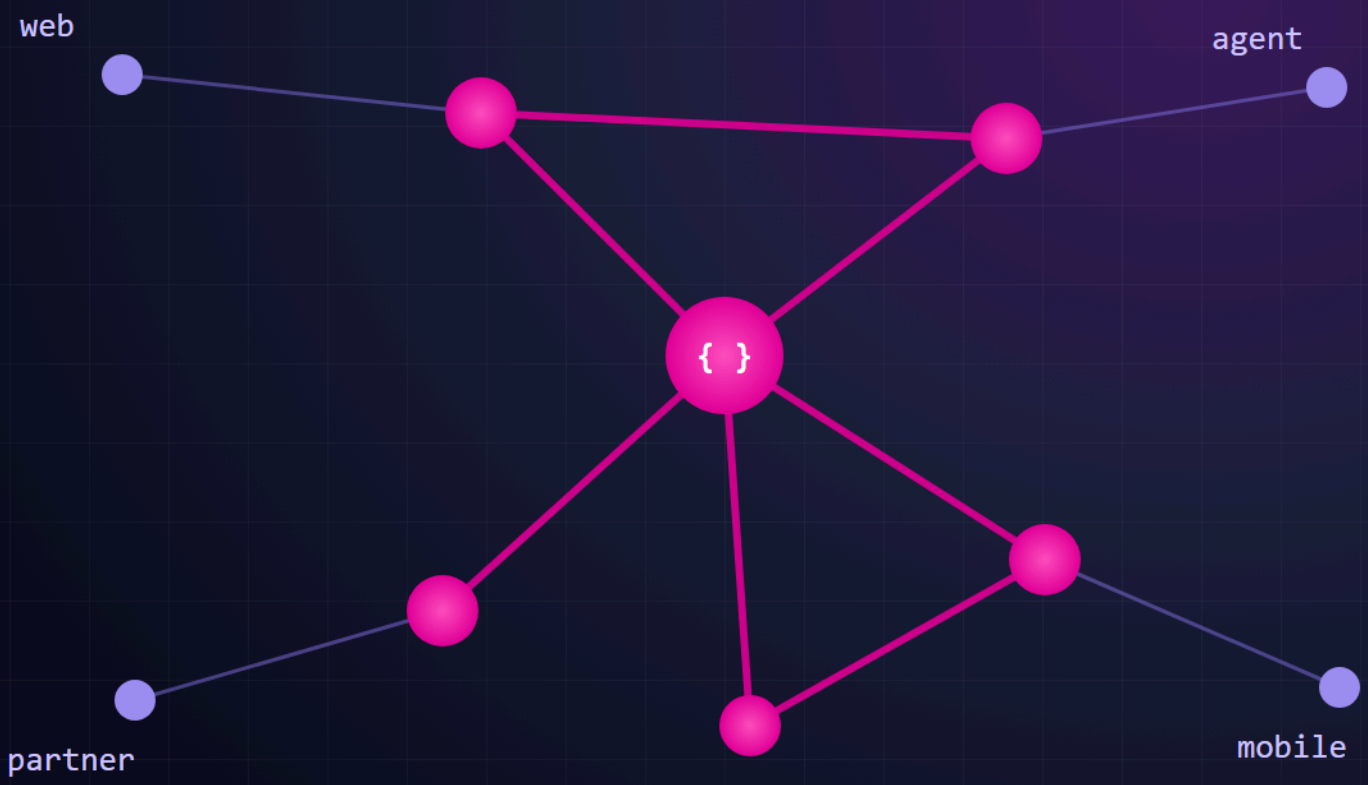


THE COMPLETE PRACTICAL GUIDE

GraphQL

The API Language for Humans and Agents

Design, build, secure, and scale GraphQL APIs — for people, apps, and AI agents

SCHEMA DESIGN · SERVERS · CLIENTS · FEDERATION · MCP

THENEXBOOKS

GraphQL

The API Language for Humans and Agents
A Complete, Practical Guide to Designing, Building, and Scaling GraphQL APIs

First Edition

THENEXBOOKS

GraphQL — The API Language for Humans and Agents
A Complete, Practical Guide to Designing, Building, and Scaling GraphQL APIs

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Libraries, frameworks, and specifications evolve; always consult the official documentation for the versions you use.

