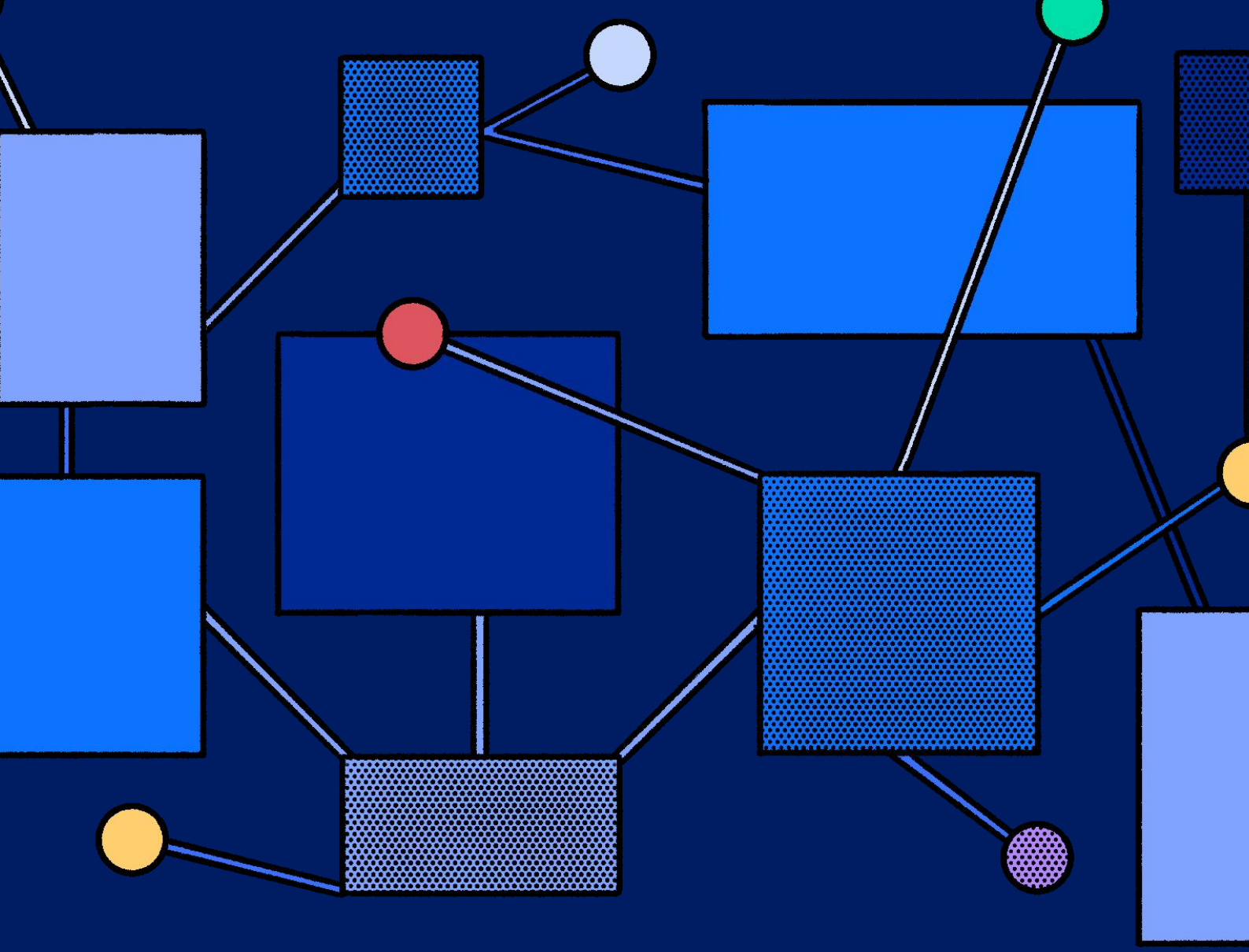
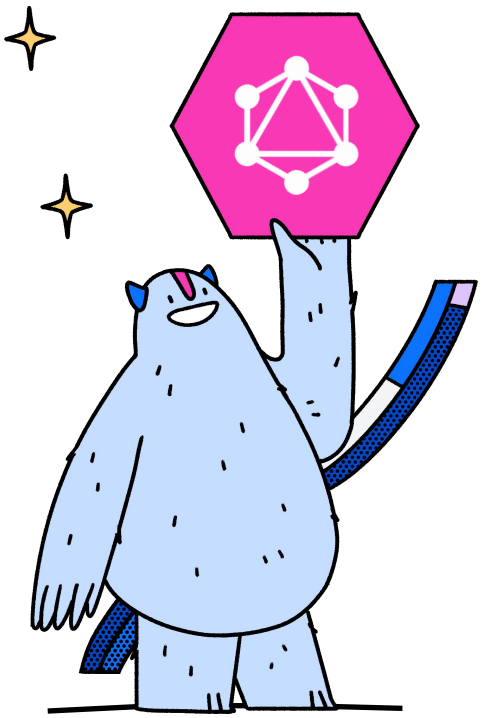


2024 Edition

The GraphQL Handbook

Design patterns and adoption strategies for API architects

HASURA



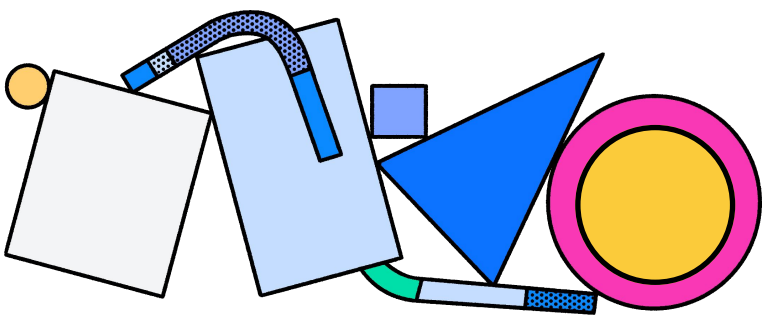
Introduction

This is a handbook on GraphQL for developers and architects. Whether you're a veteran or curious about GraphQL, this primer will help you plan and/or adapt your GraphQL adoption journey. To better understand your journey, it's important to understand the past, the present, and the potential future – how GraphQL came to be, how it's evolved since its inception, and the diverse array of tools and methodologies at your disposal to craft your path based on trade-offs exclusive to you or your team.

How to contribute

We've created a GitHub repository to promote discussions in the GraphQL ecosystem and to allow everyone to contribute to this report. Please use this repository to create issues, discussions, and to submit PR contributions.

Contents



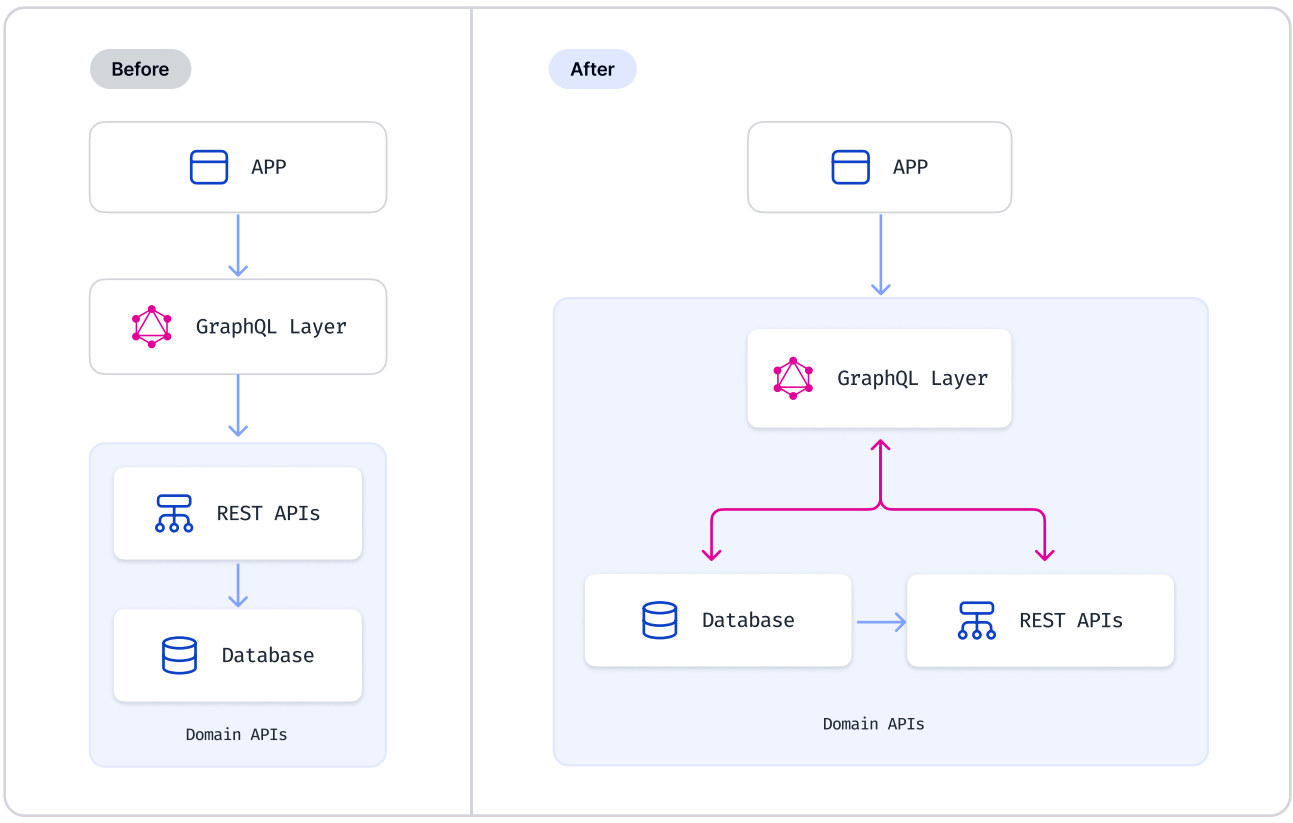
A brief history of GraphQL

The rise of GraphQL mirrors its growth and adoption in engineering teams of different shapes and sizes. Rapid innovation in frontend technologies like React, and their ability to express their data requirements in a structured, hierarchical, and async manner led to the development of GraphQL as a type-system-based contract layer built over existing backend domains to service these requirements.

