The Production-Ready Playbook

Scalable Cloud Architecture and Design Patterns for Fractional Teams — The Pareto Pattern Stack.

A YOU-SOURCE BOOK

CONTENTS

The Production-Ready Playbook

SCALABLE CLOUD ARCHITECTURE AND DESIGN PATTERNS FOR FRACTIONAL TEAMS

The Pareto Pattern Stack

The few patterns that actually take software from "works on my machine" to "runs in production for years," and the judgement to know which ones a small team can afford to carry.

Foreword

This is the book I always wanted on my own shelf and never found. Plenty come close. They list every pattern ever named, sort them into neat families, and leave you to work out which five of the hundred actually matter on a Tuesday when something is on fire and there are two of you. I wanted the opposite of a catalogue. I wanted the short list.

I have carried these patterns across most of a career, from teams with a platform group two floors down to jobs where I was the platform group. The thing nobody tells you early on is that the patterns you reach for change completely once you are the one who has to maintain them. A pattern is not free. You adopt it, and then you live with it, and the cost shows up months later as the thing a new hire cannot reason about. So over the years I kept a private list of the ones that paid rent. This book is that list, written down at last.

These are not patterns I admire from a distance. They are the ones my teams reach for every day, across the work we ship for clients and our own systems, because they are what hold up once the platform group is just us. The list stopped being private a while ago. It became the thing we hand a new engineer in their first week, the spine of how we train and onboard, the shared vocabulary that lets someone unfamiliar with a codebase still read its intentions. Writing it down here is mostly an act of putting in one place what we had already scattered across a dozen onboarding docs.

I should be candid about why it exists now and not five years ago. The patterns were always in my head. What I lacked was a tireless writing partner who would sit with me at midnight and argue about whether Saga earns its slot, draft a Dapper snippet I could then tear apart, and never once get tired of me cutting it back to the bone. The structure, the opinions, the skip-ifs, the calls about what stays out: those are mine, and I will defend every one. The endurance to

get them onto the page came from working alongside an AI that did not need sleep. That is the honest account, and I would rather give it than pretend the book wrote itself the old way.

What I have tried to build is a working engineer's bible. Not the whole law, just the verses you say from memory. There is a small set of patterns every self-respecting engineer should know cold, the way a carpenter knows a handful of joints. The ones you genuinely dream about, in the good way and occasionally the bad. The rest you can look up.

You will notice what is missing. That is the point. For every pattern I kept, three are sitting in the skip list at the back with a reason attached. A book that teaches you all twenty-three object patterns and the dozens more in the cloud catalogues is teaching you a search index, and you already have one of those. This book bets that the right twenty per cent, learned properly, beats the whole catalogue learned shallowly. I think the bet is sound. By the last chapter I hope you do too.

One more thing, said plainly because engineers can hear a sales pitch through a wall. I am not going to dress these patterns up. Each one has a carrying cost, and naming that cost is half of what the book is for. If a pattern only makes sense at a scale you do not have, I will tell you to skip it. That is the whole spirit of the thing.

> *You don't need the whole catalogue. You need the right twenty per cent, climbed in order, applied with judgement.*