GraphQL is a trademark of The Linux Foundation / GraphQL Foundation. Apollo, Relay, urql, GraphQL Yoga, Spring, Strawberry, gqlgen, Hot Chocolate, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Preface

For most of the web's history, APIs were written for one kind of consumer: a developer who reads documentation, writes client code, and ships it. That consumer is still here — but they are no longer alone. Today your API is also called by mobile apps with tight bandwidth budgets, by partner integrations you will never see, and increasingly by **AI agents** that read your schema, decide which operations to call, and act on behalf of users.

GraphQL is unusually well suited to this moment. A GraphQL API is *self-describing*: its schema declares every type, field, argument, and relationship, together with human-readable descriptions. Clients ask for exactly the data they need, and every request is validated against the schema before it runs. Those properties make an API pleasant for humans — and legible, predictable, and safe for machines.

This book is a complete, practical guide to GraphQL, from the first query to a federated, production-grade graph used by both people and agents. We will not stop at syntax. You will learn how to design schemas that age well, how to build fast and secure servers, how clients cache and generate types, how to operate GraphQL at scale — and how to make your API genuinely *agent-ready*.

Who This Book Is For

- **Backend developers** building or maintaining GraphQL servers in TypeScript, Java, Python, Go, or .NET.
- **Frontend and mobile developers** who consume GraphQL and want to understand caching, code generation, and performance.
- **API designers and architects** choosing between REST, gRPC, and GraphQL, or planning a federated graph.
- **AI engineers** connecting LLM agents to real systems through tools, MCP servers, and safe, typed APIs.

What You Will Learn

- The GraphQL language and type system in depth — and what really happens inside validation and execution.
- Schema design principles for pagination, errors, evolution, and long-lived APIs.
- How to design schemas that AI agents can discover, understand, and use safely.
- Building servers in TypeScript and other languages; solving N+1 with DataLoader; authentication and authorization.
- Real-time GraphQL with subscriptions, SSE, and incremental delivery.
- Client libraries, normalized caching, persisted operations, and end-to-end type safety with code generation.

- Security, testing, observability, governance, and federation for large organizations.
- Three complete case studies, including an agent-ready SaaS API.

How This Book Is Organized

Part I — Foundations (Chapters 1–6) teaches the language, the type system, and the specification. **Part II — Designing Schemas** (Chapters 7–11) covers the design decisions that determine whether an API thrives or decays, including a dedicated chapter on designing for AI agents. **Part III — Building Servers** (Chapters 12–18) builds a production server step by step. **Part IV — Clients** (Chapters 19–21) looks at GraphQL from the consumer's side. **Part V — Production and Scale** (Chapters 22–25) addresses security, operations, federation, and agents in practice. **Part VI — Case Studies** (Chapters 26–28) brings everything together. The appendices provide cheat sheets, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Orbit Store

Throughout the book we build **Orbit Store**, a fictional e-commerce marketplace. Merchants sell products with variants and inventory; customers browse, review, and place orders; payments and shipments flow through the system. The Orbit Store API is consumed by a web app, a mobile app, partner integrations, and two AI agents: a shopping assistant for customers and an operations agent for merchants.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause bugs, data leaks, or outages in production.

Code appears in `monospace`. Server examples use TypeScript unless stated otherwise. Features that are still proposals or drafts are labeled as such.

Contents

Preface

PART I — FOUNDATIONS

1 Why GraphQL: The API Language for Humans and Agents

2 Core Concepts: Operations, Fields, and the Graph

3 The Type System and SDL

4 Querying in Depth

5 Mutations and Input Design

6 Inside the Spec: Validation, Execution, Introspection, and Responses

PART II — DESIGNING SCHEMAS

7 Schema Design Principles

8 Pagination, Filtering, and Sorting

9 Errors as Data: Designing Failure

10 Schema Evolution, Deprecation, and Versioning

11 Designing Schemas for AI Agents

PART III — BUILDING SERVERS

12 Building Your First Server (TypeScript, GraphQL Yoga)

13 Resolvers, Context, and the Execution Model

14 GraphQL Beyond JavaScript: Java, Python, Go, and .NET

15 The N+1 Problem, DataLoader, and Server Performance

16 Authentication and Authorization

17 Real-Time GraphQL: Subscriptions, SSE, @defer and @stream

18 Custom Scalars, Directives, and File Uploads

PART IV — CLIENTS

19 GraphQL Clients: From fetch to Apollo Client, urql, and Relay

20 Caching: Client, HTTP, CDN, and Persisted Operations

21 Code Generation and End-to-End Type Safety

PART V — PRODUCTION AND SCALE

22 Securing GraphQL APIs

23 Testing, Observability, and Schema Governance

24 Federation and Distributed Graphs

25 GraphQL for AI Agents in Practice: Tools, MCP, and Guardrails

PART VI — CASE STUDIES

26 Case Study: The Orbit Store Commerce API

27 Case Study: Migrating from REST to GraphQL

28 Case Study: Building an Agent-Ready SaaS API

APPENDICES

A SDL Cheat Sheet

B Query Language Cheat Sheet

C Production Checklist

D Glossary

PART I

Foundations

The language, the type system, and the specification — everything you need to read and write GraphQL fluently and to understand exactly what a server does with every request.