As we close in on a decade of GraphQL, at a high level, here's four ways the GraphQL ecosystem is evolving →

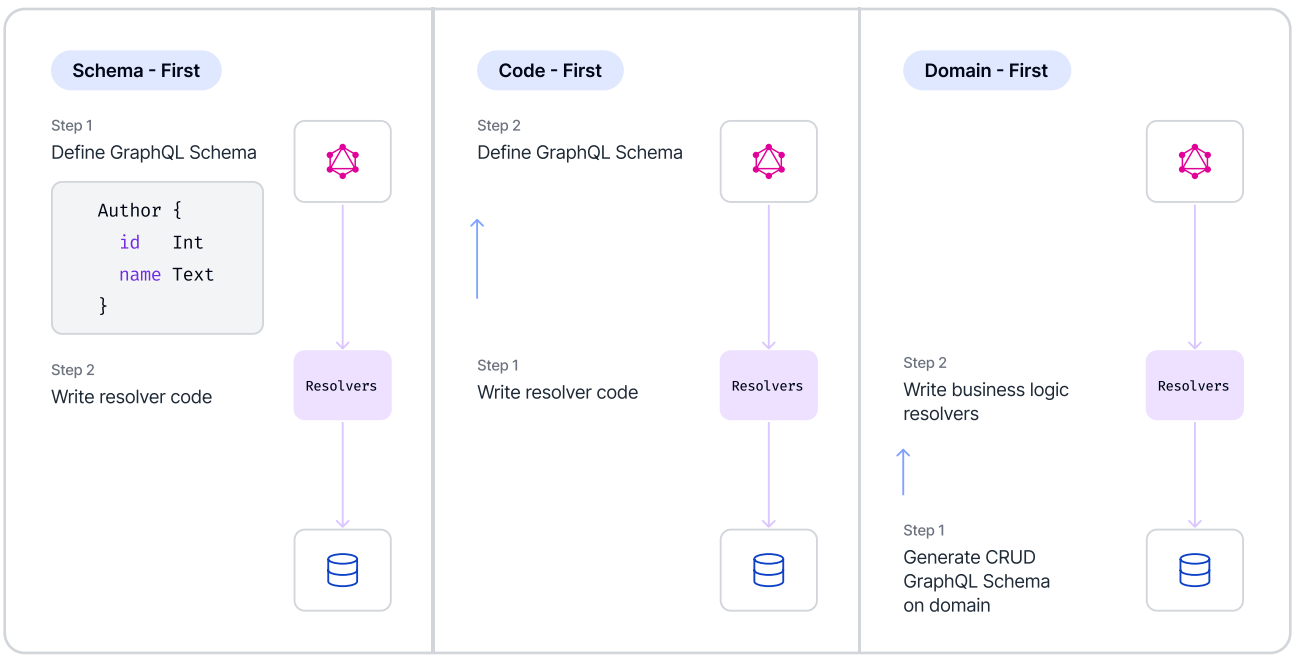
1. Back to the backend

Fueled by the success of a self-serve API layer offered by GraphQL, product requirements typically find their way back to the backend or the domain data itself. As the need for new data increases in almost every GraphQL adoption journey, GraphQL has become more of a backend engineering concern, especially with performance being a critical requirement.



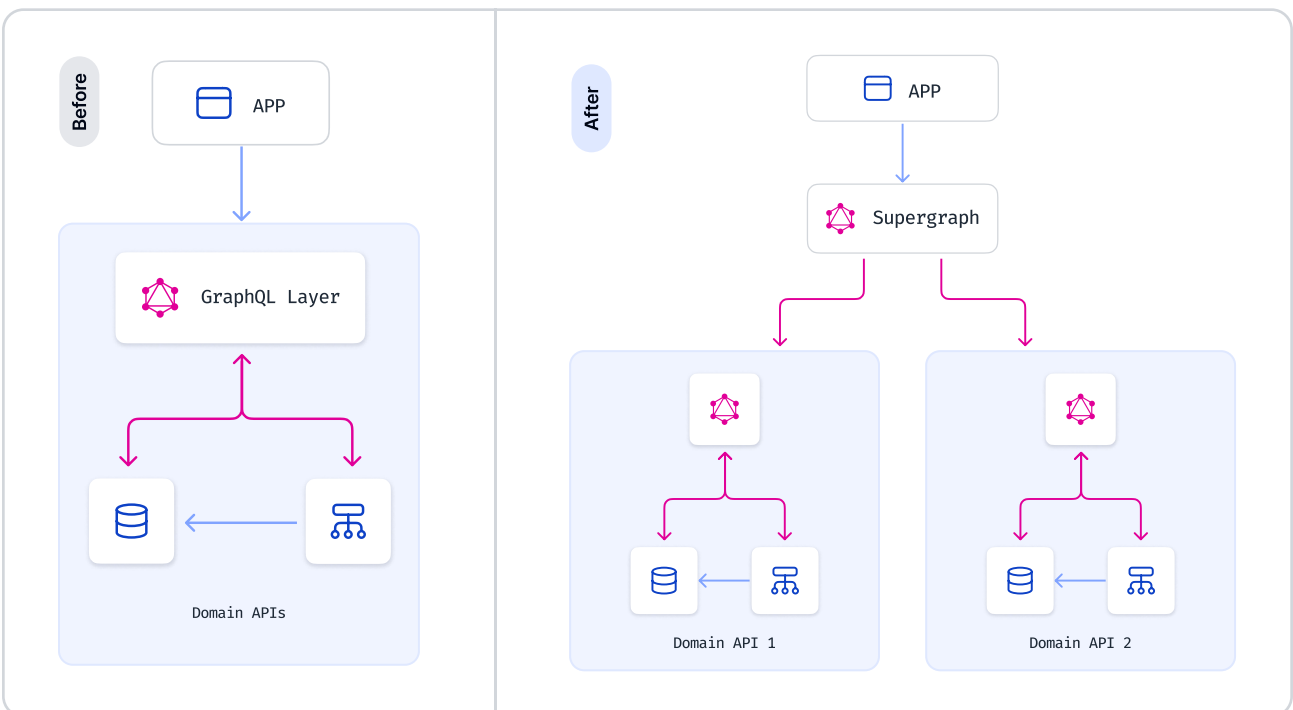
2. Diverse subgraph tooling

To aid engineers working on the implementation journeys of revenue and time-critical projects, domain-first and code-first subgraph tools emerged, making data composition over a single database or data source significantly more efficient.



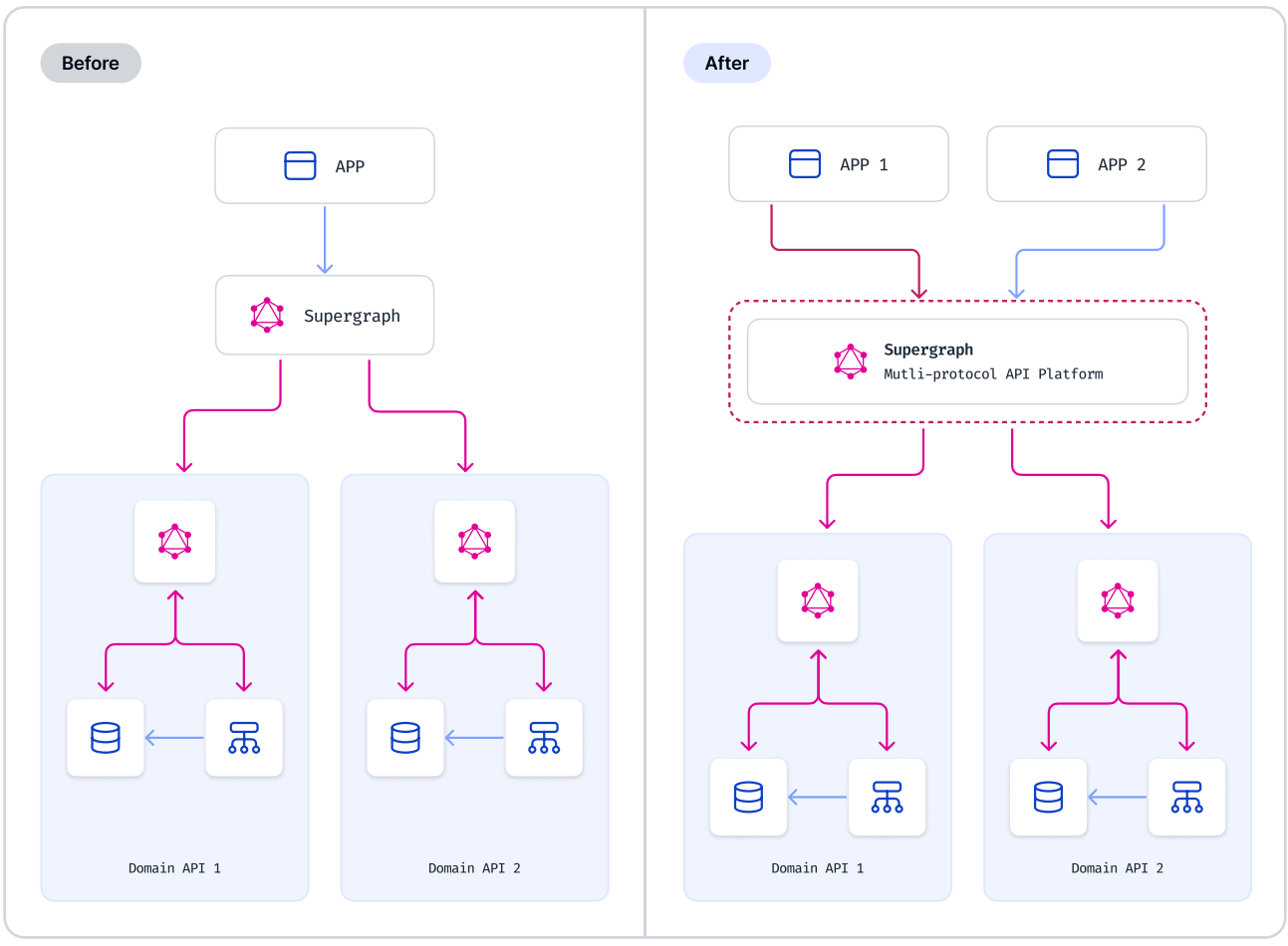
3. Composing a graph of graphs

Building on the success in a single domain, as GraphQL became more prolific, centralization and aggregation requirements led to tools adapting to provide more gateway- or IPaaS-like features. And this resulted in a self-serve composable API.