The premise

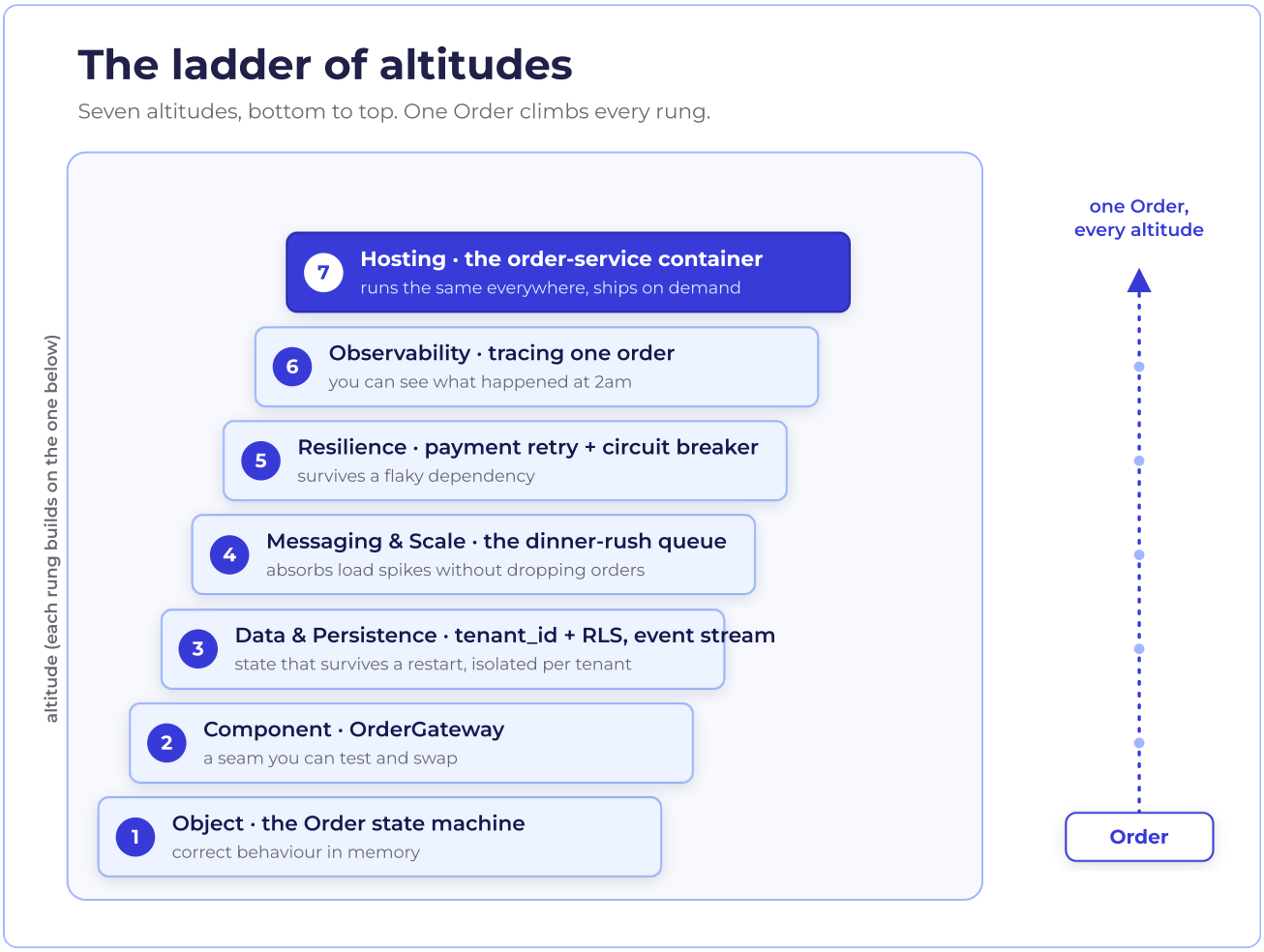
This is a field guide for working engineers and the small teams they build with: tech leads, solo builders, and the fractional engineer dropped into an unfamiliar codebase on Monday morning. It assumes you can read code faster than a paragraph about code, that you are short on time, and that you have to ship something which survives production and then keep it alive without a platform team behind you. Skip the reference manual. What you get instead is an opinionated curation of the few patterns that earn their place, each one paired with the honest signal that a small team should not reach for it yet.

The thesis

A small, learnable set of patterns takes software from "works on my machine" to "runs in production for years." Call it the 80/20 of architecture: the top few patterns at each altitude do

almost all the work, and the rest is a tax most teams cannot afford. You do not need the whole catalogue. You need the right twenty per cent, climbed in order, applied with judgement.

To prove they hold together rather than arrive as disjointed snippets, the book builds one app the whole way up. A food-delivery marketplace, the kind you have ordered from: a customer, a restaurant, a courier, and an `Order` that has to survive every altitude. The same `Order` you meet as a State machine early on is the one you trace across payment and courier matching later, and the one the reference service assembles at the end. Each pattern earns its place against a feature you can already picture.



The promise

By the last page you will own a vocabulary that builds production-grade software with a small team, and the restraint to leave the rest on the shelf.

> *Maintain what you ship.*

Start where every system starts: the catalogue problem, and the small set that solves it.

The Pareto Stack: Cloud Design Patterns for Small Teams

A small, curated set of cloud design patterns builds almost any real system. You don't need the whole catalogue. You need the right twenty per cent, climbed in order and applied with judgement.

Open any list of cloud design patterns and you start to drown. The Gang of Four catalogued 23 object-level patterns in 1994 and that book is still in print. The Azure Cloud Design Patterns catalogue adds dozens more for messaging, data, and resilience. microservices.io lists dozens again. Stack the enterprise integration patterns on top, then the persistence patterns from Fowler's *PoEAA*, and a working engineer with a deadline is staring at well over a hundred named things, each with a diagram and a tradeoff and a chorus of blog posts insisting it is the one you cannot ship without.

Two things happen to a small team in front of that pile. The first team reads the catalogue, panics, and reaches for everything: a message bus for three services, event sourcing for a CRUD form, a service mesh before the second deployment. They ship a distributed system they cannot operate, and the pager owns them by month two. The second team reads the catalogue, panics, and reaches for nothing: no retries, no health check, no structured logs, one connection pool quietly leaking until it falls over at 3am. Both teams lost to the same enemy. Too much choice, not enough judgement.

> *Completeness is the catalogue's job. Curation is yours.*

This book takes the opposite bet. There is a finite, learnable set of patterns that carries software from "works on my machine" to "runs in production for years," and it is far smaller than the catalogue suggests. The book curates a small core of patterns at each of seven altitudes: object, component, data and persistence, messaging and scale, resilience, observability, and hosting. A few dozen in all, applied with judgement, that build pretty much any modern, cloud-agnostic, containerised system a small team will actually be asked to build. We call it the Pareto stack. The right twenty per cent.