1 · Why GraphQL

2 · Core Concepts

3 · The Type System and SDL

4 · Querying in Depth

5 · Mutations and Input Design

6 · Inside the Spec

CHAPTER 1 · PART I — FOUNDATIONS

Why GraphQL: The API Language for Humans and Agents

GraphQL began as a pragmatic fix for a mobile app that was drowning in API calls, and it has grown into one of the most widely used ways to describe and consume data over the network. In this chapter you will learn where it came from, which problems it actually solves, how it compares honestly with REST, gRPC, and tRPC, and why a typed, self-describing schema has become unusually valuable now that AI agents are API consumers too.

A Short History: From a News Feed to a Foundation

In 2012, Facebook was rebuilding its native iOS and Android applications. The existing mobile apps were thin wrappers around web views, and the company had decided to go fully native for performance. That decision exposed a problem with the APIs underneath. The News Feed is a deeply nested structure: stories contain actors, actors have profile pictures, stories have comments, comments have authors, authors have their own pictures, and so on. The server-side data already lived as a graph of connected objects, but the APIs available to the mobile teams were a mix of resource-oriented endpoints and bespoke, screen-specific endpoints. Fetching a single screen of the feed required either many round trips over a slow mobile network or a custom endpoint that had to be changed every time the screen design changed.

A small team inside Facebook — Lee Byron, Nick Schrock, and Dan Schafer are the names most associated with the effort — designed a query language that let the client describe the exact shape of data it needed, in a single request, against a typed description of everything the server could provide. The server would validate the request against that description and return JSON whose shape mirrored the request. They called it GraphQL. It powered Facebook's mobile apps internally for about three years before it was shared with the world.

In 2015, Facebook publicly introduced GraphQL, published a draft specification, and open-sourced a reference implementation in JavaScript, `graphql-js`. The choice to publish a *specification* rather than just a library turned out to matter enormously. Because the language, the type system, validation rules, and execution semantics were written down precisely, independent teams could build compatible servers in Java, Python, Go, Ruby, .NET, Rust, and many other languages, and client tools could work against any of them.

In 2018 the project moved to a neutral home: the **GraphQL Foundation**, hosted by the Linux Foundation, with members from across the industry. The foundation stewards the specification, `graphql-js`, and related projects such as `graphql-http` and GraphiQL, and runs working groups

where changes to the spec are proposed and debated in the open. Formal editions of the specification have been cut periodically; throughout this book, when we talk about "the spec," we mean the GraphQL specification (October 2021 edition and later). Proposals that are still moving through the process — incremental delivery with `@defer` and `@stream`, the GraphQL-over-HTTP specification, and the Composite Schemas specification for distributed graphs — will always be labeled as drafts or RFCs when we meet them.

> **NOTE**
>
> GraphQL is a *specification* for a query language and a type system, plus rules for validating and executing requests. It is not a database, not a storage engine, and not tied to any particular transport. Most deployments use HTTP, but the spec itself does not require it.

The Problems GraphQL Was Built to Solve

It is easy to describe GraphQL as "a better REST," but that framing hides the specific pains it addresses. If you understand those pains, you will also understand when GraphQL is not worth it. There are four recurring problems.

Over-fetching

A typical resource endpoint returns a fixed representation. `GET /products/42` returns every field the backend team thought a product should have: descriptions in several formats, all variants, all images, SEO metadata, warehouse codes. A product-card component on a phone needs a title, a price, and one thumbnail. Everything else is transferred, parsed, and thrown away.

Under-fetching and the round-trip cascade

The opposite problem is at least as painful. To render an order-history screen, a client might need the customer, their recent orders, each order's lines, each line's product, and each order's shipment status. With purely resource-oriented endpoints, that becomes a waterfall: fetch the customer, then fetch the orders, then for each order fetch lines, then for each line fetch the product. Each step depends on IDs returned by the previous one, so the requests cannot all run in parallel. On a mobile network with 100 ms or more of latency per round trip, a four-level waterfall is visibly slow before the server has done any real work.