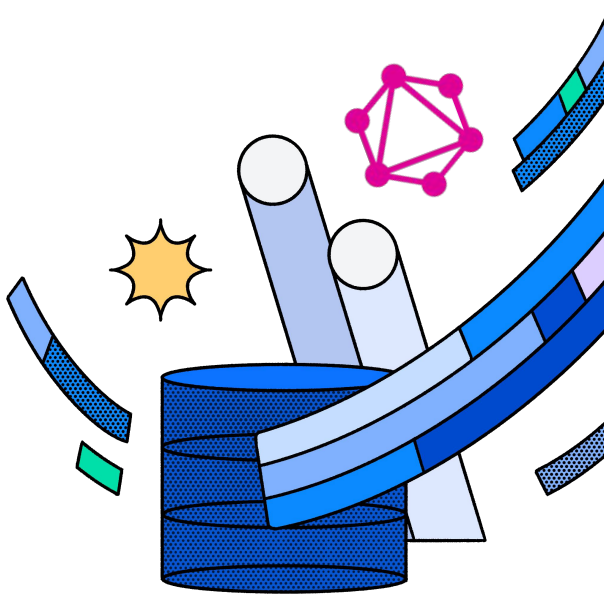


4. Lingua franca for API platforms

The one invariant in its entire journey has been the fact that GraphQL is the first (increasingly) commonly adopted language specification for types backing HTTP APIs (GraphQL trumps other contenders like OData.), regardless of the source of these types. With the power of composition (when supported) and gateway-like centralization-friendly features, GraphQL is fast becoming the substrate for building multi-team, domain-spanning API platforms.



Nine years of GraphQL



2012 The origins of GraphQL at facebook

2015 Spec is open sourced at React.js conf

2016 Apollo Server (JS), Apollo Client and Relay are released

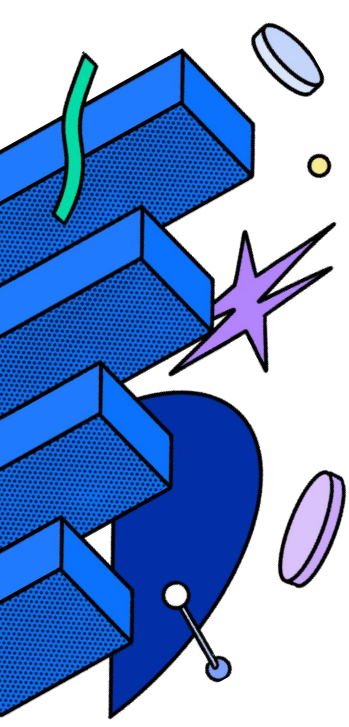
2017 First GraphQL user conference

Walmart, IBM, and The New York Times adopt GraphQL

Postgraphile debuts compiler-style domain-driven GraphQL API tooling

Schema-stitching feature is released as part of GraphQL-tools

GraphQL subscriptions is added to the specification



2018 GraphQL tools like urql, graphql-request, GraphCMS are released

Hasura open-sources GraphQL Engine, another compiler-style domain-driven GraphQL API tool.

2019 GraphQL Foundation is formed under the Linux Foundation umbrella

Apollo Federation v1 released

2020 Hasura Data Federation and GraphQL Mesh are released

5 years of GraphQL 🎉 [Bonus 📽️ video]

2021 Current stable version of GraphQL spec released

Apollo Federation v2 released

2022 GraphQL Gateway tools like Wundergraph, StepZen & Grafbase released

🐣 GraphQL API management platforms like Stellate and Inigo are release

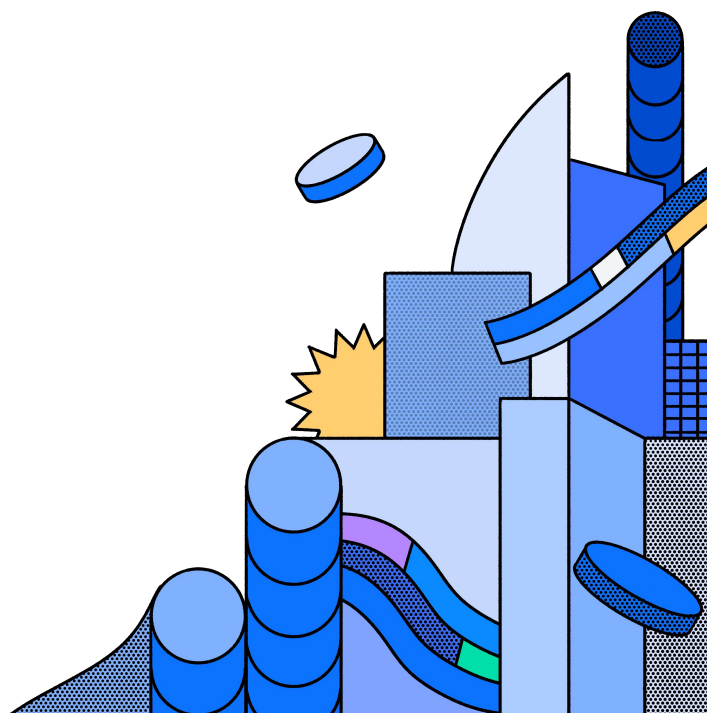
2023 Emergence of Supergraph and Data Federation.

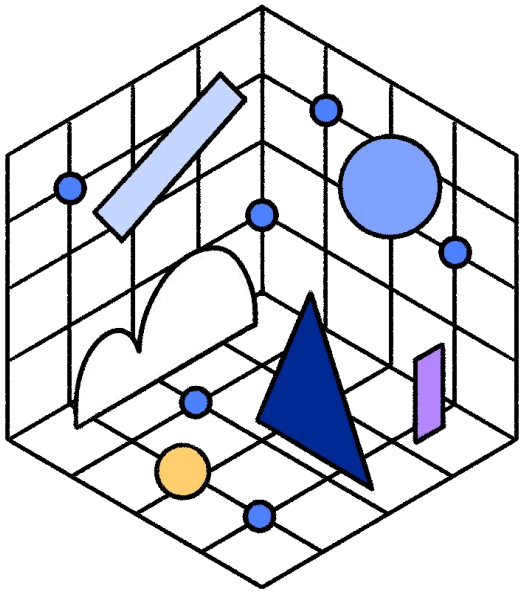
GraphQL Fusion, a distributed GraphQL gateway, is released

GraphQL Foundation announces working groups with expanded scope

2024 GraphQL Foundation starts composite schemas working group

Hasura DDN combines GraphQL with edge infrastructure.





GraphQL specification and foundation

Established in November 2018, in partnership with the Linux Foundation, the GraphQL Foundation aims to advance GraphQL by fostering collaboration among ecosystem stakeholders such as GraphQL adopters, and tooling developers like Hasura, The Guild, Apollo, and others.

By providing educational resources, promoting best practices, and supporting open source tools and standards, the foundation ensures the sustainable growth and widespread adoption of GraphQL. It hosts various working groups dedicated to specific areas such as tooling, specifications, and ecosystem development, enabling stakeholders to collaborate on improving interoperability and driving innovation within the GraphQL ecosystem.

