on top of the underlying domain APIs.



Before

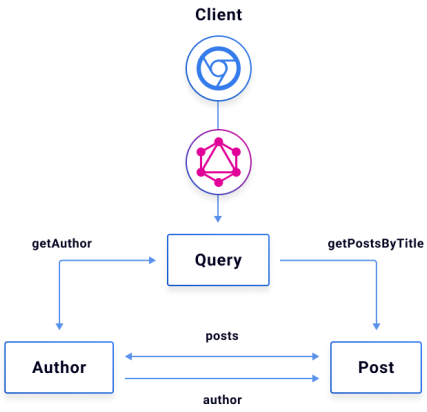


After

The internet is rife with “GraphQL vs. REST” debates, but it is not an either/or scenario. The GraphQL Schema Definition Language (SDL) can define disparate concepts from data that lives across a variety of services, including REST, gRPC, and even SOAP APIs. GraphQL servers can simplify fetching data from underlying APIs and help handle caching, request deduplication, and errors while resolving operations.

Apps can fetch all the data they need in a single GraphQL query, without waiting on BFF teams to implement a hand-crafted experience API. Multiple apps can share a common GraphQL API contract to reduce duplicate effort and improve consistency in the customer experience.

Client teams use a query language to describe the data they need, and only have to query a single endpoint.



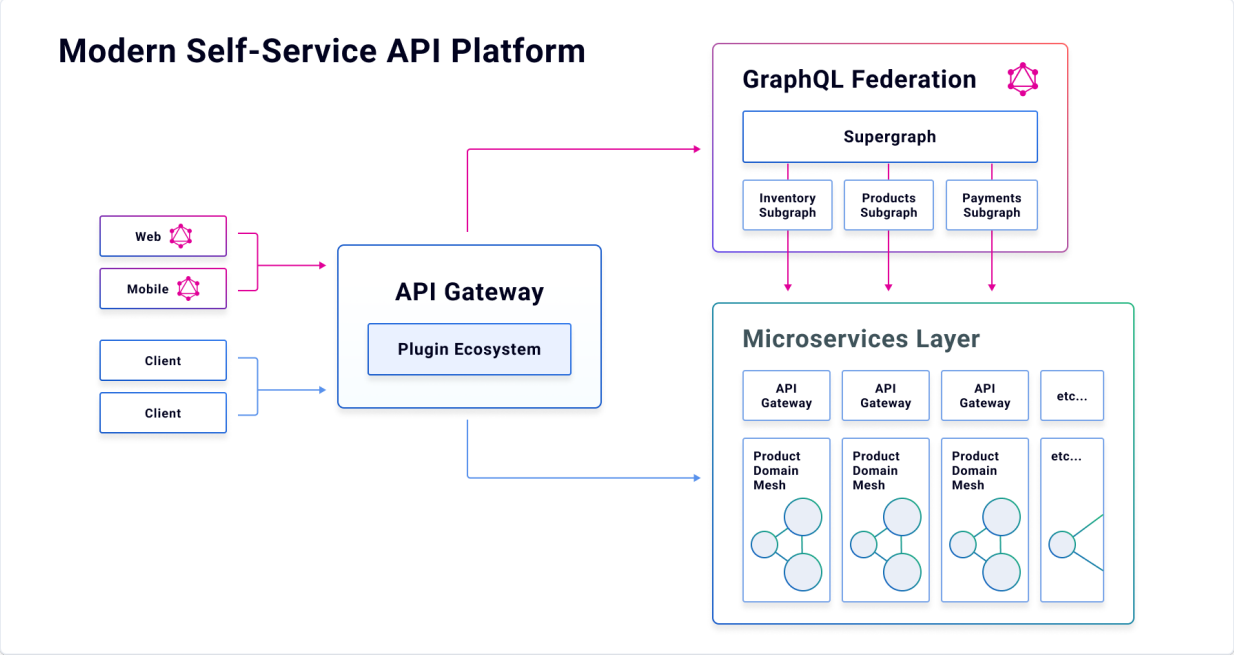

```
type Author {
    id: Int !
    firsName: String
    lastName: String
    posts: [Post]
}

type Query {
    posts: [Post]
    author(id: ID!): Author
}
```

```
type Post {
    id: Int!
    title: String
    author: Author
    votes: Int
}
```

Key Pillars of a Modern API Platform with GraphQL

With the core requirements and technologies above, we can now outline how these technologies fit together within a modern API platform. There are three fundamental pillars: API gateways, GraphQL Federation, and a microservices layer.



API Gateways for Managing Connectivity at the Edge and Between Applications

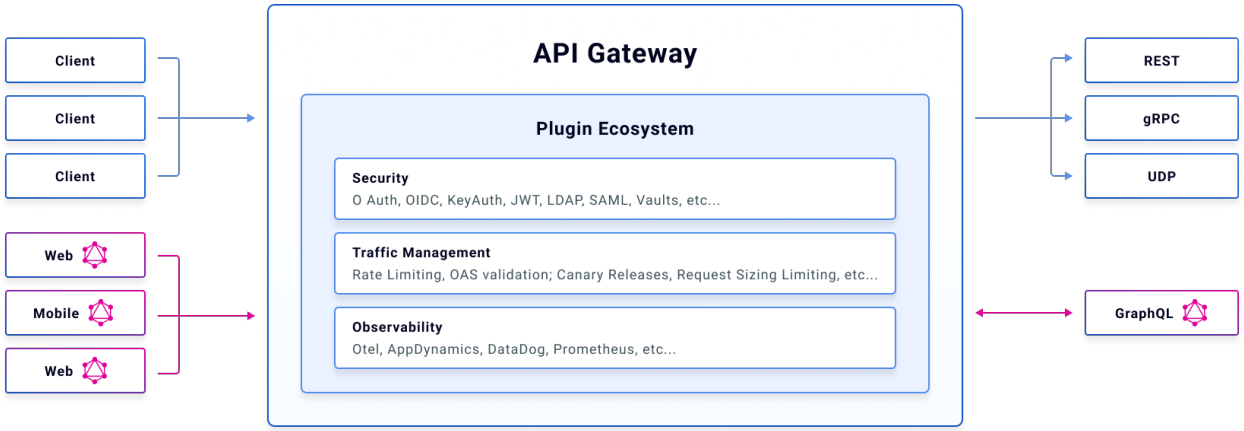
The first pillar, the API gateway, provides the critical function of mediator between API consumers and providers. API gateways are deployed in multiple locations through the architecture. First, an API gateway is positioned at the infrastructure's edge, negotiating traffic between front-end API requests and backend services. Additionally, API gateways are placed within the infrastructure to mediate traffic between application boundaries.

The gateways do much more than just this: they provide a critical layer of isolation for services, enforce security, and manage traffic flow. The gateway's position in the architecture necessitates high performance for large-scale traffic handling.

Modern gateways must support next-generation protocols like GraphQL, gRPC, and Kafka while coexisting with existing services that utilize REST or lower-level protocols. Even further, API gateways must support a broad set of requirements across key aspects such as security, traffic management, and system observability.

Diverse and evolving security standards are difficult to implement at the service layers. The API gateway provides a flexible, unified, and consistent security layer for all the services. Traffic management techniques, like rate limiting, can provide basic service availability protection but also enable direct business value features like customer tiering.