GraphQL moves the traversal to the server. The client sends one document describing the whole tree, and the server — which sits next to the databases and services, on a fast network — walks the graph and returns the result in one response.

Versioning pain

When a resource-oriented API needs to change the shape of a response, the safe move is usually a new version: `/v2/products`. Versions multiply, old ones linger because some mobile client installed two years ago still calls them, and every change becomes a negotiation about whether it is "breaking."

GraphQL changes the unit of compatibility. Because clients select fields explicitly, the server can add new fields and types at any time without affecting existing clients — nobody receives a field they did not ask for. Fields that should go away are marked with `@deprecated`, which tools surface to developers, and the server can measure exactly which clients still select them before removal. Many large GraphQL APIs have run for years with a single, continuously evolving schema and no version numbers. Chapter 10 covers the discipline this requires; it is not free, but it is far more granular than versioning a whole API.

Weak contracts

Finally, many HTTP APIs have contracts that live in prose documentation, in an OpenAPI file that may or may not match the running server, or only in the heads of the people who wrote them. Clients discover mismatches at runtime, usually in production. GraphQL makes the contract the schema itself. The server cannot serve a field that is not in the schema, and it validates every incoming request against the schema before executing anything. The schema can be queried at runtime through **introspection**, so tools always see what the server really exposes — not what someone remembered to document.

What GraphQL Looks Like

Before comparing GraphQL with alternatives, it helps to see a request and response side by side. Here is a query a mobile client for our running example, Orbit Store, might send to render an order-history screen:

`OrderHistory.graphql`

```
query OrderHistory {
  viewer {
    displayName
    orders(first: 3) {
      id
      placedAt
      status
      total { amount currency }
      lines {
        quantity
        product { title thumbnailUrl }
      }
    }
  }
}
```

And here is the shape of the response:

```
{
  "data": {
    "viewer": {
      "displayName": "Ana",
```

```
      "orders": [
        {
          "id": "T3JkZXI6MTAwMQ==",
          "placedAt": "2026-03-14T09:21:00Z",
          "status": "SHIPPED",
          "total": { "amount": "84.50", "currency": "EUR" },
          "lines": [
            {
              "quantity": 2,
              "product": {
                "title": "Nebula Ceramic Mug",
                "thumbnailUrl": "https://cdn.orbit.example/p/9/t.jpg"
              }
            }
          ]
        }
      ]
    }
  }
}
```

Notice three things. First, the response mirrors the query: every key in the JSON corresponds to a field the client selected, in the same nesting. Second, the client got exactly the fields it asked for and nothing more. Third, one request fetched data that would have taken several dependent calls in a typical resource-oriented design. We will dissect this pattern in detail in Chapter 2; for now, keep the mental picture of a client describing a tree and the server filling it in.

An Honest Comparison: REST, gRPC, tRPC, and GraphQL

GraphQL is one tool among several good ones. Choosing well means knowing what each approach optimizes for. The following table summarizes the trade-offs; the prose after it explains the nuances that a table cannot.

Aspect	REST	gRPC	tRPC	GraphQL
Contract	Optional (OpenAPI)	Required (.proto)	TS types	Required (schema)
Wire format	Usually JSON	Protobuf binary	JSON	Usually JSON
Transport	HTTP	HTTP/2	HTTP	Usually HTTP
Client picks fields	No (by convention)	No	No	Yes
Languages	Any	Many	TypeScript	Many
HTTP caching	Excellent	Limited	Moderate	Needs work
Streaming	SSE, WebSocket	Built in	Subscriptions	Subscriptions
Introspection	Via OpenAPI	Reflection (opt.)	No runtime	Built in
Best fit	Public resources	Service-to-service	TS monorepos	Varied clients

REST

REST is an architectural style, not a specification, and "REST API" in practice usually means "JSON over HTTP organized around resources." Its greatest strength is that it aligns with the web itself. URLs identify resources, HTTP verbs carry intent, status codes carry outcome, and the entire HTTP caching infrastructure — browser caches, reverse proxies, CDNs — works out of the box for `GET` requests.

The weaknesses are the ones described above: fixed response shapes, waterfalls for nested data, coarse versioning, and contracts that are optional rather than enforced. None of these are unsolvable — sparse fieldsets, compound documents (as in JSON:API), and careful OpenAPI discipline all help — but each API solves them differently.

gRPC

gRPC, open-sourced by Google in 2015, is a remote procedure call framework built on HTTP/2 and Protocol Buffers. You define services and messages in `.proto` files, generate client and server code in many languages, and exchange compact binary messages. It supports unary calls and client, server, and bidirectional streaming. For internal service-to-service traffic, where both ends are under your control and throughput matters, gRPC is excellent: strongly typed, fast, and efficient on the wire.