GraphQL.org ⟶

Working Groups ⟶

Latest stable (Oct'21)
GraphQL Spec ⟶

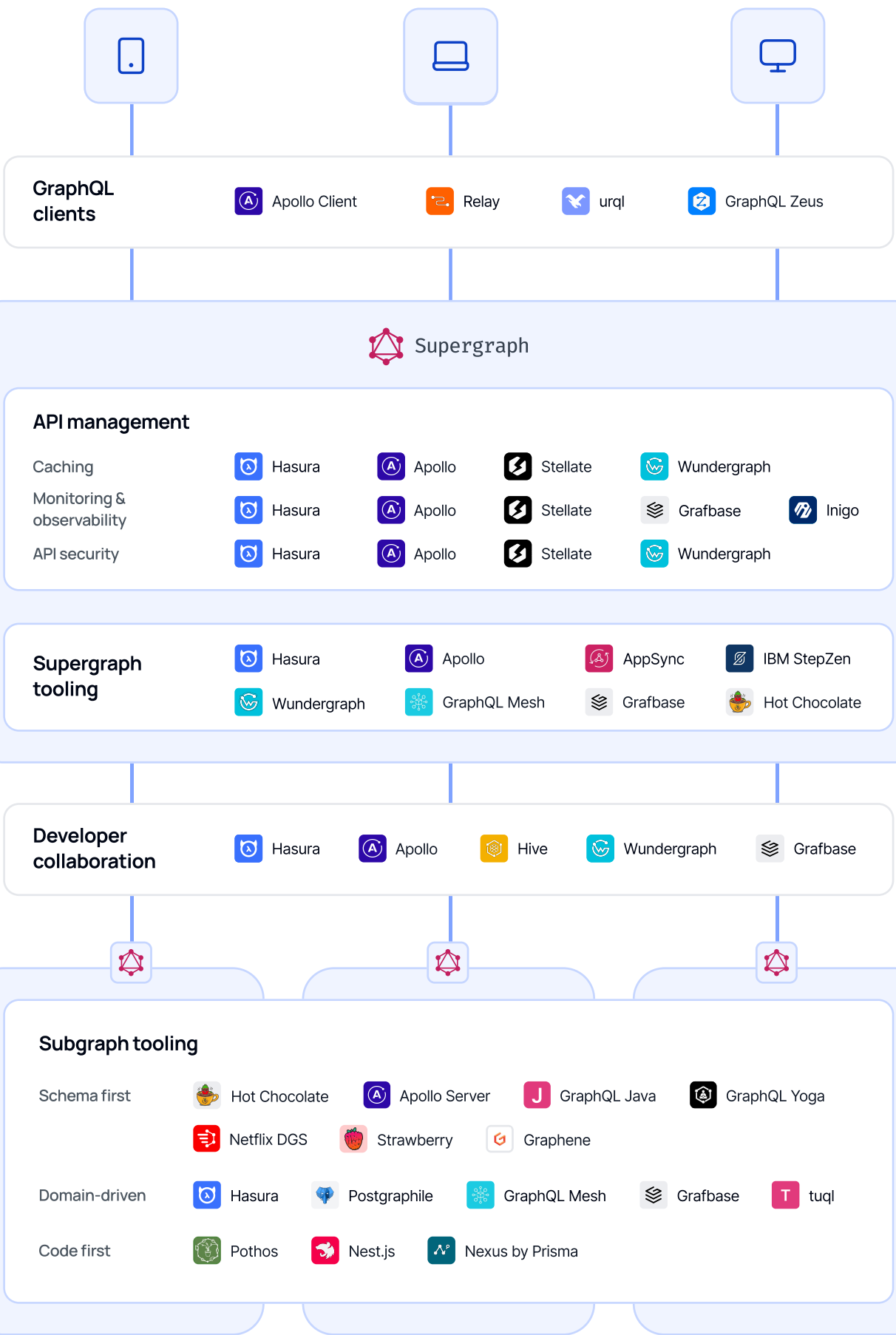
Current working draft
GraphQL Spec ⟶

The GraphQL landscape

Like other recent Linux Foundation projects (think containers and Kubernetes), if you can't already tell from the timeline in the first section of this handbook, the GraphQL ecosystem has been a hotbed of ideas and innovation. As an API query language specification, the GraphQL specification doesn't dictate how a GraphQL server should be implemented or even how it should behave outside of some core aspects (introspection, validation, error handling, etc.). This loose definition has resulted in unbridled experimentation with architectures and adoption styles based on the requirements of a diverse set of early adopters.

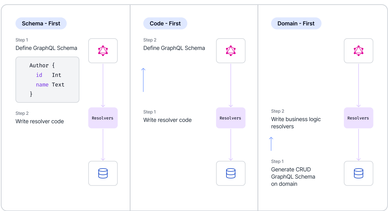
Therefore, for the first time, we present a new taxonomy to understand the various tools and projects in the GraphQL ecosystem to better equip GraphQL practitioners with a mental model that helps them choose the right tool for the right job.

There is no wrong or right approach, there are only trade-offs that one must be ready to make depending on their unique conditions. It's also important to know that you can pick more than one approach or tool for different parts of your application or API platform.



Subgraph tooling

A GraphQL server requires two core components: a schema and some way to resolve this schema. There are three approaches to creating a GraphQL server and exposing a GraphQL API from scratch, depending on whether you start with the schema or resolver or generate them from your existing domain.



Schema-first approach

In the schema-first approach to GraphQL API design, we begin by explicitly defining the GraphQL schema, which serves as a blueprint for the API. This schema is written in the GraphQL Schema Definition Language (SDL) and outlines the types, queries, mutations, and subscriptions that the API will support. It acts as a contract between frontend and backend developers, ensuring clear communication and understanding of the API's capabilities.

Once this SDL is available, developers need to implement this contract by writing resolvers (code). In turn these resolvers interact with existing code and/or data sources (DBs, third-party APIs, etc.). The amount of resolver code and its complexity (and therefore the time to deliver it) are directly proportional to the quality of the API and the density of your data sources' schema. For example:

- Each entity join (like posts with comments – see below) in the same or different database needs glue code. The more joins you want, the more glue code you need with this approach.

- Consider the following query →

```
query {
  posts (limit: 10, offset: 20, order_by: "id desc", where: {comments:
{upvotes: {gt: 0}}}) {
    id
    title
  }
```

If you want a rich API that supports operators (like limit, where clauses, etc.), you'll need to write resolver code – and lots of it. Since these tools must interact with resolver code to generate a GraphQL server, schema-first tools are language-specific.

Schema-first tools	Supported
Apollo Server	JS
GraphQL Yoga	JS
Netflix DGS	Java
GraphQL Java	Java
Hot Chocolate	.Net
Graphene	Python

Code-first approach

In this approach, developers focus first on writing resolvers, often in a type-safe language, with the schema generated automatically, ensuring consistency between the schema and the resolvers. Once again, the interaction of the GraphQL schema with the resolver code results in these tools being language specific as well.

Schema-first tools	Supported
Nexus by Prisma	JS/TS
Pothos	JS/TS
Nest.js Code First	TS
Graphql-Kotlin	Java/Kotlin

Domain-driven approach

This approach autogenerates both the schema and resolvers by introspecting the underlying domain i.e. the data sources (to generate CRUD resolvers) and any existing code (similar to code-first tools, to reuse any business logic), significantly reducing the need for boilerplate code and accelerating subgraph development.

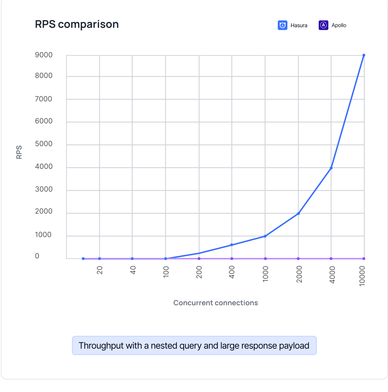
These tools are often language-agnostic and can work alongside most languages and frameworks. However, given that these tools need to introspect databases, they have varying degrees of support for different databases.

Domain-driven tools	Description	Supported languages
Hasura	Provides a powerful platform to instantly create a GraphQL backend over most popular databases, with real-time capabilities and a strong focus on performance.	PostgreSQL, MongoDB, MySQL, SQL Server, Oracle, Snowflake, etc. See the complete list.
Postgraphile	Specializes in PostgreSQL databases, offering a quick setup for a GraphQL API layer.	PostgreSQL
AWS AppSync	GraphQL API platform, specialized in DynamoDB and custom logic with serverless functions.	DynamoDB, RDS/Aurora Postgres/MySQL
tuql	Focuses on SQLite, useful for smaller projects or lightweight applications.	SQLite

The distinction between domain-driven and resolver-based tooling represents a fundamental architectural choice in GraphQL server implementation, affecting performance, flexibility, and scalability.

- **Performance**: Here's an example comparison of the throughput from subgraphs built using schema-first (apollo-c architectures).

- **Flexibility:** If the quality of the domain-driven APIs is not high (i.e. they are not expressive or forward-looking enough to keep up with demand a few months/quarters down the line). More on this in the section on GraphQL maturity phases.



Throughput with a nested query and large response payload

Comparing different styles: Challenges / limitations

Approach	Pros	Cons	Use case(s)
Schema-first	• Theoretically, the most flexible approach • Handwritten schemas can have expressive documentation • Easier to perform API mocking with data	• Performance limitations for DB queries, since queries are resolved and not compiled • Hard to refactor code rapidly • Prone to runtime errors	• When you are adding GraphQL on top of existing REST APIs which are un-typed and cannot be inferred.
Code-first	• Skip writing the GraphQL SDL, auto-infer types leading to less duplication work • Easy to refactor code • Faster to prototype	• Autogenerated documentation from inferred schema not expressive • Heavily dependent on framework tooling for the language of choice	• Very similar to schema-first use cases • Large refactoring and iteration
Domain-first	• Skip writing GraphQL SDL *and* resolvers • High-performance APIs • Highly composable APIs within entities of the same domain	• Tied to the database(s) • Some limitations with schema extensibility for autogenerated APIs	• GraphQL on databases, gRPC and REST APIs with OpenAPI spec

Supergraph and federated gateways

A supergraph represents a sophisticated architectural and operational framework designed to amalgamate and scale various data domains into a unified, composite graph of entities and operations. This modern approach merges the advantages of a centralized, monolithic system – characterized by high cohesion and streamlined governance – with the flexibility and scalability offered by a federated microservices execution model, which emphasizes loose coupling and distributed ownership. It's the best of both worlds.

In the contemporary landscape, where rapid expansion of data volumes and microservice architectures significantly complicates the consumption of data and APIs, the relevance of supergraphs has escalated. This complexity manifests in extended time-to-market durations, the increased challenge of managing technical debt, and the intricacies of inter-team communication.

As such, supergraphs stand at the forefront of solutions, addressing these challenges by simplifying integration, enhancing operational efficiency, and fostering a more coherent technological ecosystem.

Key components of a supergraph:

- **Unified semantic layer**: A unified semantic layer that describes entities, operations, and relationships spanning one or more data domains.
- **Composable API:** Compose data within entities of the same source or across multiple sources.
- **Federated SDLC and governance:** The ability for multiple teams to collaborate on evolving the supergraph's unified semantic layer and the API schema, with no centralized ownership or single point of operational failure.
- **Query planning and distributed execution:** A standardized API schema to access entities and invoke operations, as well as traverse and compose along relationships – typically GraphQL.