Gateways must provide API observability for real-time performance tracking and enable seamless day-2 operations with features like zero-downtime upgrades, auto-scaling, and self-healing capabilities. Engineering teams need gateways to operate across a wide range of compute infrastructure, from virtual machines to Kubernetes and other containerized platforms. Modern teams require end-to-end automation capabilities and flexible governance models.



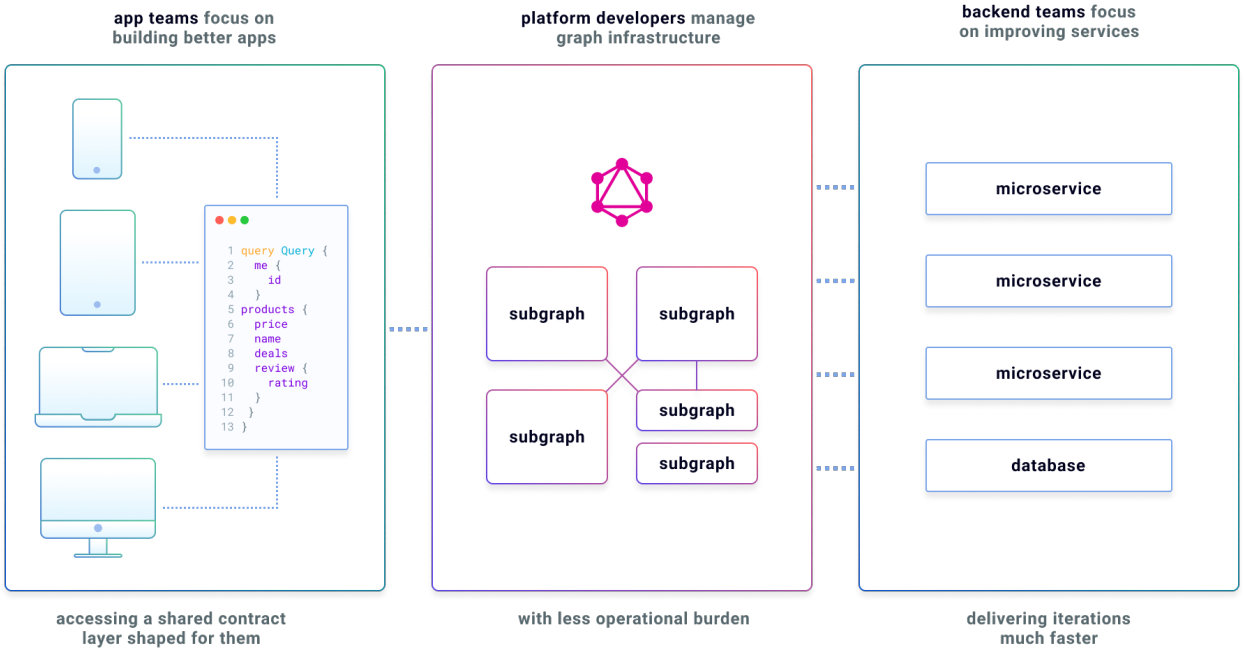
A Supergraph Provides a Service Access Layer for Client Teams

GraphQL provides a standardized schema definition and query language to access all the data and services in an organization — a layer on top of your existing APIs. When a monolithic GraphQL server is used to replace one or multiple BFFs, it can quickly grow in size. This monolithic GraphQL API can quickly become a bottleneck that is untenable for one team to manage. Other teams that want to contribute to the graph are forced to use the language and framework picked for the GraphQL monolith. And as the graph increases in size and complexity, there is no clean separation of concerns for ownership. So how do you use GraphQL at scale?

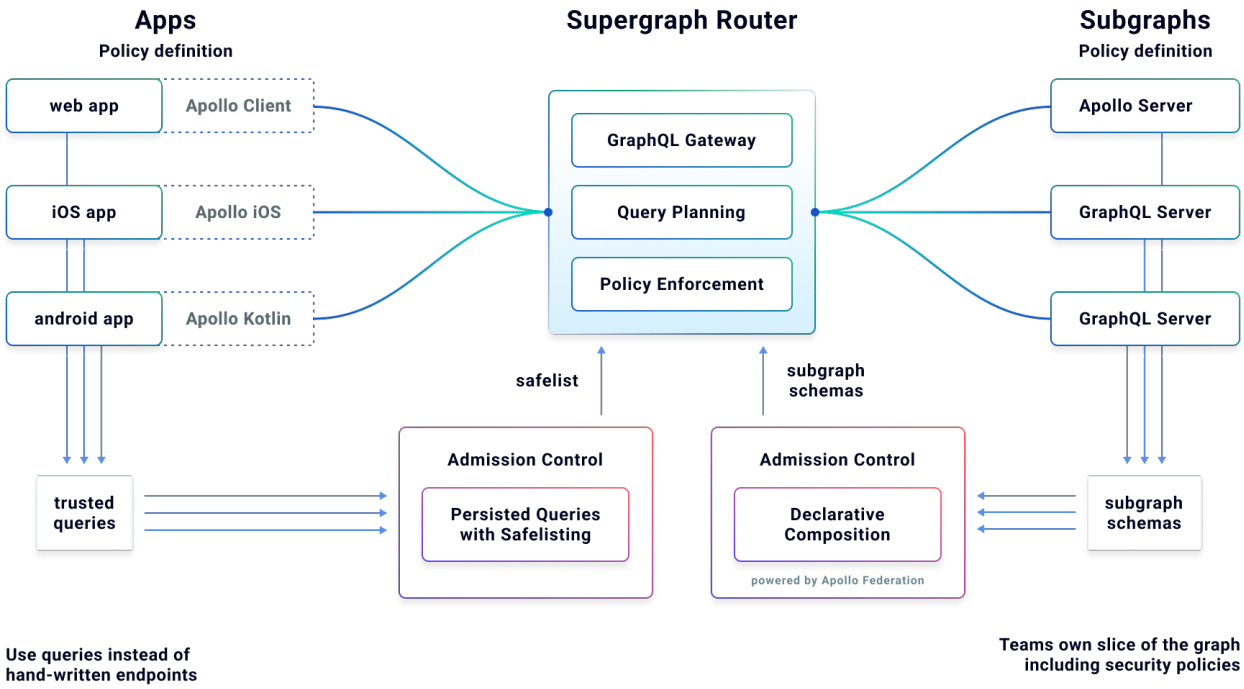
Companies like Netflix use GraphQL Federation as a better way to scale GraphQL. This architecture provides the simplicity of the monolith for client teams but the modularity of a more decoupled approach for service teams. In a federated graph, individual GraphQL APIs are maintained by separate teams on separate servers and can be written in a variety of subgraph frameworks, so each team can use their language of choice. These individual subgraphs sit behind a single API router that serves as an access point for clients. A process called composition takes all subgraph schemas and intelligently combines them into one schema, ensuring a consistent and performant runtime This defines a supergraph architecture — a graph of graphs — and enables service teams to support more clients with greater consistency and less redundant work.

A supergraph should continuously evolve to meet the needs of the business. Automation and tooling are key to evolving a graph without introducing breaking changes. Teams publish subgraph schemas to a schema registry as subgraphs are deployed and become available for use. The registry provides a source of truth for a supergraph continuous integration (CI) build that composes subgraphs, checks linting rules, and assesses the impact of potentially breaking changes using observed traffic.

When a supergraph build completes, the composed supergraph schema is continuously deployed to a fleet of supergraph routers, so applications can immediately query and use new fields and types.