To keep the patterns honest, the whole book teaches them through one running example: a food-delivery marketplace, the kind you already use to get dinner (Uber Eats, DoorDash, Deliveroo). Three sides: a customer who orders, a restaurant that cooks, a courier who delivers. Every chapter solves a real feature of that same app instead of inventing a throwaway snippet, so you watch the patterns compose rather than meet each one in isolation. The `Order` you

meet at the object level is the same `Order` that gets retried under load in the resilience chapter and assembled into a reference Order Service at the end. By the last page the example is a small system, built from features you already understand because you have ordered from one.

Patterns are vocabulary, not architecture

Start with what a pattern actually is, because the catalogue obscures it. A pattern is a name for a shape you'd have arrived at anyway. "Circuit Breaker" is not an architecture; it's a word for the thing you do when you stop hammering a dead payment provider. "Strategy" is a word for swapping one behaviour for another behind a stable interface, which is exactly what `CourierMatchingStrategy` does when it picks the nearest courier today and the cheapest one tomorrow. The value of the name is that your teammate reads it in a pull request and knows the shape in one second, with no paragraph of explanation.

That makes patterns a shared language, not a building. You can know every word in the dictionary and still write a bad sentence. Naming the pattern does none of the design for you. You still have to decide where the boundary goes, what the failure mode is, and whether this service even needs the thing.

> *A pattern is a word, not a wall. You still have to design the building.*

So the goal of learning patterns is fluency, not coverage. You want enough vocabulary that you can read a system, name its parts, and reach for the right shape without reinventing it. Beyond that point, more catalogue is a tax. The hundredth pattern you half-remember is not knowledge; it's a thing you'll cargo-cult into a codebase that didn't ask for it.

The selection criterion

If we're cutting the catalogue down, we need an honest rule for what stays. Here it is, and it runs through every chapter: **maximum production-readiness per unit of team capacity.** A pattern earns its slot by what it buys you in production, measured against what it costs you to carry. Cleverness doesn't enter into it, and neither does whatever's current this quarter. A pattern that only looks good in an architecture diagram scores below zero, because now someone has to build the diagram.

"Per unit of team capacity" is the part the catalogues leave out. A pattern that a fifty-person platform team runs without thinking can sink a team of three, because the three carry it forever. Every pattern you adopt is one you maintain. There's code to understand and infrastructure to run, and one night there's a failure mode to debug at 3am. That carrying cost is the over-engineering tax, and we'll name it for every pattern in the book. A pattern with no production payoff you can state in a sentence is not a pattern you've chosen. It's one you've cargo-culted.

This is why "what it buys you in production" is the framing for every pattern's payoff. If you can't finish the sentence "I'm adding this because in production it..." then you haven't earned the pattern yet. Curation is not us being precious. It's the only way a small team gets the value of the catalogue without the weight of it.