Supergraph and GraphQL gateways	Description
Hasura	Supergraph that works with any subgraphs, irrespective of language/framework
Apollo Router	Federation with Apollo Federation compatible subgraphs
GraphQL Mesh	Federation of data sources with type merging supported
AWS AppSync Merged APIs	Supports only AppSync-based subgraphs for federation
Wundergraph Cosmo	Works with any subgraph
IBM StepZen	Federate using @materializer directive and supports Apollo Federation v2 spec
GraphQL Fusion	Open Federation Spec for distributed GraphQL
Grafbase	Simple composing of schemas for federation

API management

Caching

Implementing a caching layer can significantly enhance API performance by reducing redundant data source requests. Traditional REST API caching strategies do not map directly to GraphQL, necessitating a more nuanced approach. GraphQL operates with a single endpoint, mostly with a HTTP POST request and GraphQL doesn't conform to regular HTTP status codes for responses. This forces a caching strategy that requires tinkering with query responses.

Automatic Persisted Queries (APQ) can also be utilized for caching purposes. APQ is a protocol that reduces the overhead of repeatedly sending the same GraphQL documents to the server by sending a hash of the query instead of the full query text. This approach significantly reduces network traffic and payload size, which can enhance performance, especially for large GraphQL documents.

By using APQ, clients can send only the query hash to the server, and if the hash is not found on the server, the client can make a second request to register the query hash with the original query. This mechanism allows for efficient caching of queries on the server side, improving network performance by reducing bandwidth utilization and speeding up GraphQL client loading times.

APQ is particularly effective when clients send queries as GET requests, enabling them to leverage browser caching and integrate with Content Delivery Networks (CDNs). Overall, APQs serve as a valuable tool for optimizing GraphQL performance through query caching.

Hasura, Stellate, and custom solutions like Apollo with Redis offer robust caching solutions tailored to GraphQL's unique data-fetching mechanisms.

Caching services / tools	Description
Hasura	GraphQL query response caching; APQs
Stellate	GraphQL edge caching
Apollo GraphOS	DIY setup with Redis; APQs
Grafbase	GraphQL edge caching

API security

GraphQL diverges from REST by centralizing its interactions through a single endpoint rather than using multiple URLs, challenging traditional URL-based filtering methods. This model simplifies requests to primarily POST, with WebSockets for real-time data, contrasting REST's use of varied request methods.

Such architecture necessitates heightened security measures, as the singular/graphql endpoint can attract attackers, particularly if interactive tools like GraphiQL or an API's introspection query are accessible without proper authentication. Securing the GraphQL endpoint is critical to protect system integrity and data privacy.

- Allow lists: Safely allow a limited number of GraphQL operations that are consumed by the client.
- Rate limiting: Restrict the number of GraphQL operations for a given time duration. Restrict it by the client making the query, the IP address the query originates from or the database value like user_id who is making the request.
- Depth limiting: Restrict a GraphQL operation based on depth of the query.
- Node limiting: Limit based on the number of nodes (field with a selection set)

Traditional API gateways have typically served two primary functions in API architectures: security (authentication, authorization, threat protection) and management (rate limiting, logging, analytics).

In the context of GraphQL, these gateways face unique challenges due to GraphQL's operation model – single endpoint, complex queries, and rate limiting modeled by the query structure. Given these considerations, the role of traditional API gateways in the context of GraphQL security can be seen as evolving to meet these new challenges.

Gateways must adapt to support GraphQL-specific features including fine-grained access control, query complexity analysis with rate lmiting, depth limiting, etc.

API security tools / services	Capabilities*
Hasura	Rate limits, query depth limit, node limit, allow lists, disable introspection, response limit, database timeouts
Apollo Router	Operation limits (depth/height/root fields), disable introspection, rate limit via traffic shaping plugin
GraphQL Mesh	Rate limits, query depth limit, node limit, disable introspection all via plugins
Wundergraph Cosmo	Rate limits, query depth limits, disable introspection

Performance

The performance of GraphQL services is influenced by factors such as query complexity, network overhead, backend performance, and the efficiency of data-fetching strategies. With the evolution of GraphQL tooling and libraries, developers now have access to a comprehensive set of features designed to enhance performance, including intelligent caching mechanisms, query batching, and schema optimization techniques.

Modern GraphQL tooling has significantly advanced in addressing performance bottlenecks commonly associated with N+1 query problems, over-fetching, and under-fetching of data. Libraries like DataLoader have become standard in solving N+1 problems by batching multiple requests to the backend into a single request.

Additionally, advancements in static query analysis and persisted queries have allowed for optimizations at the transport layer, reducing the size of requests and improving the overall efficiency of GraphQL operations.

Domain-driven tooling has gone one step further to improve performance much more for database queries by taking an incoming GraphQL request and compiling it to a highly optimized database query (SQL/NoSQL).

Optimizing for performance

Optimizing GraphQL performance involves a multi-faceted approach, focusing on both server-side and client-side strategies. Below are key factors and strategies for optimizing GraphQL performance:

Performance factors	Improving performance
Query optimization	Compile database queries as opposed to resolving.
Efficient data loading	Implement DataLoader or similar batching and caching mechanisms to avoid the N+1 query problem for non-database queries or cross-subgraph joins.
Monitor GraphQL requests	Identify inefficient, slow queries with GraphQL specific request metrics, like query response times, database timeouts, etc.
Caching	Intelligent caching on the server and client. Server-side caching using query response caching at the edge or use Automatic Persisted Queries. Use client libraries to reuse previously fetched data with unique IDs.
Rate limits	Prevent abuse by implementing rate limiting and query complexity analysis.

Observability and Monitoring

Observability refers to the ability to introspect and understand the internal state of the system by observing metrics from outside. In the realm of GraphQL, as applications move to production, it is important to capture insights about the GraphQL API's performance, health, and usage metrics.

- Get visibility into API availability
- Gain insights on API performance
- Run a health check for data sources

Applying the REST philosophy of measuring performance of an HTTP endpoint for GraphQL will quickly fail, since GraphQL has only one endpoint. Measuring GraphQL API performance will require a deeper look into the actual query, the level of depth involved in the query.

In a GraphQL application, these are the essential metrics and context to capture:

- **Real time and scalable**
  Time of query and query execution time
- **Response metrics**
  Logging the response status codes of queries, mutations, and subscriptions
- **System information**
  GraphQL server version and originating IP address of the client
- **Query details**
  Actual query payload / query hash / unique identifier

Monitoring and observability tools

- Hasura
- Apollo GraphOS
- Stellate
- Inigo
- Wundergraph

Identifying and addressing slow queries is crucial for maintaining system efficiency. For example, in a domain-driven approach to GraphQL, one can spot high-latency queries and employ database optimization strategies such as `EXPLAIN/ANALYZE` commands.

Tracking errors

- Check and filter response with errors
- Leverage dev mode for debugging
- Enhance logging levels to improve error diagnosis

Together, observability and monitoring enable teams to transition from reactive troubleshooting to a more predictive and preventative maintenance strategy, ensuring that applications remain performant, reliable, and responsive to the needs of end-users. While there are challenges in monitoring GraphQL, existing tooling in the ecosystem addresses the concerns.

Developer collaboration tools

GraphQL allows multiple teams to work effectively by defining a shared schema that represents the entire data model of an application that can span multiple domain teams. However, managing a shared schema can be challenging as the system evolves.

Collaboration tools like schema registries address this by providing centralized repositories for storing and/or validating unified GraphQL schemas. These tools enable teams to collaborate on schema changes, track system evolution, and enforce schema integrity (via regression tests on production traffic, for example). By leveraging schema registries, teams can iterate rapidly, deliver high-quality unified GraphQL schemas, and focus on innovation without integration challenges or API design inconsistencies.

GraphQL maturity model

This GraphQL maturity model aims to enable GraphQL practitioners or anyone curious about GraphQL by helping them anticipate the challenges of all stages of a GraphQL adoption journey, and use these insights from others' experience to fine-tune their current efforts and be better prepared.

Caveats

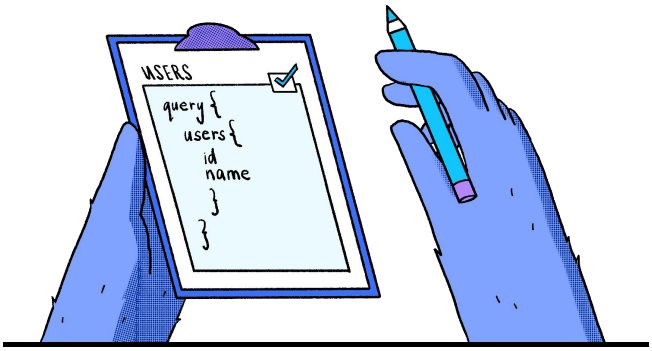
- **Iterations and nonlinearity:** Adoption journeys are not linear. Most organizations will go through multiple iterations of building GraphQL APIs/layers. So it's possible, for example, to go from stage 4 to stage 1 as you pick up the task of building a new subgraph.

- **Org structure and multiple stages:** More than one independent or dependent GraphQL journey could be playing out in an organization, sometimes even in the same team. So it's possible to belong to different stages at the same time.

- **Measure of complexity, not competence:** It's normal to not need anything more than a single simple GraphQL API on a domain database and staying in stage 1 for years. A position in the maturity model is more indicative of the data-access patterns that need to be supported, and less about the ability of teams to support more elaborate patterns.