Beyond distributed schema ownership, a self-service API platform requires that security and performance policies also be owned in a modular fashion. Teams can define policies along with their schema and code changes to streamline development and reduce errors associated with multi-team handoffs.



These graph-native policies are declaratively composed and validated by the supergraph CI build pipeline and delivered to the supergraph router which enforces the policies at runtime, at the edge of your graph. Enabling teams to own their slice of the graph, including security and performance policies, is key to achieving developer self-service.

In summary, a supergraph provides the self-service experience of GraphQL at scale:

1. **A scalable architecture:** A supergraph should provide modularity for service teams and an intuitive experience for client teams. It should perform all of the orchestration required to deliver GraphQL at scale.
2. **Reliable and secure infrastructure:** API teams should be able to maintain, secure, scale, and observe the graph. Organizations must consider the unique requirements around performance and in particular caching and deduplication, as well as graph-native security including schema-driven authorization, operation safelisting, query depth and cost limits, quotas, rate limiting, and traffic shaping.
3. **Standardized workflows:** Teams should be able to collaborate on their portion of the graph for optimal developer experience and composability. The supergraph CI/CD (continuous integration/continuous delivery) build should be automated using a schema registry as a source of truth. Organizational best practices should be ensured with static analysis and linting rules. Breaking changes should be prevented using schema and operation checks against observed runtime traffic for smooth operation in production environments.

Service Mesh for Managing the Microservice Layer

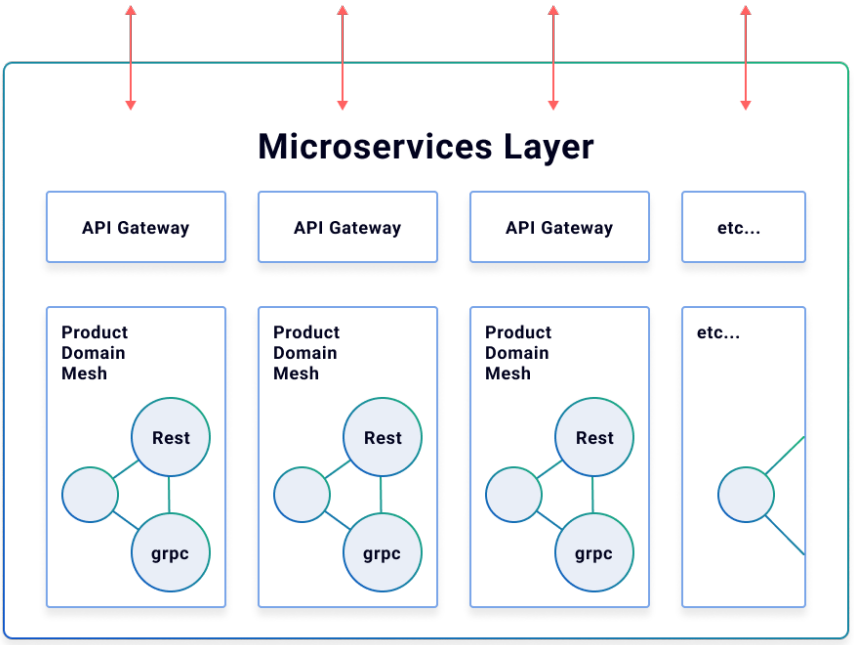
At the foundation of the architecture is the microservices layer, where services are managed by autonomous teams following domain-driven design principles. These services are the data access layer, delivering the essential business functionality to clients, including end-user applications, GraphQL subgraphs, or other domain services. Services may provide their functionality using a variety of technologies including REST, gRPC, and proprietary protocols.

Modern engineering teams are turning to service meshes to manage the microservices layer. A service mesh offers a dedicated network layer that provides automated service discovery, load balancing, automatic retrying, circuit breaking, and failure recovery. Combined, these functions improve availability and responsiveness even in the face of network failures or service disruptions. Service meshes also provide advanced capabilities such as A/B testing, canary releases, and rate limiting.

Security is another critical advantage of a service mesh. It enables consistent and centralized security policies to be applied across all communication channels. Features like mutual TLS (mutual transport layer security or mTLS) authentication, authorization, and encryption can be readily implemented and enforced. This safeguards sensitive data and aids in preventing unauthorized access and potential breaches. Modern service meshes enable secure communication across the full distributed environment by adhering to zero-trust security principles.

Observability, which involves understanding the behavior of applications, is significantly improved through a service mesh. Features like distributed tracing, metrics collection, and logging provide valuable insights into the flow of requests and responses within the application. This facilitates quicker identification and resolution of performance bottlenecks or errors, leading to enhanced overall system performance.

Inter-domain traffic is managed by an API gateway, allowing the same principles applied at the networking edge to mediate traffic between application boundaries.



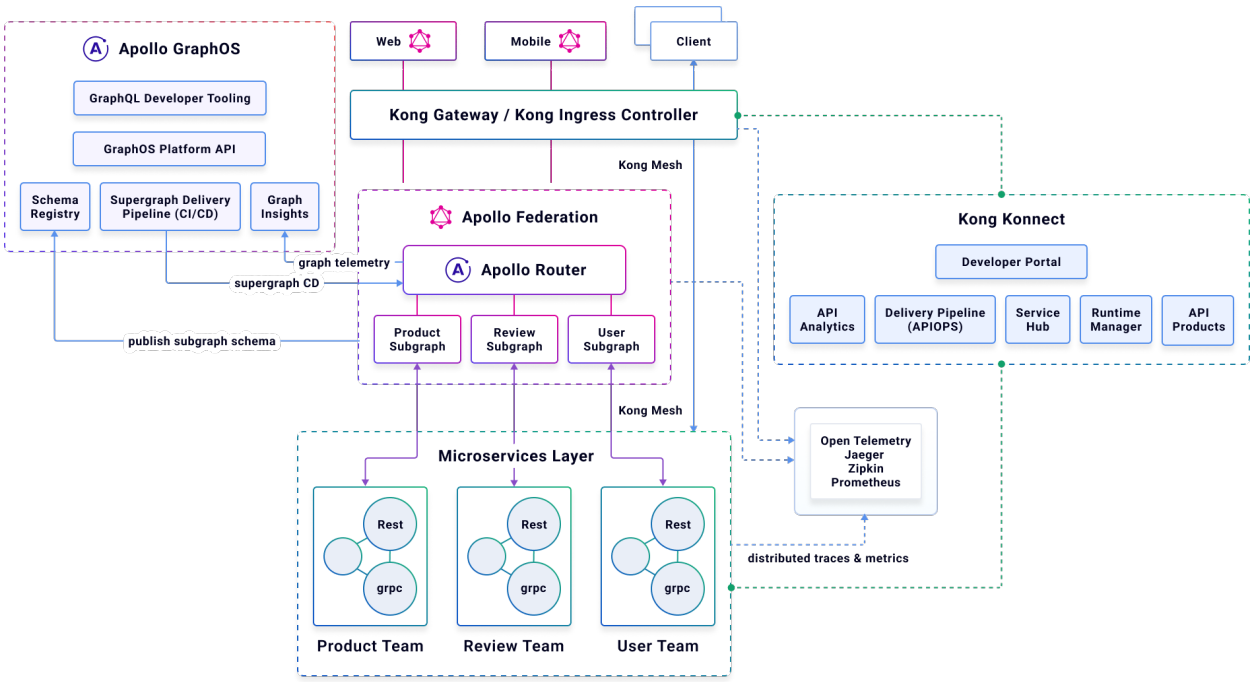
In summary, a service mesh offers multiple benefits that enhance application reliability, security, and observability. It abstracts communication complexities, bolsters security, and provides deep insights into service behaviors. Moreover, its adaptability to diverse technological ecosystems ensures that organizations can build and manage resilient and secure distributed applications effectively.