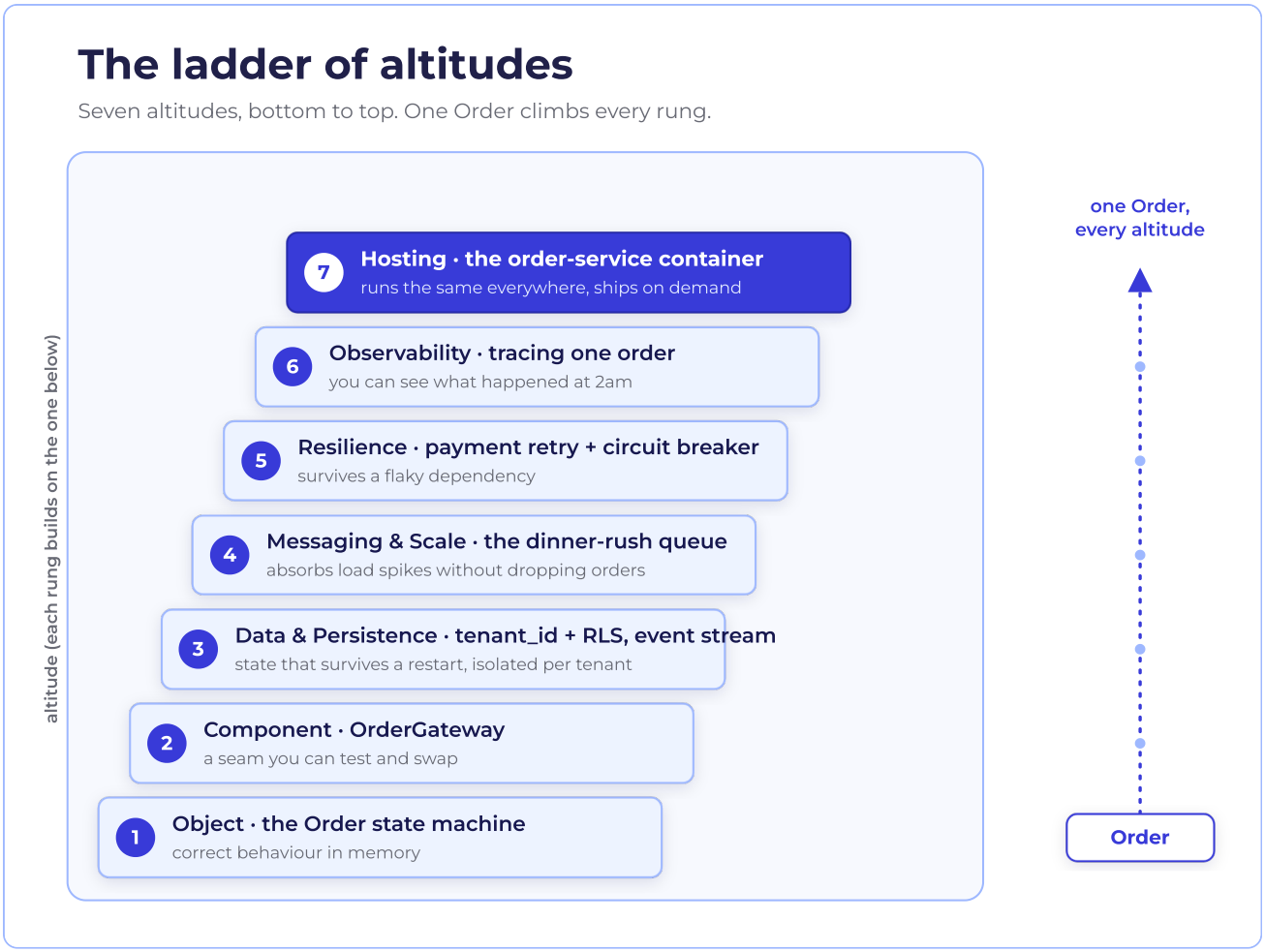
Maintain what you ship

The north star sits underneath the selection criterion: **maintain what you ship.** Production-ready is not a maximum you chase; it's a bar you clear and then keep clearing, with the same small team, a year from now, when the person who wrote the clever bit has moved on.

That single constraint reorders the whole catalogue. It promotes the boring, durable patterns (a health endpoint on the order service, a retry with backoff on the payment call, a thin `OrderGateway` over a stored procedure) and demotes the impressive, heavy ones (full event sourcing for a menu, a saga before you have a distributed transaction). The patterns that survive contact with a small team are the ones you can still reason about when you're tired and the incident channel is loud at the dinner rush.

> *Production-ready is a bar you clear, not a maximum you chase.*

A finite set, then, scored by what it buys you against what it costs to carry, held to one rule: you have to be able to maintain it. That's the stack. The rest of Part I sets up how to read it, and the rest of the book climbs it.



The set only makes sense as a ladder, so that's where we start: seven altitudes, each with its own job, and a tax for climbing too high too soon.

The Ladder of Altitudes

Seven altitudes from the object to the container, and the discipline that keeps you from climbing past your team's reach.

The Pareto stack is a set, but a set you read in any order is just a pile. The patterns earn their power from where they sit relative to each other. So the book gives them an order, and the order is a ladder.

The seven rungs

Start at the code and rise to where it runs. Each rung answers a different question, and each assumes the rung below it is already solid. To keep the rungs from arriving as disjointed snippets, the book climbs all seven through one familiar app: a food-delivery marketplace with three sides, the customer, the restaurant (your tenant), and the courier. Each altitude solves a real feature of that same app, so you watch the patterns compose instead of meeting them one at a time and forgetting them.

Object. How do you organise code inside a single boundary so it bends instead of breaking when requirements change? This is the Gang of Four altitude: Strategy, Adapter, Decorator, and the handful of others that survived contact with thirty years of real codebases. Here the `Order` moves through its lifecycle as a State machine and `CourierMatchingStrategy` swaps nearest for fastest without an `if` ladder.

Component. How do you structure one service so it stays testable and changeable? Dependency injection as the spine, a thin gateway to your data, the domain core kept clean of I/O. This is where `OrderGateway` sits over a stored procedure and an `OrderPlaced` domain event notifies the kitchen.