Stages of maturity

Stage	Description	Sample query
Single subgraph	A crucial first step in your GraphQL adoption journey is building high-quality production-grade GraphQL subgraphs. Speed and quality of the subgraph affect not just itself but also your federated GraphQL layer. ☼ Invest in GraphQL training and evangelism, especially for API consumers – support them with GraphQL to REST tooling if needed.	`query orders { order { id ordered_on } }`
Multiple subgraphs	Independent work streams to build independent subgraphs. ☼ Learn from the previous stage – use better or different tools if you need to.	`query orders { order { id ordered_on } }`
Multiple subgraphs	Independent work streams to build independent subgraphs. ☼ Learn from the previous stage – use better or different tools if you need to.	`query products { product { id name } }`
Schema stitching	Unify independent schemas to make API consumption easier. Schema stitching can accommodate type merging with some input to resolve conflict. ☼ Schema stitching can handle lightweight relationships between types. If you end up doing this, be sure to invest in some collaboration tooling.	`query authors_articles { author { id name } articles { id title } }`
GraphQL federation	The current art is possible in the GraphQL ecosystem to hierarchically integrate multiple subgraphs by allowing relationships to be defined between types sourced from separate subgraphs. ☼ • At this stage, collaboration tools are critical to avoid teams iterating independently. • Performance and composability issues in subgraphs usually surface at this stage – so this is a good time to revisit subgraph design for future iterations.	`query authors_their_articles { author { id name articles { id title } } }`
? or Hasura supergraph	There's a limit to the access patterns that can be supported with GraphQL federation architectures or anything that implements composability at the level of GraphQL subgraphs. ☼ To implement powerful queries, such as the example here, you'll need the help of more advanced subgraph tooling – remember, subgraph quality dictates supergraph quality! See this for more details.	`query { author (where: {article: {title: {_like: "%dune%"}}}) { id name articles { id title } } }`



GraphQL production checklist

When deploying GraphQL subgraphs as part of a supergraph architecture, ensuring that each component is optimized, secure, and ready for production is crucial. This checklist serves as a critical step in preparing for a smooth deployment and ongoing maintenance of a robust, scalable, and secure GraphQL ecosystem.

APIs in production need authentication and authorization, and best practices like some of the ones outlined in the 12 Factor App architecture. But here are some GraphQL-specific nuances that need to be considered while going to production.

Production-ready checklist	Implementation concerns
✓ Disable GraphQL introspection	For public-facing APIs, disable GraphQL introspection, enable limited set of queries via allow listing
✓ Setup rate limiting	Rate limiting both at subgraph and supergraph layer
✓ Setup operation limits	GraphQL node limits, depth limits
✓ Regression testing with schema registry	Configure test suites to avoid breaking changes and track them via schema registry
✓ Tracing, logging, and metrics	Configure more tracing / export logs / setup OpenTelemetry exports

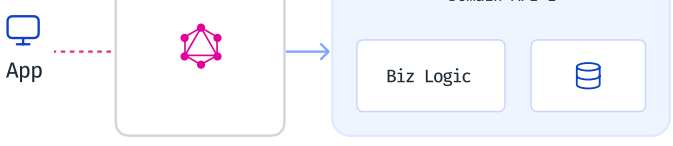
Insights and resources for practitioners

As a GraphQL practitioner, you might need to navigate a typically tricky path to adoption or success, especially when trying to get your team on board. The following set of resources will help you make sense of what GraphQL displaces, or coexists with, how it can be leveraged in large technology initiatives, and how to advocate for its adoption.

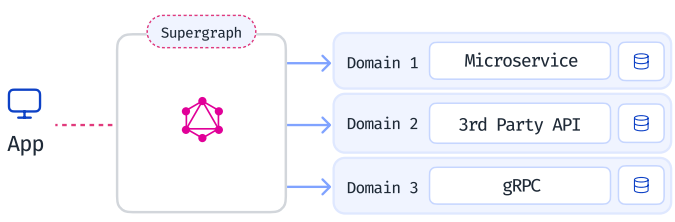
GraphQL architectures

GraphQL is an API technology that can and is leveraged along with other engineering initiatives. Here are some examples of contexts in which enterprises are adopting GraphQL:

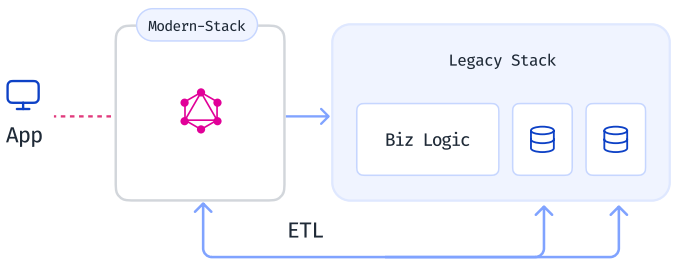
Subgraph



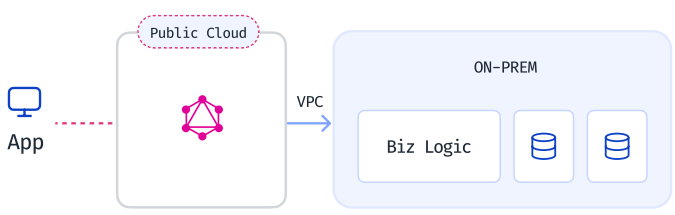
Common data platform



Technology modernization



On-premise to cloud migration



GraphQL vs. REST

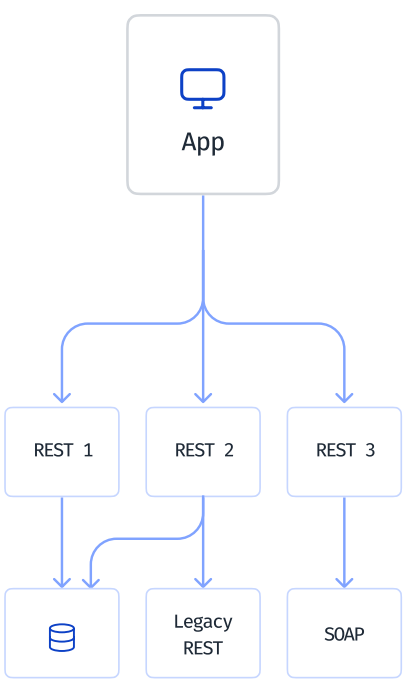
Too often GraphQL is pitted against REST as an emerging pattern for building HTTP APIs. While that may be true in some contexts, as backend developers, you may have to support both GraphQL and REST(or non-GraphQL) APIs in your organization as some consumer teams may not be ready for GraphQL. As a practitioner you may also need to tackle the challenges of migrating from REST to GraphQL.

Migrating from REST to GraphQL

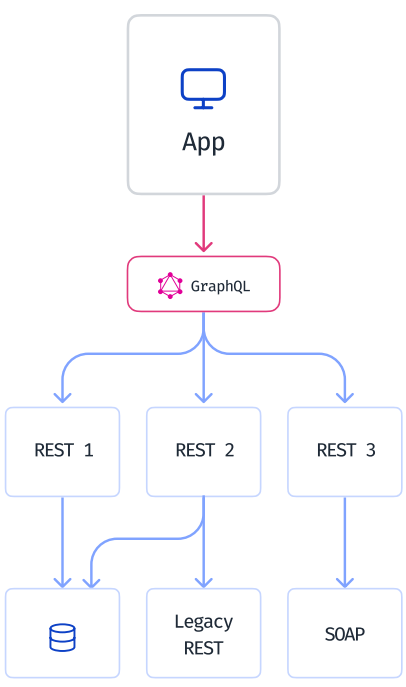
There are several patterns for teams looking to “move from” REST to GraphQL depending on how this cutover can be managed. If your frontend team cannot change existing integration immediately, you may need to maintain both sets of APIs until this migration is complete in the frontend application.

You can also adopt GraphQL iteratively by first building a facade over existing REST APIs and then gradually migrating to REST. Here’s one model depicting this kind of change:

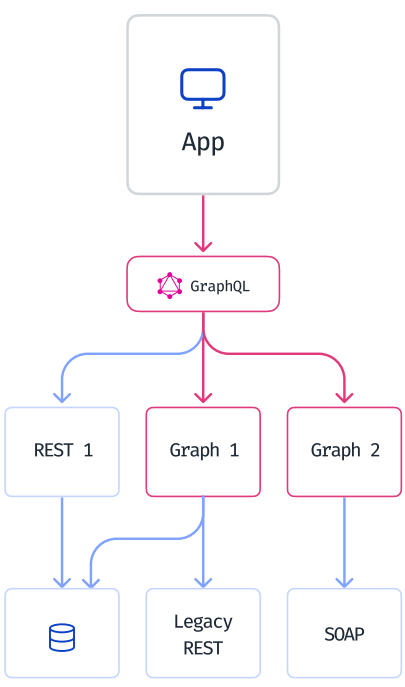
Phase 1



Phase 2



Phase 3



REST tools in the context of GraphQL

If you’re migrating away from REST in a project, and have done so sufficiently, you can also evaluate the option to discard some REST tools that may no longer be necessary. API gateways and integration tools like IPaaSs are the kind of tools that can be scrutinized for their utility in a GraphQL world.

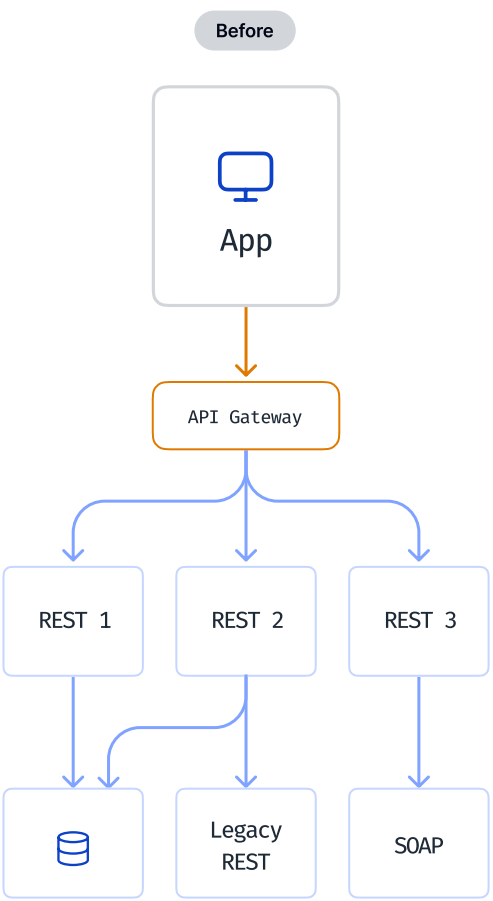
API gateways

API gateways have typically served two functions: Authentication and security, and API management (caching, rate limiting, etc.).