It is less comfortable at the edge. Browsers cannot speak native gRPC directly, so web clients need gRPC-Web and a proxy, or an alternative such as Connect. Responses have fixed shapes, like REST. Many organizations happily use gRPC between services and GraphQL at the edge, with GraphQL resolvers calling gRPC backends.

tRPC

tRPC takes a very different approach: in a TypeScript codebase where the client and server share a repository, it lets the client call server procedures with full type inference and no code generation step and no separate schema language. The types *are* the contract.

The limits follow from the design. tRPC is TypeScript-only; a Swift mobile client or a Python partner integration gets none of the type benefits. There is no language-neutral schema to publish, no runtime introspection, and no client-side field selection. tRPC is a great choice for a private API between a TypeScript frontend and backend that deploy together. It is not designed to be a public or multi-client API.

GraphQL

GraphQL's sweet spot is an API consumed by *many different clients with different needs*: web, iOS, Android, partner integrations, internal tools, and increasingly AI agents. The schema is a language-neutral, enforced, introspectable contract. Clients select exactly the fields they need, traverse relationships in one request, and evolve independently of the server.

The costs are real, and we will not hide them. HTTP caching does not work by default because most requests are `POST`s to a single URL; you need persisted operations or `GET` requests to recover it (Chapter 20). A naive server implementation suffers from the N+1 query problem (Chapter 15). Because clients can write arbitrary queries, you must defend against expensive or malicious ones with depth limits, cost analysis, and allowlists (Chapter 22). Errors do not map neatly onto HTTP status codes, which surprises operations teams (Chapter 9). And there is more machinery to learn — a type system, a query language, resolvers, and tooling.

> **TIP**
>
> These approaches are not mutually exclusive. A common, healthy architecture uses gRPC or REST between internal services, GraphQL as the unified API that clients and agents talk to, and plain REST endpoints for things like webhooks, file downloads, and health checks.

A Self-Describing Schema: Why It Matters for Humans

The single feature that most distinguishes GraphQL from the alternatives is that the API describes itself, in a machine-readable form, at runtime, and that description is guaranteed to be accurate because the server uses the same description to validate and execute requests. Here is a fragment of Orbit Store's schema written in the GraphQL Schema Definition Language (SDL):

schema.graphql (excerpt)

```
"""
A product listed by a merchant. Prices live on variants, not on the product.
"""
type Product {
  id: ID!
  title: String!
  "Markdown description written by the merchant."
  description: String
  merchant: Merchant!
  variants: [ProductVariant!]!
  averageRating: Float
  reviews(first: Int = 10, after: String): ReviewConnection!
}
```

For a human developer, this does several jobs at once. It is documentation: the triple-quoted strings are descriptions that tools display next to fields. It is a type contract: `title` is a `String` that is never null (the `!`), `description` may be null, `variants` is a non-null list of non-null variants. And it is a map of the graph: you can see that from a product you can reach its merchant, variants, and reviews.

Because this schema is available through introspection, tools can build rich experiences on top of it without any extra configuration:

- **Interactive explorers** such as GraphiQL and Apollo Sandbox offer autocomplete, inline documentation, and instant validation as you type a query.

- **Code generators** such as GraphQL Code Generator produce TypeScript types, typed hooks, Swift or Kotlin models from the schema and your operations, so a field rename becomes a compile error instead of a production incident (Chapter 21).
- **Schema registries and CI checks** diff schema versions and flag breaking changes before they ship (Chapter 23).

A Self-Describing Schema: Why It Matters for AI Agents

The same properties that help humans turn out to matter even more for AI agents — software built around large language models that decide at runtime which tools to call and with which arguments. Orbit Store has two such consumers: a **shopping assistant** that helps customers find products, compare options, and track orders, and a **merchant operations agent** that helps sellers adjust inventory, answer questions about sales, and handle routine support. Neither was hand-coded against specific endpoints. Both reason about the API from its description. Three properties make GraphQL a strong fit.

Discoverability

An agent needs to know what it *can* do before it decides what to do. With GraphQL, the complete, current list of types, fields, arguments, enums, and their descriptions is available through a standard introspection query, or as SDL text that fits comfortably into a model's context for small and medium schemas. Descriptions written for humans — "Prices live on variants, not on the product" — are exactly the kind of guidance that helps a model avoid a wrong turn. Enum types such as `OrderStatus` give the model a closed list of valid values rather than a free-text field it has to guess about.