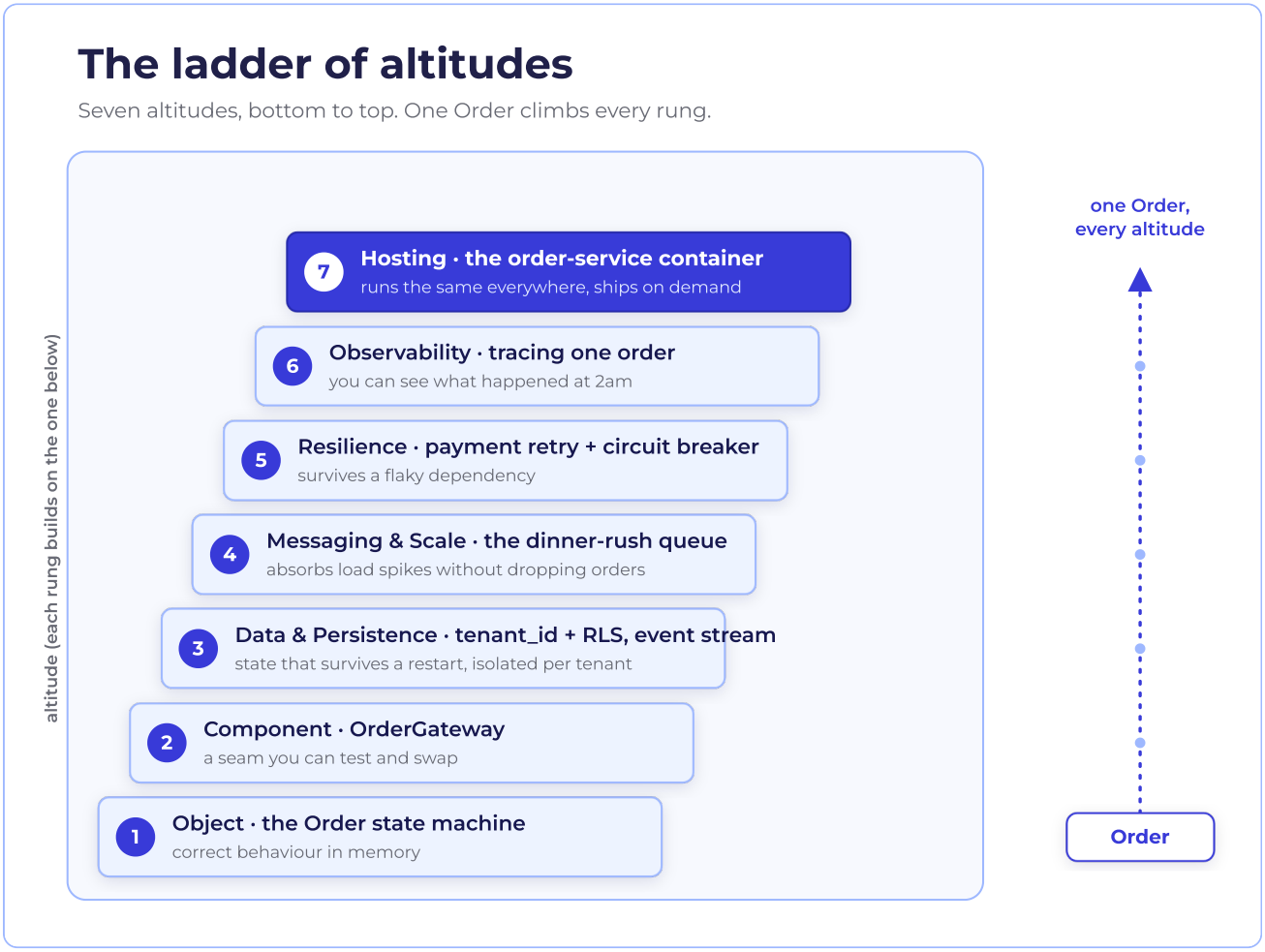
Data and persistence. How do you store state so it survives, scales for reads, and tells you who changed what? Multitenancy, optimistic concurrency, the event-sourcing combo for the rare case that needs it. Restaurants are the tenants, sharing one schema behind `tenant_id` and row-level security.

Messaging and scale. How do you move state between services without losing work when one of them falls over? Queues, competing consumers, the outbox that ties a publish to a database transaction. The dinner rush is the spike; couriers compete to pick up the next assignment off the queue.

Resilience. How do you stay up when the things you depend on don't? Retry, circuit breaker, bulkhead, timeout. The patterns that turn "the payment provider hiccuped" into a non-event, plus the idempotency that stops a retry charging the customer twice.

Observability. How do you see inside a running system, and how does it wake someone when it's genuinely in trouble? Structured logs keyed on `order_id`, the golden signals at rush, traces that thread one order across order, payment, and courier.

Hosting. How does all of it run somewhere without marrying one cloud? The container as the portability boundary, state pushed outside it, the same order-service image on GCP (AWS, Azure).



> *Object to container. Below the bottom rung there is only syntax; above the top one there is only somebody else's billing console.*

How to read this

You can read the ladder top to bottom in one sitting, and the chapters that follow are built for exactly that. But the structure is also a reference. The day a dinner rush takes the order service down, you go to the messaging rung. The week you onboard the second restaurant, you go to persistence and read about multitenancy before you reach for it.

Every pattern in Part II is taught the same way, four beats, no exceptions. The problem in one sentence. The shape, in six to fifteen lines of C#. What it buys you in production. And the skip-if, the honest signal that a small team shouldn't reach for this one yet. The fourth beat is the one most catalogues leave out, and it's the one that protects you.

The over-engineering tax

Here is the counter-discipline, and it runs underneath every chapter that follows. A pattern is not free. You pay for it once when you add it and again every time someone has to read the code, debug it at 3am, or change it without breaking the three things it quietly touches. That recurring cost is the over-engineering tax.

The catalogues never mention it. They describe what a pattern solves and stop, as if adoption were a one-time purchase. It isn't. Every pattern you adopt is one you carry forever, or until you do the unglamorous work of ripping it out. A five-person team with no platform group behind it has a hard ceiling on how much architecture it can carry. Past that ceiling, the next clever abstraction doesn't buy resilience. It buys a liability with good intentions.