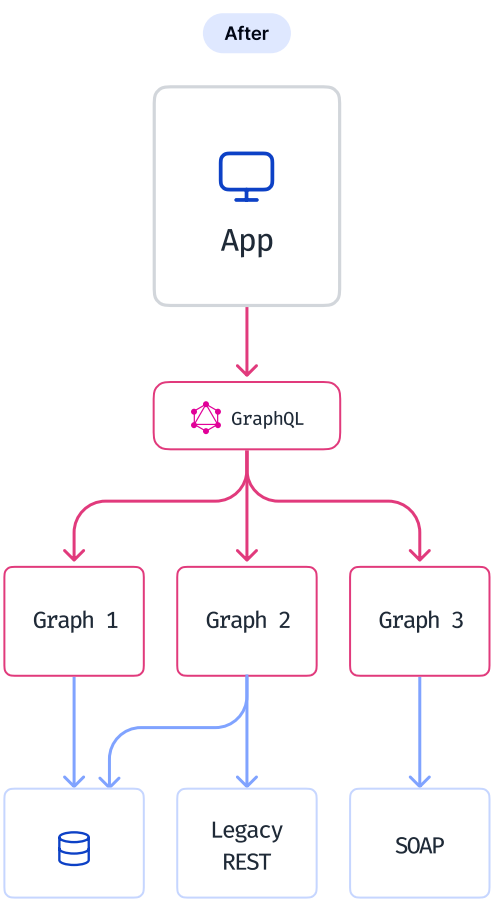
API management tooling is tightly coupled with the API format (i.e. API limiting in the context of GraphQL goes beyond simple rate limits and also needs to handle GraphQL-specific concerns like depth limits to effectively guarantee quality of service.) GraphQL tooling has evolved to handle these use case. Many teams, however, continue to rely on API gateways to centrally handle authentication, mostly due to organizational policies or muscle memory.

Depending on your context, you can either use these tools together or discard REST-based tooling if it’s no longer relevant.

Before



After



API integration platforms or IPaaSs

In the context of REST APIs, integration platforms like Mulesoft, Boomi, etc. solve a very REST problem (i.e. compensating for the lack of inherent aggregation and composition capabilities). However, aggregation and composability are natural byproducts of GraphQL’s type system and design. So unless you use IPaaSs for their gateway-like features (ie. authentication, security, etc.), you can easily replace them with federation tooling.

AI and GraphQL

If you are an application engineering team in 2024, you're either building AI services and products or figuring out how to use AI in your applications or developer workflows. These are early days and the AI ecosystem is in constant flux, but the day is not too far away when it will be imperative to build a natural language interface in your app (or completely rethink the interactivity altogether). Given this, it's not unnatural to want to know if and how GraphQL and AI could play well together.

LLMs, RAGs, and data

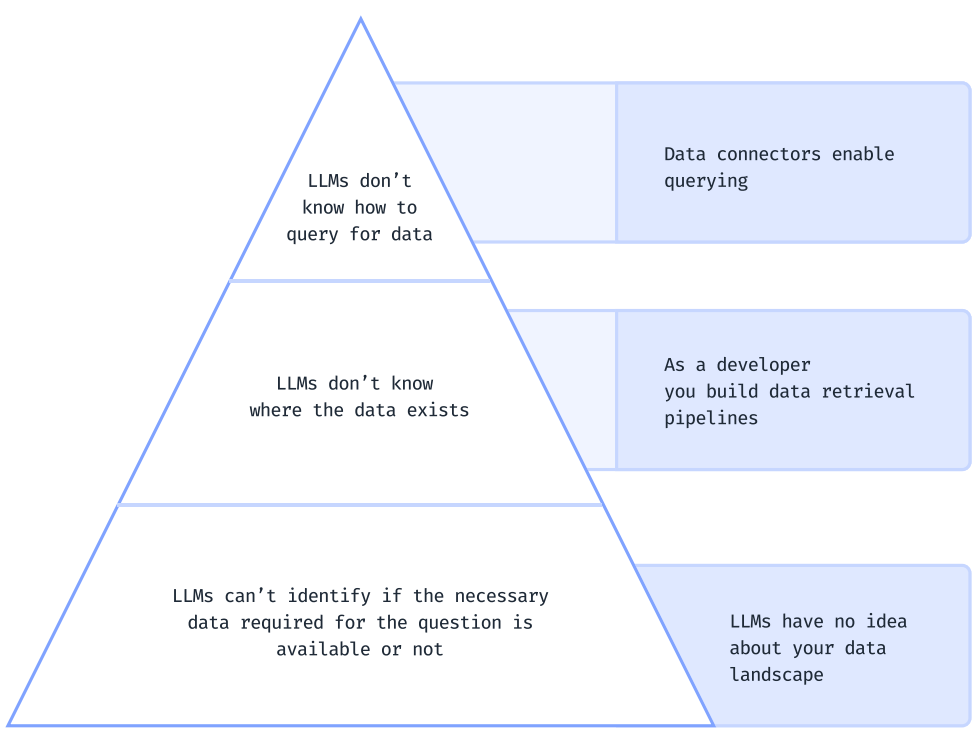
There's plenty of literature on AI fundamentals but here's a quick primer:

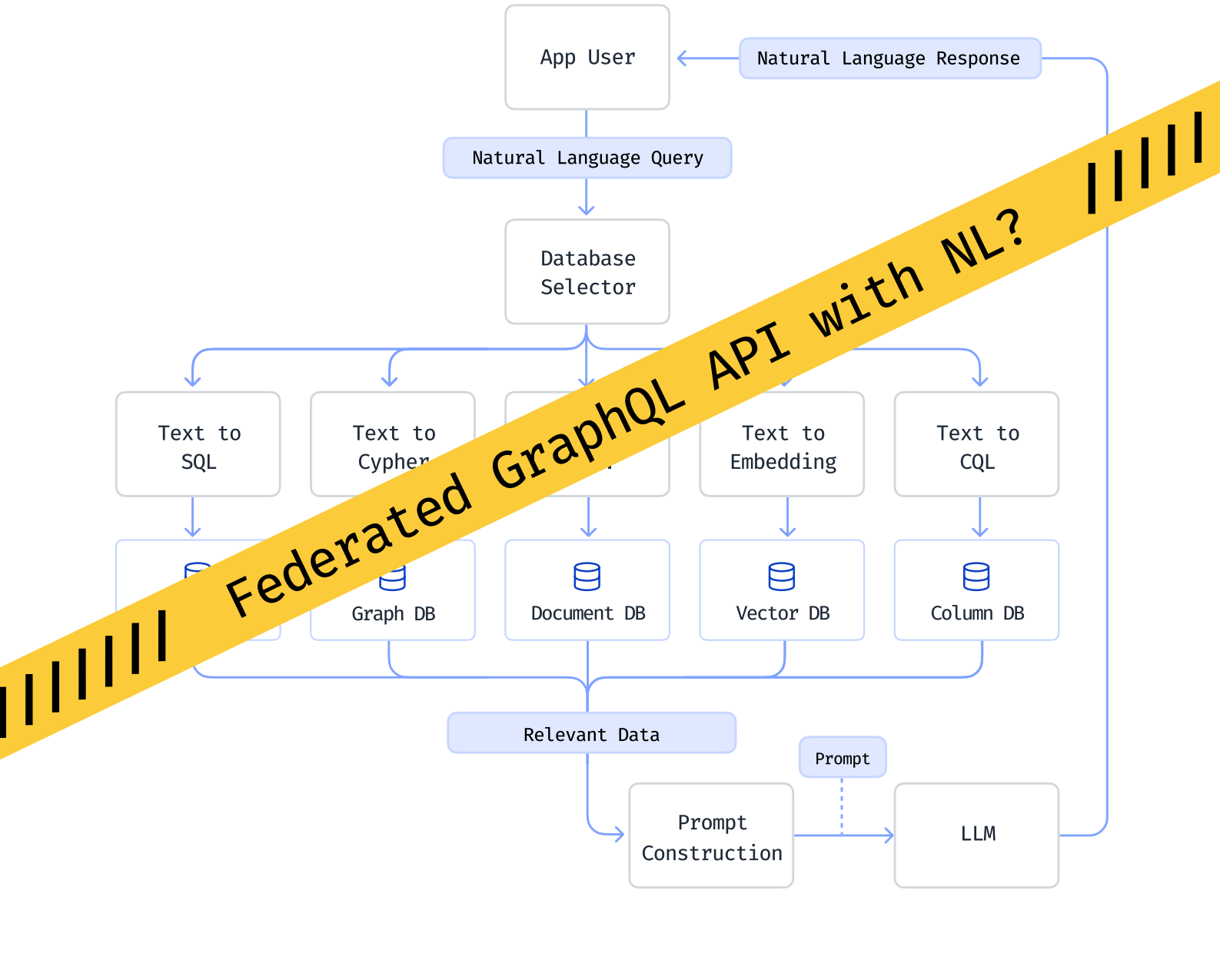
LLMs are machine learning models that are trained on data to comprehend user queries (called prompts) and generate human language text. LLMs usually do a good job of responding to prompts and they are getting better. However, from time to time they make up facts (called hallucinations) because of issues with the training data, etc. To prevent this hallucination, a process called retrieval-augmented generation (RAG) is used to support the LLM with additional, sometimes specialized data from additional data sources like databases, CRMs, etc. (his blog post from Nvidia does a great job of explaining it all!)

The problem in this space is that your data model is not transparent to your AI stack.

These problem statements should sound familiar to anyone who's encountered GraphQL federation. GraphQL federation architectures solve a similar problem (i.e. retrieving information from disparate sources of information based on a single user input, aka GraphQL query). So if you could enable your federated GraphQL API with natural language processing capabilities, you actually have a great solution for LLMs because the core problem is still one of federation.