Precise field selection

Every token an agent reads costs money and consumes finite context. A REST response that dumps forty fields of product data, including long HTML descriptions and nested arrays the agent does not need, wastes context and can distract the model with irrelevant detail. With GraphQL, the agent asks for `title`, `averageRating`, and the price of the cheapest variant, and gets back a compact JSON object containing precisely that. Smaller, more focused responses tend to be cheaper to process and easier for a model to use correctly.

Validation before execution

Language models make mistakes: a misspelled field, a string where an integer belongs, a missing required argument. In GraphQL, every request is validated against the schema *before* any resolver runs. A malformed operation fails fast with a structured error that names the problem and its location in the document:

```
{
  "errors": [
    {
```

```
      "message": "Cannot query field \"prce\" on type \"ProductVariant\".
        Did you mean \"price\"?",
      "locations": [{ "line": 4, "column": 7 }]
    }
  ]
}
```

(The message is wrapped here for the page; the server returns it on one line.) The "Did you mean" suggestion shown is what `graphql-js` produces; other servers word their messages differently, and some production deployments disable suggestions to avoid leaking schema details. Either way, nothing was executed and no data was touched, and the agent receives an error it can use to correct itself and retry. For mutations — operations that change data — validation before execution is a genuine safety property: an agent cannot accidentally trigger half of a side effect because of a type error in its request.

> **WARNING**
>
> Validation checks that a request is *well-formed*, not that it is *wise*. A perfectly valid mutation can still cancel the wrong order. Authorization, confirmation steps, and scoped permissions for agents remain your responsibility; Chapters 16, 22, and 25 cover them in depth.

None of this makes GraphQL magically "AI-native." A poorly named, undocumented GraphQL schema is as confusing to a model as it is to a person, and a huge schema can overwhelm a model's context just as a huge OpenAPI file can. Chapter 11 is devoted to designing schemas that agents use well, and Chapter 25 shows how to expose GraphQL operations as tools, including through the Model Context Protocol (MCP), the open protocol Anthropic introduced in November 2024 for connecting models to tools and data. The point for now is that GraphQL gives you strong raw material: a typed, validated, introspectable contract with room for rich descriptions.

When Not to Use GraphQL

A book about GraphQL that never told you when to avoid it would not be worth your money. GraphQL is often the wrong choice in these situations:

- **One client, one team, one language.** If a single TypeScript web app talks to a single backend that deploys with it, tRPC or a few well-typed REST endpoints may give you most of the benefit with far less machinery.
- **Simple, cache-heavy public resources.** A public catalog of mostly static documents, served to anonymous users through a CDN, benefits enormously from plain `GET` URLs with cache headers. GraphQL can be cached (Chapter 20), but you are working against its defaults.
- **High-throughput internal RPC.** Service-to-service calls with fixed shapes and strict latency budgets are gRPC's home turf. Field selection adds little when both ends are owned by the same team.

- **Bulk data transfer and files.** Exporting millions of rows, streaming video, or uploading large files is better handled by dedicated endpoints or object storage with signed URLs. GraphQL can coordinate these flows but should not carry the bytes.
- **Webhooks and event delivery to third parties.** Pushing events to a partner's URL is a plain HTTP concern; GraphQL subscriptions are for clients that hold a connection open to you.
- **Teams without capacity to operate it.** Query cost limits, DataLoader batching, persisted operations, schema governance, and observability are not optional at scale. If your team cannot invest in them, a simpler API will be more reliable.

A useful heuristic: GraphQL pays off in proportion to the *variety* of clients and the *connectedness* of the data. Many clients over a rich, interconnected domain — a commerce platform, a social product, a SaaS application with dashboards, mobile apps, and integrations — is where it shines. One client over a flat set of resources is where it is overhead.

Common misconceptions

A few myths come up often enough to address directly:

- **"GraphQL is a database query language."** It is not. It has no joins, no storage model, and no knowledge of your database. Resolvers fetch data from wherever it lives — SQL, NoSQL, microservices, third-party APIs — and the schema should model your domain, not your tables.
- **"GraphQL always returns HTTP 200."** Historically many servers did, but the draft GraphQL-over-HTTP specification defines when non-200 status codes should be used, particularly with the `application/graphql-response+json` media type. Chapter 6 explains the details.