> *A pattern your team can't maintain is not an asset on the books. It's a debt you took on to look senior.*

The tax shows up most plainly in patterns that arrive early. Think of event sourcing on the menu-admin screen, a four-table CRUD job where a restaurant edits prices and toggles items in stock. Or a message bus wired between two services that could have shared a function call, and the Kubernetes cluster standing guard over a single off-peak analytics worker that fits in one container and idles most of the day. None of these are wrong patterns. They're right patterns filed under the wrong year, and the cost lands every day until someone notices.

When not to reach

The deferring question is simple, and you should ask it out loud before every adoption: *what breaks if I don't add this yet?* If the honest answer is "nothing breaks for a year," you have your answer. Boring is a position, not a failure. Run on one restaurant until a second one forces multitenancy. Keep a clean monolith until something it can't carry forces microservices. And leave a stored procedure behind `OrderGateway` until the day you actually need an event log to answer who changed what.

This isn't a licence to ship naively. It's the opposite. Knowing a pattern well enough to skip it on purpose is harder than reaching for it on reflex, because the reflex feels like progress and the restraint feels like doing less. The skip-if attached to every pattern is your shortcut to that judgement, distilled. Chapter 11 gives the discipline its full treatment, altitude by altitude. For now, plant the rule and let it sit.

Production-ready is a bar you clear, not a maximum you chase. The team that clears it with the fewest moving parts wins, because they're the team still able to change the system in a year. Maximum production-readiness per unit of team capacity, the criterion from Chapter 1, cuts both ways. It rewards the right pattern and it punishes the premature one with exactly the same arithmetic.

The bottom rung is the code itself. Start there, where a pattern is still just a few lines you can read aloud.

Object Level: The Patterns That Earn Their Keep

The Gang of Four shipped twenty-three patterns. A working team leans on a third of them here, plus two the catalogue never named. This chapter is that working set.

This is the bottom rung. Before you organise a service, before you choose a database, you organise the code inside a single class or a small cluster of them. That is what the object-level patterns do: they give shape to behaviour that would otherwise sprawl into branching and copy-paste.

The Gang of Four catalogue (Gamma, Helm, Johnson, Vlissides, *Design Patterns*, 1994) is the canonical source here, and it is also where over-reach starts. Twenty-three patterns is a curriculum, not a toolkit. The patterns below are the ones a small team reaches for again and again. Most come from the Gang of Four; two, Money and the Null Object, come from outside it, because the object altitude was never only theirs. Each gets the four beats we hold for every pattern in Part II: the problem in a sentence, the shape, what it buys you in production, and the skip-if.

We anchor them all to one running example, carried through the whole book: a food-delivery marketplace with three sides, the customer who orders, the restaurant that cooks (the tenant), and the courier who delivers. The `Order` you meet here as a State machine is the same `Order` you will see survive a flaky payment gateway in the resilience chapter and get assembled end to end in the reference service. Watch the patterns compose around it.

> *Patterns are vocabulary, not architecture. Knowing the word "Strategy" does not design anything. It just means two engineers can name the same thing.*

What the compiler already gave you

Some GoF patterns won so completely that the language absorbed them. You will not find them taught here, because teaching them would mean teaching you to hand-roll something the runtime hands you for free.

Observer is the clearest case. In 1994 you wrote a subject that held a list of observers and looped over them on change. In C# that is an `event`, or `IObservable<T>` if you want the reactive shape. Iterator is `IEnumerable<T>` and `yield return`; you have never written a `MoveNext()` by

hand and you never should. Builder, for most uses, collapsed into object initialisers, collection initialisers, and `with` expressions on records.

When the compiler gives you the pattern, the pattern is not a design decision any more. It is syntax. Reaching for the textbook version signals you learned the catalogue but not the language.

Strategy

The problem. You have one operation with several interchangeable algorithms, and the choice belongs to the caller or the configuration, not to a branch buried in the method. When an order is ready, you have to pick a courier, and "pick" means different things in different cities: the nearest courier, the one who will get there fastest given traffic, or the cheapest to dispatch.

```
public interface ICourierMatchingStrategy
{
    Courier? Match(Order order, IReadOnlyList<Courier> available);
}

public sealed class NearestCourier : ICourierMatchingStrategy
{
    public Courier? Match(Order order, IReadOnlyList<Courier> available) =>
        available.MinBy(c => c.DistanceTo(order.Restaurant));
}

public sealed class FastestCourier : ICourierMatchingStrategy
{
    public Courier? Match(Order order, IReadOnlyList<Courier> available) =>
        available.MinBy(c => c.EtaTo(order.Restaurant));
}
```

The dispatcher holds an `ICourierMatchingStrategy` and never knows which one it got. New matching rule (cheapest, highest-rated, batched), new class, no surgery on the old ones.