In summary, your GraphQL stack and investment can play a critical role in supporting your inevitable AI play.





GraphQL value and ROI modeling

GraphQL can be of tremendous value to your organization. As a practitioner, you may be asked to estimate the costs and benefits of adopting GraphQL and prove out the ROI on paper before testing the hypothesis in practice. Here are some recommendations on how to build an ROI model for adopting GraphQL (and also on making good tradeoffs during the implementation phase as well).

4 steps to estimate GraphQL costs and benefits

At a high level, to build an ROI model for GraphQL, you will need to:

1. Be sure of your motivation and articulate your goals clearly.

2. Build an architecture and an operating model for your engineering teams:
   - Define an API quality rubric; be mindful of how your architecture will evolve over time.
   - Choose approaches that deliver quickly without sacrificing quality.
   - Identify redundant legacy components.

3. Build an ROI model with estimates based on architecture and its evolution.

4. Test your hypotheses with PoCs and iterate.

Factors influencing ROI

The devil is indeed in the details. There are several factors that influence an adoption journey and how you perceive and derive benefits from it. The following table studies the benefits of GraphQL across three different dimensions:

Dimension	Notes
Strategic vs. Tactical benefits	Strategic benefits provide a competitive advantage – the ability to leverage data across the entire org to build AI-driven user experiences. These benefits are hard to articulate accurately, but they can be game changers in the long run. Tactical benefits are easier to model, as well as benchmark and test. They kick in in the near term, so time is saved in building an API platform using GraphQL vs REST.
Schema-first vs. Domain-driven approaches	The two approaches have been contrasted above. When it comes to efficiency and productivity, domain-first GraphQL is naturally orders-of-magnitude better. To leverage this advantage, define your API quality requirement and test domain-first tools.
Frontend vs. Backend impact	GraphQL will ultimately make frontend engineers or API consumers more productive regardless of how you build GraphQL APIs, because GraphQL APIs inherently solve the API composition problem. GraphQL can only benefit backend engineers with the right approach. Traditionally, GraphQL APIs were built by frontend engineers. But over time, as the new domain data inevitably arose, GraphQL has become more of a backend engineering team concern.

What comprises a good ROI model?

Qualitative insights: A well thought out narrative for the motivation to use GraphQL, key goals or pain points to be addressed, architecture, and associated benefits.

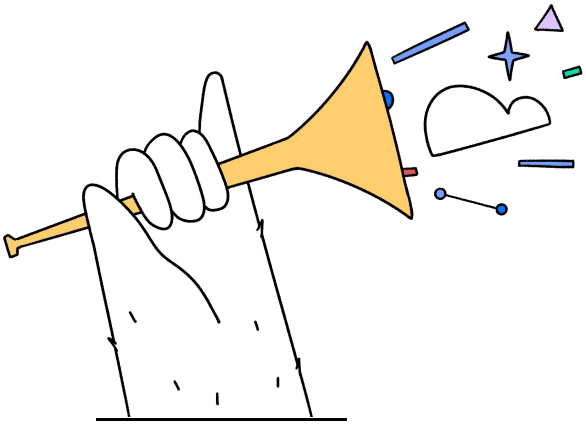
Quantitative gains: Pick your top quantitative gains. Keep your ROI model simple by focusing on key measurable gains that will help pay for the adoption effort. This can be a combination of cost savings, reduced hiring requirements, costs of redundant tools, profitable business outcomes, etc.

Example ROI models

The following simplified ROI models are based on feedback from GraphQL practitioners.

Goals	Costs	Benefits	ROI
Fortune 10 bank using domain-first subgraph tooling			
API platforms to: Provide a unified view to all consumers (website, mobile app, and other services). Improve customer experience, product autonomy, development efficiency, and integration capabilities. • Accelerate time-to-market • Reduce the need for constant API updates and maintenance	Building GraphQL services using efficient domain-driven tools and federating over them (vs. building a REST APIs platform). A few sprints for the backend engineering team.	Several million in cost savings (from choosing domain-driven GraphQL over REST).	~10X
Fortune 100 communications company using schema-first tooling			
Strategic: Improve time to market for features. **Tactical:** Improve developer productivity by reducing API chatter between frontend and backend engineering teams. Reduce operational overhead of maintaining APIs.	**Two-phased approach:** GraphQL facade to test benefits for frontend team. • 2-3 sprints for four engineers Building schema-first subgraphs to replace 80-100 REST endpoints. • 1-year effort for entire backend engineering team • Ongoing efforts to support new data requirements	The frontend team is 2x more productive. The backend team is 30% more productive post implementation.	~1.5X

Contact us to get the value and ROI model for your organization.



Summary

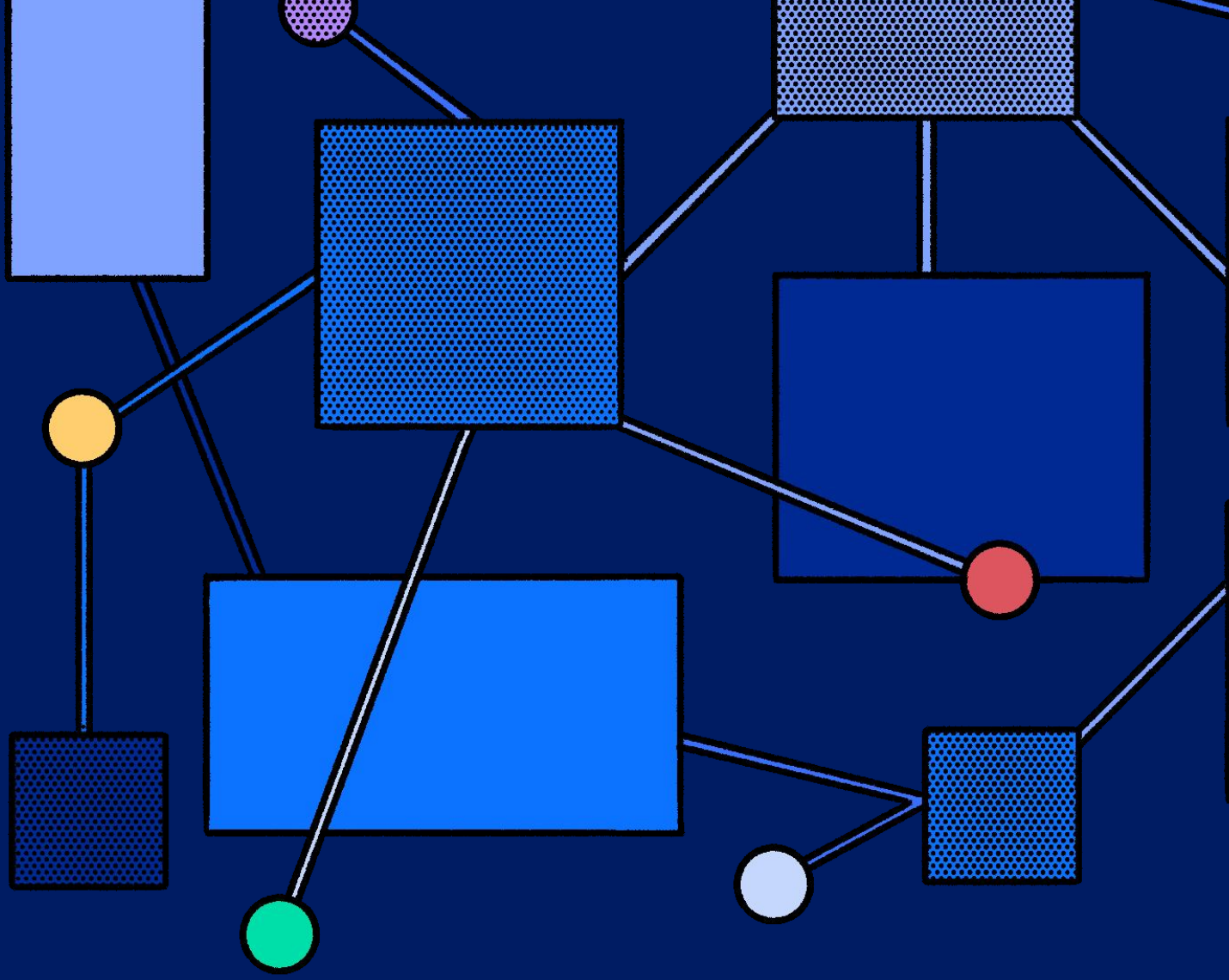
In this white paper, we have explored the expansive landscape of GraphQL as it stands in 2024, marking nearly a decade since its inception. From its beginnings as a response to the evolving needs of frontend technologies to its current status as a lingua franca for API design, GraphQL has undergone significant transformations.

This journey reflects not only the adaptability of GraphQL but also its ability to drive innovation in the realm of API development and integration.

Key highlights from our exploration include the back-to-the-backend trend, where GraphQL's success in frontend applications has necessitated a deeper integration and performance optimization at the backend level. The emergence of diverse subgraph tooling and the composition of graphs have further demonstrated GraphQL's flexibility and its capacity to streamline the development process across various domains. Additionally, the establishment of the GraphQL Foundation and the evolving GraphQL specification underscore the community's commitment to fostering an open, collaborative environment for GraphQL development.

As we look toward the future, it's clear that GraphQL will continue to play a pivotal role in shaping the API landscape, driven by its community, adaptability, and the ever-growing ecosystem of tools and practices. For developers and architects, the journey with GraphQL is far from static – it is a path of continuous learning, adaptation, and exploration, offering the promise of more efficient, scalable, and flexible API designs.

Whether you are just beginning your GraphQL journey or looking to deepen your existing expertise, the future of GraphQL is bright, and its potential to transform the API landscape is limitless.



To speak to our
team reach us on
hasura.io/contact-us

HASURA