Reference Architecture for a Modern API Platform

The following reference architecture details how Kong and Apollo technologies can be used to build, deploy, and operate modern APIs. Included is an overview of how a GraphQL request is processed at runtime through the different layers of the architecture.

Kong provides an API gateway (Kong Gateway), service mesh (Kong Mesh), and multi-cloud API management (Kong Konnect) for the microservices in your organization. You can preserve existing investments and workflows for your domain APIs and enhance them by weaving a GraphQL layer into your existing architecture. Apollo provides a GraphQL developer platform (GraphOS), which includes developer tooling, a schema registry, and a supergraph CI/CD pipeline and high-performance supergraph runtime (Apollo Router).

Several aspects of this reference architecture are cross-cutting in nature, including security, traffic shaping, and observability. Security is often handled with a defense-in-depth or zero-trust approach, where each layer of the stack provides security controls for authentication, authorization, and blocking malicious requests. Client-side traffic shaping with rate limits, timeouts, and compression can be implemented in the API gateway or supergraph layer, and subgraph traffic shaping (including deduplication) can be implemented at the supergraph layer. Observability via Open Telemetry is supported across the stack to provide complete end-to-end visibility into each request via distributed tracing along with metrics and logs.



Build, Deploy, and Operate APIs with a Modern API Platform

Every layer of the modern API platform presents unique challenges for building, deploying, and managing APIs. Kong and Apollo technologies tackle these challenges, promoting developer efficiency and autonomy while ensuring that organizational and design best practices scale seamlessly across multiple teams.

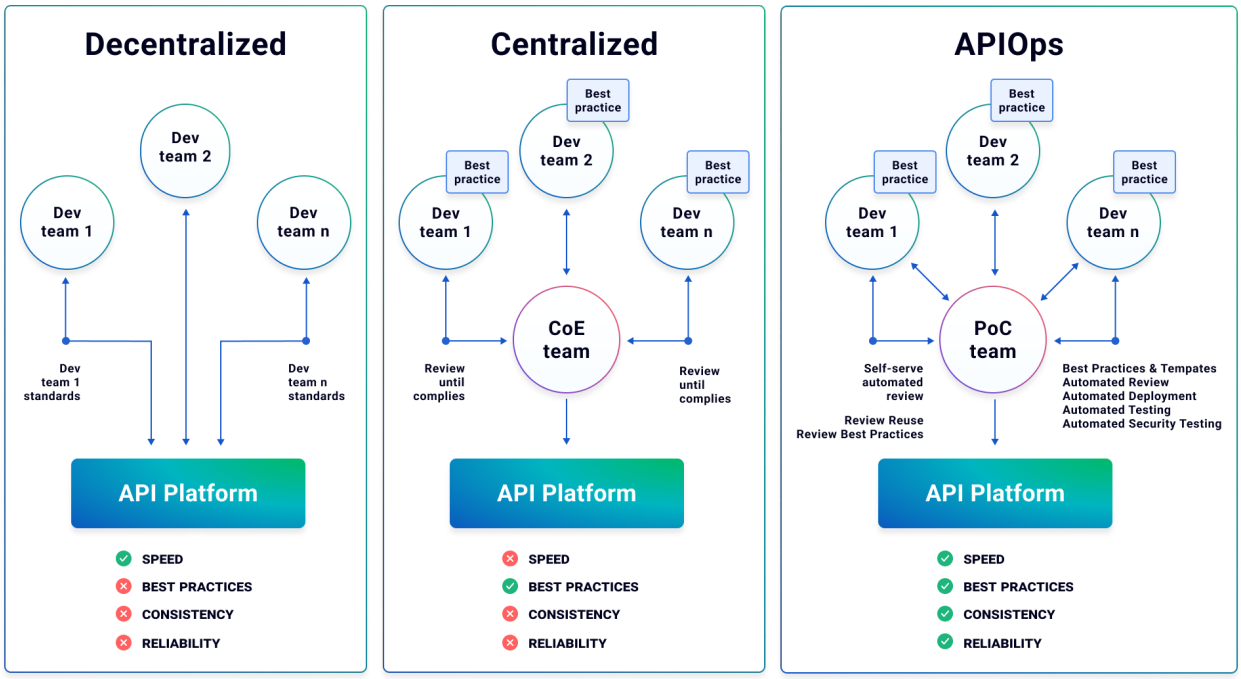
Developer tooling is designed to integrate into your existing software development lifecycle (SDLC) and CI/CD workflows. Developer portals make API service discovery simple, and schema registries make it easier to publish and consume APIs at different layers of the stack. API security and performance policies can be defined and managed at each layer, along with observability capabilities to understand API usage, the impact of breaking changes, and operational concerns like performance and monitoring. Each layer of the stack has tools to make enabling these capabilities easy for platform engineers.

API Gateway

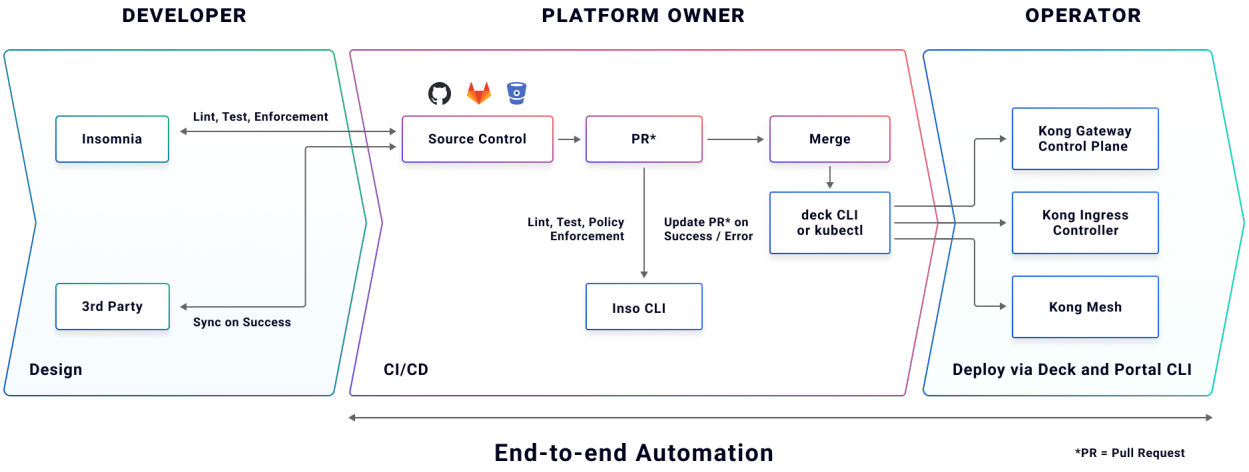
Kong Gateway is the world's most adopted API gateway. It offers strong performance, scalability, and extensibility. Compatible with various systems, including bare metal, Kubernetes, and other containerized platforms, the gateway accommodates a variety of protocols and can integrate with both traditional and newer technologies.