Introducing Orbit Store

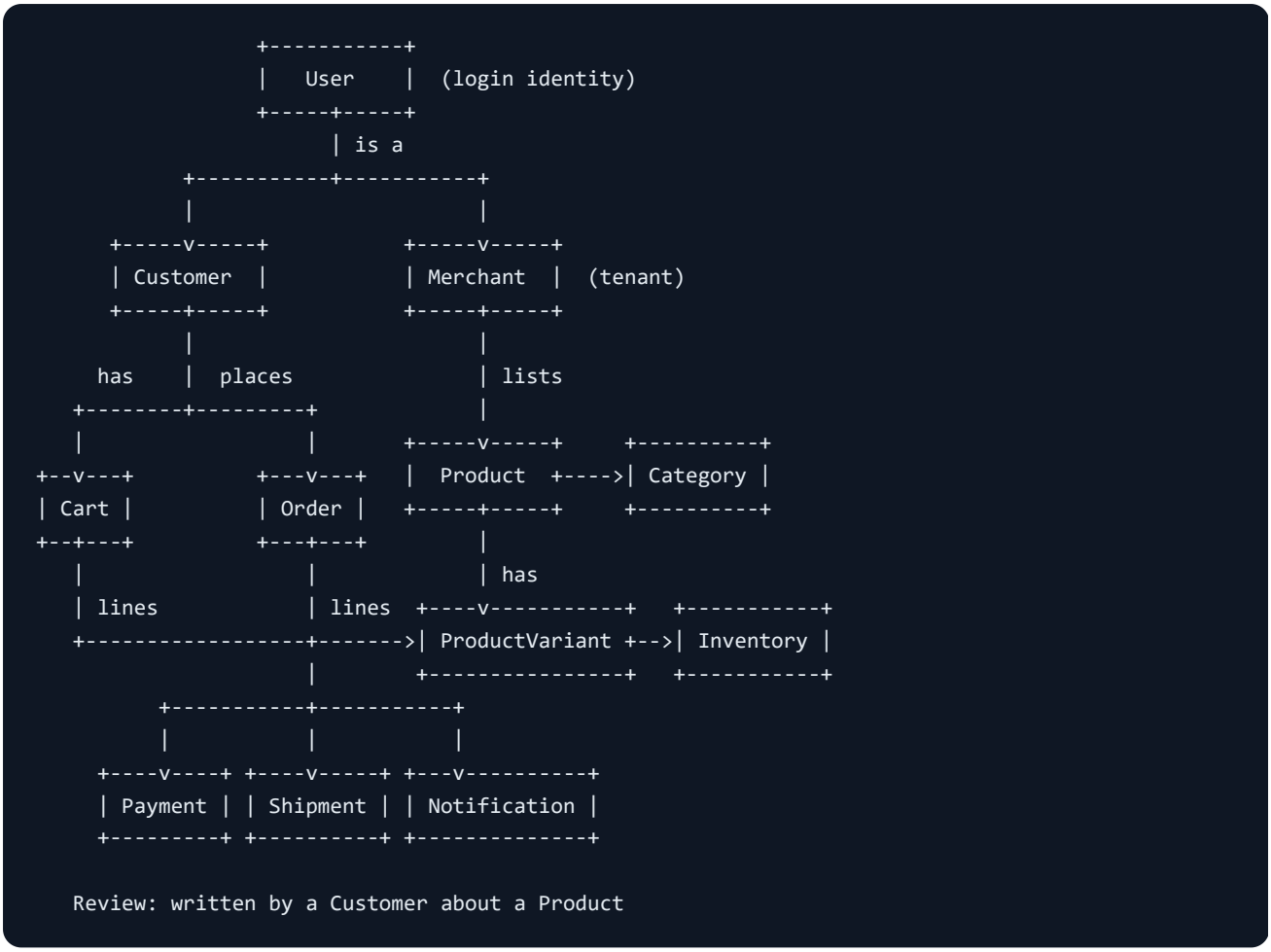
Throughout this book we will design, build, secure, and scale one API: **Orbit Store**, a fictional multi-merchant e-commerce marketplace. Using a single, realistic domain lets each chapter build on the last and lets us show trade-offs that only appear when a schema has real relationships, real permissions, and real clients.

The business

Orbit Store is a marketplace, not a single shop. Independent **merchants** — ceramicists, bike-part suppliers, specialty coffee roasters — sign up and list products. **Customers** browse across all merchants, add items to a single cart, and check out once; the platform splits the order among merchants behind the scenes. Each merchant is a *tenant*: they see only their own products, orders, inventory, and customers' order details, never another merchant's. This multi-tenant boundary will matter a great deal when we get to authorization in Chapter 16.