What it buys you in production. Strategy is the antidote to the growing `switch`. Each algorithm is isolated, unit-testable on its own, and addable without touching the others. That last property is the Open/Closed Principle, the "O" in Robert C. Martin's SOLID (*Agile Software Development: Principles, Patterns, and Practices*, 2002): open to extension, closed to modification.

It also makes behaviour configurable: the same binary runs nearest-courier matching in a dense city and fastest-courier in a sprawling one, decided per tenant at startup.

Skip-if. You have one algorithm, or two that will never grow past two. A single `if` is not a design smell; it is an `if`. Strategy earns its keep when the set of behaviours is open-ended or selected at runtime, not when you are dressing up a boolean.

> ***In the front-end.*** *The customer's restaurant list needs the same shape. Sort by delivery time, rating, or price, and filter by cuisine or dietary tag, each a small strategy the React component selects from a dropdown rather than a nest of conditionals in the render path.*

Adapter

The problem. You depend on an interface your code expects, but the thing you actually have speaks a different one, often because it is a third-party SDK or a legacy class you cannot change. Charging a customer means calling a payment provider, and Stripe, Adyen, and a local processor each expose their own shape.

```
public interface IPaymentProvider
{
    Task<PaymentResult> Charge(Money amount, Card card, CancellationToken ct);
}

// The vendor SDK exposes its own shape. Wrap it.
public sealed class StripePaymentProvider(StripeClient client) : IPaymentProvider
{
    public async Task<PaymentResult> Charge(Money amount, Card card, CancellationToken ct)
    {
        var intent = await client.PaymentIntents.CreateAsync(
            new PaymentIntentCreateOptions
            {
                Amount = amount.Cents, Currency = amount.Currency, PaymentMethod = card.Token
            }, cancellationToken: ct);
        return new PaymentResult(intent.Status == "succeeded", intent.Id);
    }
}
```

Your order code talks to `IPaymentProvider`. The vendor's surface lives behind the adapter and nowhere else.

What it buys you in production. The vendor's shape stops at the seam. When you swap Stripe for Adyen, or add a regional processor a new market demands, you write one new adapter; the rest of the order flow never learns the supplier changed. It also makes the dependency mockable, because your code depends on your interface, not the SDK's. One altitude up this same instinct grows into the Anti-Corruption Layer, which insulates not one call but a whole model.

Skip-if. The external interface is already clean and you only call it in one place. An adapter you write once and never benefit from is a layer of indirection charged against the reader's attention. Wrap things you swap, mock, or want to quarantine.

> ***In the front-end.*** *The checkout screen wraps the maps SDK and the payment SDK the same way. A thin* `IMapView` *or* `IPaymentSheet` *around Google Maps or Stripe Elements keeps your components from binding to a vendor's prop names you do not control.*

Decorator

The problem. You want to add behaviour to an object without modifying it or subclassing it for every combination. An order's price is rarely the base total. Surge multiplies it at peak hours, a promo code knocks money off, and a loyalty tier shaves a percentage, and any of the three can apply at once.

```
public sealed class SurgePricing(IPricing inner, ISurgeClock clock) : IPricing
{
    public Money PriceFor(Order order) =>
        inner.PriceFor(order) * clock.MultiplierAt(order.PlacedAt, order.Restaurant);
}

public sealed class PromoPricing(IPricing inner, PromoCode code) : IPricing
{
    public Money PriceFor(Order order) => code.ApplyTo(inner.PriceFor(order));
}
```

Each decorator implements the same `IPricing` it wraps, so they stack: loyalty around promo around surge around the base total, each unaware of the others.

What it buys you in production. The adjustments stay out of the core total. Base pricing computes the sum of items and nothing else; surge, promo, and loyalty live in their own wrappers you compose at registration, in an order you control. Add a "first order free delivery" rule next month as one more decorator, no edit to the four that already work. This is the in-process cousin of the Proxy and the middleware pipeline you will meet at the component altitude.

Skip-if. You only ever need one adjustment and it will never stack. At that point an extension method or a single inline check is less ceremony. Decorator pays off when behaviours

combine, because the alternative is a subclass explosion for every permutation of surge, promo, and loyalty.

> ***In the front-end.*** *The React equivalents are hooks and higher-order components. A `withRetry` wrapper around a fetch hook, or a `useSurgeBanner` that layers onto a price display, adds behaviour to a component without rewriting it.*

Command

The problem. You want to turn a request into an object, so you can queue it, log it, retry it, or undo it, rather than calling a method and losing the call the instant it returns. Placing an order, cancelling one, adding an item to a cart: each is an intent worth capturing as a value, not a method call that vanishes when it returns.

```
public interface ICommand
{
    Task Execute(CancellationToken ct);
}

public sealed record PlaceOrder(Guid CartId, Guid CustomerId) : ICommand
{
    public Task Execute(CancellationToken ct) =>
        OrderService.Place(CartId, CustomerId, ct);
}
```

Once `PlaceOrder` is a value, you can put it on a queue, write it to a log, hand it to a worker, or hold it for a retry. The invoker and the receiver stop knowing about each other.