Kong Gateway is designed to optimize today's application modernization needs through automation across the full lifecycle of APIs and microservices. With Kong Gateway, developers can instantly add traffic control, security, authentication, and transformation functionality leveraging a wide array of out-of-the-box plugins ensuring best practices are followed without stifling flexibility and productivity. Kong Gateway can also run natively on Kubernetes, with Kong Ingress Controller.

APIOps is a process that takes the proven principles of DevOps and GitOps and applies them to API platform management. Kong provides full support for APIOps automation to ensure reliable and repeatable API delivery. Kong Gateway configuration can be managed using a REST-based API or a modern declarative configuration system including drift detection. Full and partial declarative configurations can be stored in version control systems and assembled, validated, and applied through CI/CD pipelines.



This tooling enables developers to autonomously design and develop APIs, including generating gateway configurations from OpenAPI Specifications (OAS). Federated governance defined by the API platform team is applied to both the individual service and the composed API, ensuring that delivered APIs are consistent, reliable, and secure.



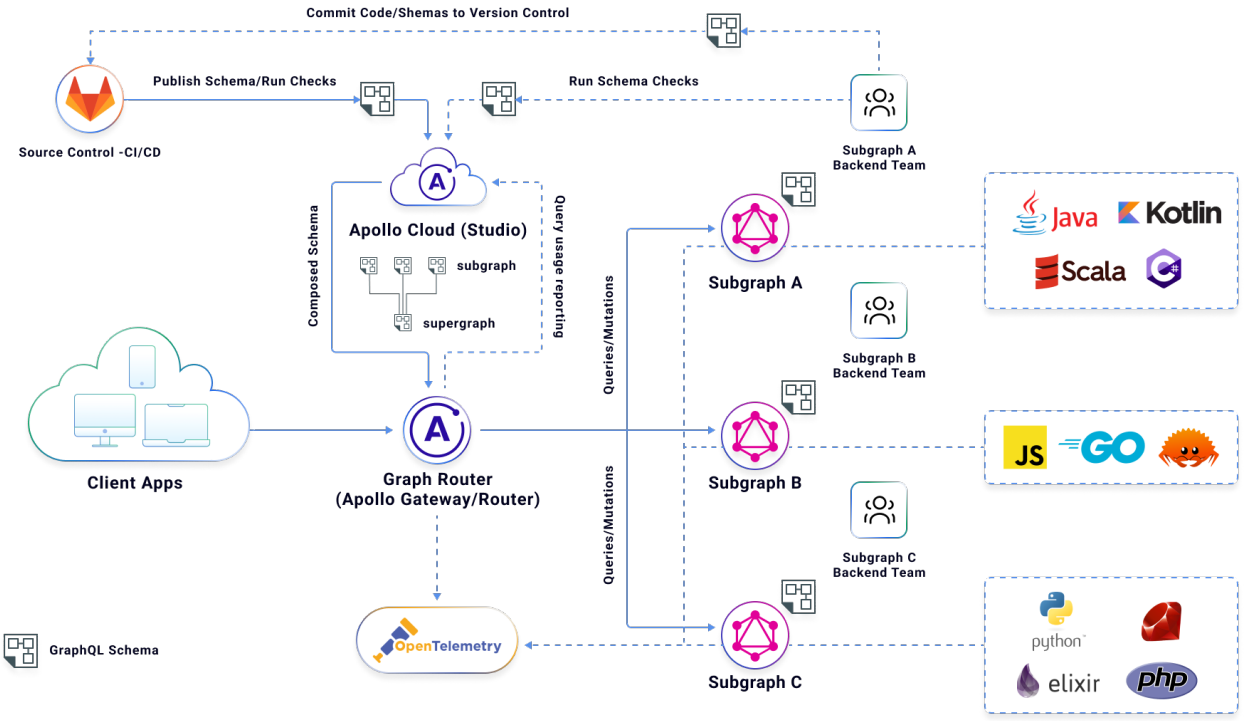
Supergraph Developer Tooling and CI/CD Pipelines with Policy Controls

Apollo GraphOS provides everything needed to build and deploy any number of GraphQL changes per day with streamlined multi-team collaboration powered by Apollo's federated GraphQL architecture. GraphOS provides a schema registry, managed build pipeline with customizable schema linting, graph-native observability with developer decision support, and breaking change prevention to ensure changes can be deployed with speed and safety. GraphOS includes developer tooling for app developers (schema explorer, query builder, docs) as well as subgraph developers (30+ compatible subgraph frameworks, schema insights, Rover CLI for dev laptop and CI/CD pipelines).

A platform team is typically responsible for configuring the supergraph build pipelines in Apollo GraphOS and configuring and operating the supergraph runtime fleet for each environment (dev, test, prod) using Apollo Router. This enables subgraph developers to autonomously own and contribute their slice of the graph without the platform team becoming a bottleneck. Graph variants are created for each environment (dev, UAT, prod) and security is configured to restrict who can publish to each supergraph variant.

With the supergraph infrastructure in place, each subgraph team can independently build and deploy their subgraphs, and then make them available in the supergraph by publishing to the Apollo GraphOS Schema Registry. This is typically done as the last step in a CD workflow or with a post-deploy analysis job if a progressive delivery controller is used — once an updated subgraph is ready to accept traffic.

Once a new subgraph schema is published, GraphOS automatically composes it into a new supergraph schema, runs linting checks defined by the API platform team to ensure best practices, and if all checks pass, the router fleet will pull a new supergraph schema via CD via a multi-cloud Apollo Uplink endpoint.



Subgraph teams can also shift left the detection of breaking changes with GraphOS schema checks, using developer tooling (Rover CLI) both on the developer laptop and in the CI workflows for each subgraph. GraphOS provides GraphQL field usage insights that help API developers understand which apps and app versions are using a field, how much traffic, and the potential impact of breaking change and which app they need to coordinate with.

Apollo Router processes all inbound GraphQL requests, plans and executes requests across subgraphs, and enforces graph-native policies for security, performance, and operational concerns. At this point, the router fleet is able to serve traffic and the Apollo GraphOS provides developer documentation with updated supergraph schema, so application teams can immediately use new fields and types in the GraphQL queries that apps need to power new customer experiences.

Microservices Management with Service Mesh

Microservices bring many benefits, but also many disadvantages. The network is unreliable, service discovery is hard, and there are few controls around who is accessing data from each API.

Service meshes abstract these concerns away from developers, reducing the complexity of service development by removing the need to manually build these capabilities directly into service code. This allows development teams to concentrate on building value instead of redundant connectivity code across languages and projects.

Kong Mesh is an enterprise-grade service mesh built on top of Envoy that solves these issues with a focus on simplicity, security, and scalability. Kong Mesh automates service discovery, security, advanced traffic routing, observability, and failover logic using centralized policies. These policies are applied through a REST-based management API or through a kubectl-like declarative resource model.

Kong Mesh additionally supports zones, which allow organizations to model physical network connectivity across their environment. Platform teams can design zones around concepts such as Kubernetes clusters, cloud providers, regions, data centers, or network latency boundaries. Services within zones can communicate directly, and cross-zone communication is handled out-of-the box. Service meshes can be deployed within or across zones for maximum flexibility.

Kong Mesh can be installed and managed standalone or through Kong Konnect, and it operates on Kubernetes or virtual machine deployments. Meshes can be created per line of business, team, project, or environment thanks to Kong Mesh's ability to create multiple isolated service meshes within the same cluster.