The domain model

The core types and their relationships look like this:



A few design decisions are worth calling out now, because later chapters depend on them:

- **User vs. Customer vs. Merchant.** A `User` is a login identity. A user may act as a `Customer`, as staff of a `Merchant`, or both. Separating identity from role keeps authorization rules clean.
- **Prices live on variants.** A T-shirt `Product` has `ProductVariant`s for each size and color, and each variant has its own SKU, price, and `Inventory` record.
- **Orders are snapshots.** An `OrderLine` records the price and title at the time of purchase, because product data changes after the fact.
- **Money is not a float.** Amounts are represented as a `Money` type with a decimal string amount and an ISO 4217 currency code. Chapter 18 turns this into a custom scalar discussion.
- **Notifications are events.** Order updates, shipment tracking changes, and low-stock alerts are delivered as `Notification`s, which also feed real-time subscriptions in Chapter 17.

The consumers

Orbit Store has four kinds of client, each with different needs — which is precisely why GraphQL is a good fit:

Client	Main needs	Constraints
Web storefront	Browse, search, checkout	SEO, fast first paint

Mobile app	Same, plus push updates	Slow networks, old versions
Partner integrations	Sync catalog, orders	Stable, long-lived
Shopping assistant	Search, compare, track	Token budget, safety
Merchant ops agent	Inventory, sales, support	Tenant scope, approvals

The web storefront wants rich product pages rendered on the server. The mobile app wants the smallest possible payloads and must keep working for users who have not updated in a year. Partners want predictable, stable operations that will not change under them. The shopping assistant wants compact, well-described data and must never place an order without the customer's explicit confirmation. The merchant operations agent must be strictly confined to one merchant's data and must request approval before making large inventory or pricing changes.

A first taste of the schema

Here is a small slice of Orbit Store's entry points, just to set expectations. Chapter 3 builds out the complete SDL.

schema.graphql (entry points, excerpt)

```
type Query {
  "The currently authenticated user, or null for anonymous requests."
  viewer: User
  product(id: ID!): Product
  searchProducts(query: String!, first: Int = 20): [Product!]!
  order(id: ID!): Order
}

type Mutation {
  addToCart(variantId: ID!, quantity: Int!): Cart!
  checkout(cartId: ID!): Order!
}

type Subscription {
  orderStatusChanged(orderId: ID!): Order!
}
```

Three root types — `Query` for reads, `Mutation` for writes, `Subscription` for real-time events — are the doors into the graph. Everything else is reached by following fields from these roots. Later chapters will refine these fields considerably: pagination will move to connections (Chapter 8), mutations will take input objects and return payload types with typed errors (Chapters 5 and 9), and agent-facing operations will get careful descriptions and guardrails (Chapter 11).

The server stack

Unless a chapter says otherwise, our server examples use TypeScript, **GraphQL Yoga** as the HTTP server, and `graphql-js` as the execution engine underneath. Where Apollo Server is the more natural choice — for example, in parts of the federation discussion — we will say so. Here is roughly the

smallest possible Orbit Store server, to show that the core idea is not complicated:

server.ts

```
import { createServer } from "node:http";
import { createSchema, createYoga } from "graphql-yoga";

const products = [
  { id: "1", title: "Nebula Ceramic Mug" },
  { id: "2", title: "Comet Trail Running Socks" },
];

const schema = createSchema({
  typeDefs: /* GraphQL */ `
    type Product {
      id: ID!
      title: String!
    }
    type Query {
      product(id: ID!): Product
    }
  `,
  resolvers: {
    Query: {
      product: (_parent: unknown, args: { id: string }) =>
        products.find((p) => p.id === args.id) ?? null,
    },
  },
});

const yoga = createYoga({ schema });
const server = createServer(yoga);

server.listen(4000, () => {
  console.log("Orbit Store API on http://localhost:4000/graphql");
});
```

A type definition, a resolver function that returns data for a field, and a server that exposes it over HTTP: that is the whole pattern. Everything else in this book — pagination, errors, authorization, batching, caching, federation, agent tooling — is about doing this well at scale. Chapter 12 builds a real server step by step.

How This Book Is Organized

The book is divided into six parts, designed to be read in order the first time and used as a reference afterward.

Part	Chapters	Focus
I — Foundations	1–6	Language, types, spec
II — Designing Schemas	7–11	Modeling, errors, evolution