Ingress traffic to a mesh is handled by an API gateway. Kong Mesh supports many different ingress controllers, but it works best when paired with Kong Gateway via the Kong Ingress Controller. All external traffic passes through the gateway and is subject to any policies defined at the gateway level, similar to those defined in the edge gateway.

The API Request Lifecycle

To help visualize how this architecture works, let's look at how a GraphQL query request traverses the architecture layers in some detail.

Kong API Gateway

Kong Gateway sits at the edge of the network to proxy all incoming traffic. In a GraphQL request, this is typically an HTTP POST with the GraphQL query in the body of the request along with any required headers. This example shows an Authorization header containing a JSON Web Token (JWT) and a query to fetch information about the current user and a list of products:

```
POST /graphql HTTP/1.1
Host: example.com
Accept: application/json
Content-Type: application/json
Authorization: Bearer ...
{ "query": "{
      me {
        id
     }
     products {
       price
       name
       deals
       review {
        rating
       }
     }
 }" }
```

Service Routing

Kong Gateway evaluates the incoming HTTP request against configured Routes and attempts to find a match. The gateway can match routing based on a number of capabilities, including HTTP methods, host, headers, request path, and Server Name Indication (SNI). Kong Gateway supports requests over HTTP, TCP, and GRPC protocols.

In this example, a route is configured to match the path /graphql on the host example.com which matches the above request.

Authentication / Authorization

Once the Route and Service are determined, the request is run through all configured plugins, starting with security. In the above GraphQL example, JWT authentication is used. Kong directly supports JWT and dozens of other security and authentication technologies through its extensive plugin ecosystem. Requests are verified to contain valid signatures and claims, and violating requests are immediately rejected.

Traffic Management

GraphQL adds new challenges when it comes to traffic management. Not all requests are equal, which makes traditional rate-limiting strategies ineffective. Kong Gateway provides specialized GraphQL plugins that take query complexity into account when implementing rate limiting.

As GraphQL is an HTTP request at its core, you can use any of Kong's other Layer 7 plugins such as proxy caching, request validation, configurable upstream timeouts, and even mocking to unblock the UI team as the subgraph service is being built.

Kong plugins define a logical default processing order. However, operators may dynamically build a plugin dependency graph to define the plugin execution order. A common example used is allowing rate-limiting evaluation to be processed prior to authorizing requests. Requests violating any configured traffic management rules are immediately rejected.

Observability

The OpenTelemetry (OTel) standard has driven observability into the mainstream. Understanding the request lifecycle and where time is being spent is key to building reliable, performant APIs.

Kong Gateway and Kong Mesh both support OTel without any additional dependencies. Information about plugin execution, DNS lookups, and upstream performance are generated and propagated to any OTel protocol (OTLP) compatible server. Along with OTel, Kong supports many popular analytics, monitoring, and streaming platforms including DataDog, AppDynamics, Zipkin, Prometheus, Kafka, and more.

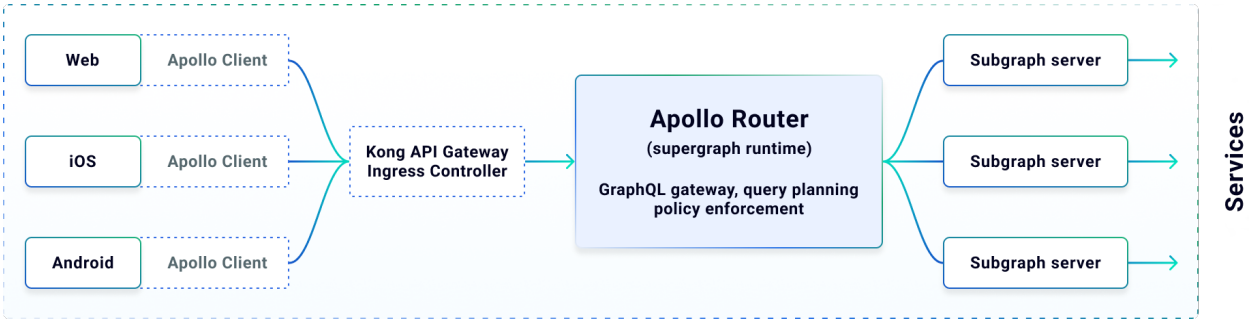
Request Forwarding

Once Kong Gateway has executed all configured plugins, it is ready to forward the request to the upstream service. Kong Gateway supports load-balancing capabilities to distribute requests across a pool of instances of an upstream service. Requests are allocated using a round-robin algorithm by default, but Kong also supports consistent-hashing, least-connections, and latency. The latency algorithm uses peak EWMA (exponentially weighted moving average), which ensures that the balancer selects the upstream target by lowest latency.

For our example GraphQL request, Kong Gateway forwards the request on to the Apollo Router, which is a high-performance supergraph runtime.

Supergraph Runtime Execution

Apollo Router processes all inbound GraphQL requests and plans and executes requests across subgraphs. In order to ensure performance, Apollo Router was written in Rust to ensure increased throughput, reduced latency, and reduced variance. It also enforces graph-native policies for security, performance, and other operational concerns, which can be configured by platform teams in YAML, Rhai scripting, or their language of choice.



GraphQL Query Parsing & Validation Against the Public API Schema

As a spec-compliant GraphQL server, the router parses and validates each GraphQL request to ensure the query conforms to the GraphQL schema. Apollo Router is powered by the declaratively composed supergraph schema that contains every type and field in your graph and which subgraph(s) they can be fetched from, including various federation directives that define build and runtime policies. The full supergraph schema may internally include fields that are otherwise `@inaccessible` for applications to use, and these are not included in the public API schema that the router validates GraphQL requests against.

Graph-Native Security and Performance Policy Enforcement

GraphQL-native security and performance policies are enforced early in the request lifecycle to block malicious traffic at the edge of your supergraph and protect the underlying microservices from excessive load. For example, subgraph schema directives like `@authenticated` and `@requireScopes` enable the Apollo Router to dynamically calculate the required JWT claims to access all the fields in a query and gracefully degrade the response by removing unauthorized fields from the query along with a suitable field-level error. Apollo Router can enforce multiple GraphQL-native security policies, including query depth and height limits, contracts, and a safelist of known queries.

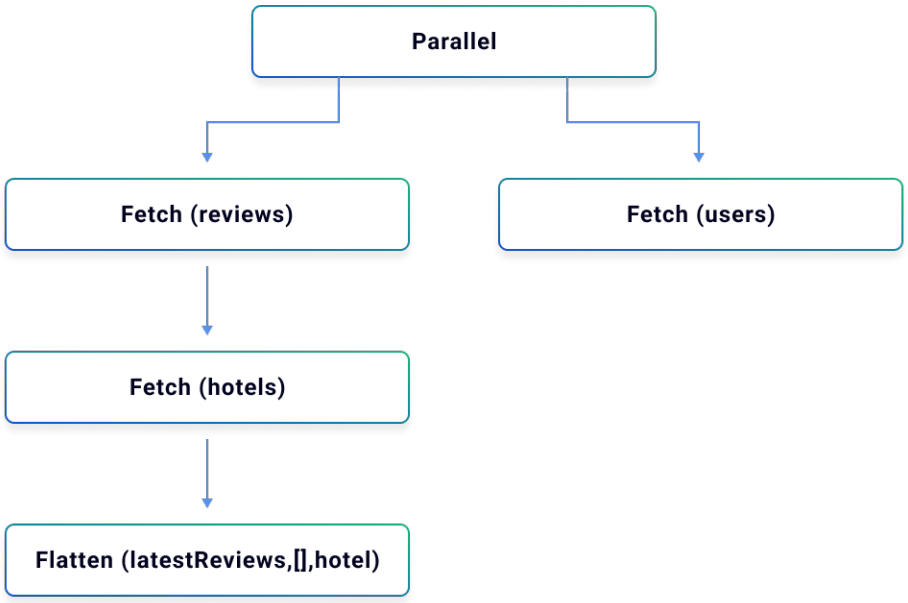
Intelligent Query Planning

Apollo Router creates an optimized query plan for each GraphQL operation (query, mutation, subscription) that minimizes the cost, complexity, and latency for each query, so it can optimally fetch and join data from multiple subgraph APIs into a single unified response for apps to consume.

With Apollo Federation 2, fields can be denormalized across subgraphs for improved performance using the `@shareable` subgraph directive to relax the default single source of

truth. Multiple subgraphs can provide the same root query fields, and subgraphs can `@override` a field to migrate it from one subgraph to another. When clients need to defer the slow part of a query, often due to a slow underlying REST API, they can specify the `@defer` directive in a query so the rest of the query can be returned immediately for more responsive UX.

Apollo Router's query planner is able to take all of the supergraph schema and directives into consideration and generate an optimal query plan. This minimizes the number of subgraph API calls and adheres to the declarative policies defined in the subgraph schemas and composed into the supergraph schema that powers the router.



Query Execution and API-Side Joins

The router then executes the query plan in parallel where possible and in sequence when API-side joins require data to be fetched from multiple subgraphs in sequence. To support API-side joins of GraphQL data (similar to how tables are joined in a database) each of the over 30 subgraph frameworks provide the ability to fetch additional data for a GraphQL entity using the available `@key` fields defined in the subgraph schemas.

The router can then satisfy the client query by orchestrating the subgraph API calls using both the standard GraphQL query, mutation, and subscription fields a subgraph provides, along with the ability to fetch additional entity fields to process API-side joins. The final result is then flattened into the requested query shape and returned to the client.

Supergraph Observability

Graph-native telemetry is emitted by the router during request execution, so Apollo GraphOS can power field usage insights and schema checks that assess the impact of potentially breaking changes. This includes query shape, field usage, and optionally select headers — but not the query response or Authorization header.

Open telemetry tracing and metrics are also natively supported by Apollo Routerd to support APM use cases for performance, monitoring, and alerting. Built-in support for Datadog, Jaeger, Open Telemetry Collector, Zipkin, and Prometheus is also included — along with options for sampling, limits, and custom attributes/resources.

Health checks, commonly used in Kubernetes deployments, are also provided to ensure each Apollo Router instance is ready before Kubernetes sends traffic to that instance.

Supergraph Runtime Extensibility

Apollo Router provides a well-defined extensibility model to hook into the relevant portions of the API request lifecycle, with full access to the GraphQL query and supergraph schema. Rhai scripting, similar to Lua scripting in NGINX and Envoy, enables lightweight in-memory manipulation of headers, cookies, and request context. Co-processors for Apollo Router allow an HTTP sidecar, written in any programming language, to hook into the request lifecycle to support more advanced and bespoke integrations.

Domain-Driven Microservices

Apollo Router makes requests to domain services that live at the microservice layer. At this stage, the subgraph service has transformed the original GraphQL request into a domain service-specific request, which may mean REST, gRPC, or even direct database queries.

API Gateway at the Mesh Edge

These requests pass through Kong Gateway as either Layer 4 or Layer 7 traffic at the mesh edge. As before, Kong's flexible routing engine checks the incoming requests for matching definitions, and any configured plugins are executed before the request is forwarded to the upstream destination.

Service Discovery / Inter-Service Connectivity

The upstream domain service may require coordination among multiple microservices or database objects to completely fulfill the request. Keeping track of a service's dependencies is hard in a microservices world, but it can be solved using Service Discovery. Kong Mesh ships with a DNS resolver to provide service naming — a mapping of hostname to virtual IPs of services registered in Kong Mesh. This allows services to communicate using simple DNS names, greatly reducing application-level code and configuration complexity.

Zero-Trust Security

Instead of relying on traditional security methods that grant access based on network location (e.g., inside or outside of a corporate network), zero-trust security operates on the assumption that threats could be anywhere, and as such, no device or service should be automatically trusted, irrespective of where they connect from.

In Kong Mesh, zero-trust security is inherently integrated by using mTLS to encrypt the traffic between services and authenticate the services to each other. By doing so, even if malicious entities gain access to the network, they cannot readily interpret or tamper with the traffic, nor can they pretend to be a valid service without the proper credentials.

Furthermore, the mesh can provide fine-grained control over which services are allowed to communicate with each other via Traffic Permissions, further tightening the security stance.

Traffic Management and Observability

Kong Mesh handles a variety of traffic reliability features by applying declarative policies to data plane proxies. Common traffic policies include health checks, retries, circuit breaking, and more. Observability details are tracked at each stage of the mesh, forwarding details to collection services and configured via policies applied to the data plane.

Once the request is fulfilled, a response is returned via the gateway to the subgraph where it is assembled into the full response for the original GraphQL request to be returned to the client.

A Modern API Platform

Kong and Apollo technologies are complementary, and when used in conjunction, they provide everything you need to power a modern API platform. Both companies have built best-in-class SaaS solutions that power the unified API platform.

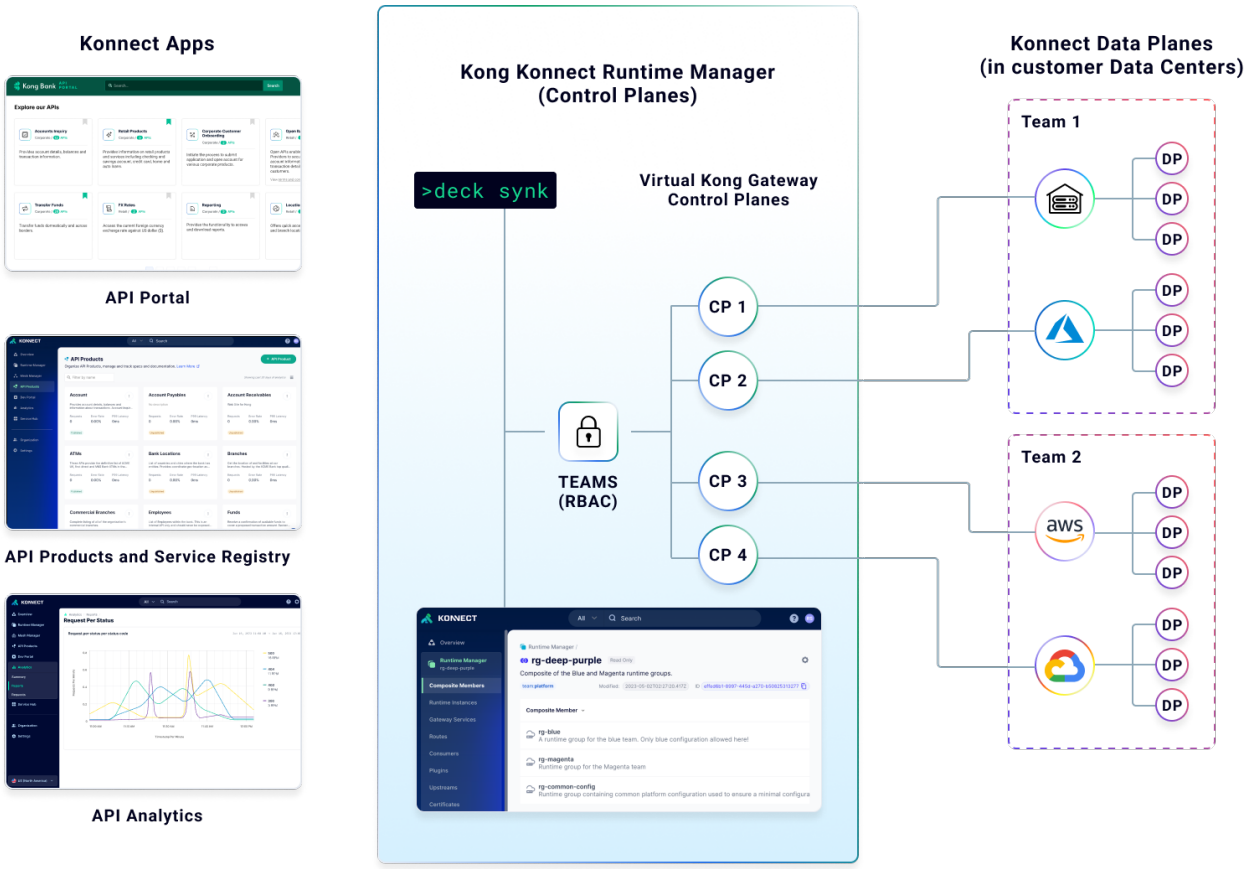
Kong Konnect

Kong Konnect provides a unified API management platform designed to decrease operational complexity and enable scaled federated governance across multiple teams. Kong Konnect enables the management of Kong Gateway, Kong Ingress Controller, and Kong Mesh, offering a single management interface across all Kong runtime technologies.

Kong Konnect is designed to deliver comprehensive API lifecycle management, ensuring adaptability across various clouds, teams, protocols, and architectural designs. It encompasses API configuration, API portals, service catalogs, and deep API analytics functionalities.

Kong Konnect empowers organizations to build and operate the API gateway and domain-driven microservice layers:

1. Deliver a global API registry — Kong Konnect's Service Hub ensures every service, regardless of technology, is cataloged and searchable, creating a single source of truth across the organization.
2. Deliver comprehensive API portals — Developers can navigate APIs, obtain detailed reference documentation, experiment with endpoints, and register applications to consume APIs — all through a single, customizable API portal.
3. Real-time monitoring and analytics — Kong Konnect provides instantaneous access to vital statistics, monitoring tools, and pattern recognition, allowing businesses to gauge the performance of their APIs and gateways in real time.
4. Employ modern operational methods — Kong Konnect enables a Kubernetes-centric operational process with the Kong Ingress Controller integration. Declarative configuration and Kong Konnect management APIs enable a DevOps-ready, config-driven API management layer.
5. Leverage an ecosystem of plugins — Kong Konnect is enabled with an extensive catalog of both community and enterprise plugins. These plugins introduce vital functionalities such as authentication, authorization, rate limiting, and caching — saving critical API developer time and resources.

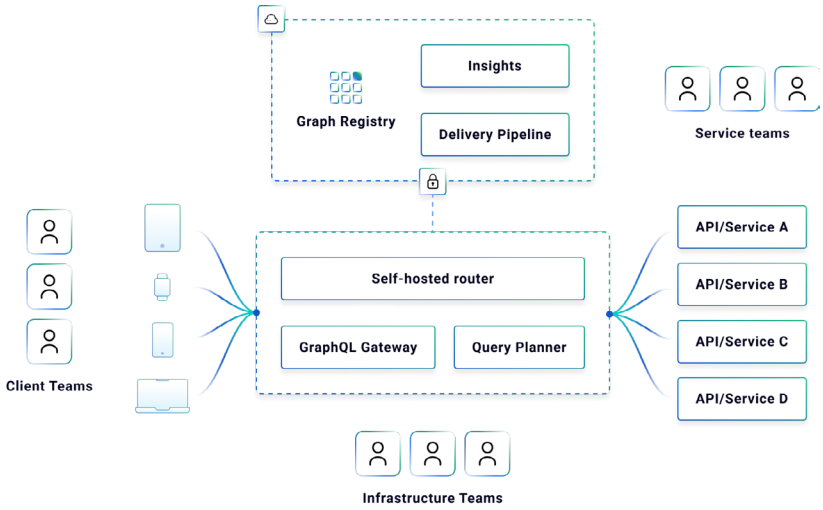

Konnect Apps
API Portal
API Products and Service Registry
API Analytics
Kong Konnect Runtime Manager
(Control Planes)
>deck synk
Virtual Kong Gateway
Control Planes
CP 1
CP 2
CP 3
CP 4
TEAMS
(RBAC)
KONNECT
rg-deep-purple
Konnect Data Planes
(in customer Data Centers)
Team 1
Team 2
DP
aws

Apollo GraphOS

Apollo GraphOS provides the architecture, infrastructure, and workflows to ship a self-service GraphQL platform. This platform provides an intuitive service access layer for client teams, enabling them to ship features faster and deliver better application performance — regardless of how many underlying services they use.

GraphOS enables organizations to build and operate the supergraph layer on top of existing APIs using:

1. Modular graph development — Monoliths cause bottlenecks that slow down app development at every scale. With GraphOS, you build your graph on a modular, scalable architecture with subgraphs that link to each other.
2. Fast, unified query execution — GraphOS links your subgraphs together into the supergraph with a blazing-fast, cloud-native runtime. Access all underlying capabilities with a single GraphQL query and get automatic support for advanced GraphQL features like @defer.
3. Safe and rapid graph evolution — Modern apps change by the hour, and your API architecture needs to do the same. GraphOS gives you the tools to develop schemas collaboratively with a single source of truth, deliver changes safely with graph CI/CD, and improve performance with field and operation-level observability.
4. Graph-native security, performance, and governance — GraphOS supports build and runtime policies that can be defined by the appropriate team and enforced by GraphOS at build-time in the GraphOS CI/CD pipeline and at runtime by Apollo Router. Distributed policy ownership and centralized policy enforcement points are key to scaling your graph efficiently, so each team can own their slice of the graph and deploy autonomously with speed and safety.
5. Advanced runtime options for enterprises — To meet the most demanding enterprise requirements, GraphOS offers a flexible runtime deployment model to give enterprise architecture and operations teams maximum control.



In summary, Apollo GraphOS is engineered to meet the challenges of modern application development head-on, offering scalability, speed, and security while supporting collaborative and autonomous team workflows.

Conclusion

This paper has outlined a robust approach for end-to-end API and GraphQL lifecycle management. Combining Kong's strengths in API management with Apollo's expertise in GraphQL, the reference architecture outlined in this paper provides organizations with a well-rounded, effective solution. The practical guidance and best practices laid out in this paper set a clear standard for what effective API and GraphQL management should look like.

Powering the API world

Konghq.com

Kong Inc.
contact@konghq.com

77 Geary Street, Suite 630
San Francisco, CA 94108
USA