What it buys you in production. Command is the object-level seed of everything in the messaging altitude. A `PlaceOrder` you can serialise is a request you can level off a queue at dinner rush, hand to a competing consumer, or replay after a failure. It also gives you an audit trail for free, because the command *is* the record of what the customer asked for.

Skip-if. The work runs inline and synchronously, completes immediately, and never needs queuing, logging, or undo. Wrapping a direct method call in a command object then buys you

nothing but a class. Reach for it when the request has to outlive the call stack.

> ***In the front-end.*** *The cart is the natural home. Model "add item", "remove item", "apply promo" as command objects and undo falls out for free: keep the executed commands on a stack and an "undo" button reverses the last one. Optimistic UI gets cleaner too, because each command knows how to roll itself back when the server rejects it.*

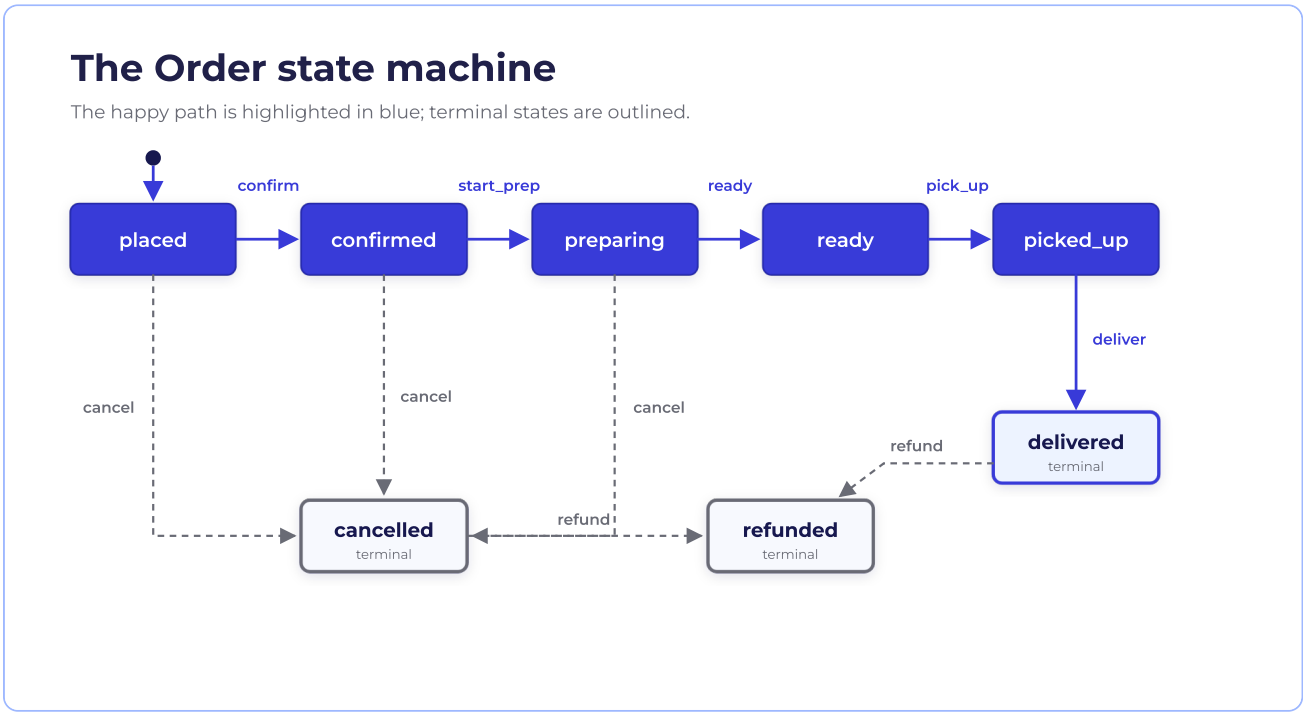
State

The problem. An object behaves differently depending on which mode it is in, and the mode logic has spread into flag checks scattered across every method. An order moves through a known lifecycle, placed, confirmed, preparing, ready, picked up, delivered, with cancellation legal only at certain points, and a pile of boolean flags cannot enforce that.

```
public interface IOrderState
{
    IOrderState Confirm();
    IOrderState Cancel();
}

public sealed class Placed : IOrderState
{
    public IOrderState Confirm() => new Confirmed();
    public IOrderState Cancel()  => new Cancelled();
}

public sealed class Confirmed : IOrderState
{
    public IOrderState Confirm() => throw new InvalidOperationException("already confirmed");
    public IOrderState Cancel()  => new Cancelled();   // refund path
}
```

Each state owns its own legal transitions. An illegal move, like cancelling an order the courier has already collected, fails loudly in one obvious place instead of slipping through a missed flag.

What it buys you in production. The legal transitions live in the type system, so the illegal ones are hard to commit by accident. An order cannot jump from `Placed` straight to `Delivered`, because no method offers that move. This matters most for anything with a lifecycle, and the order is the spine of the whole app: a wrong transition is a customer charged for food that never came, or a refund on a meal already eaten.



Skip-if. Two states and one transition. A boolean is fine and a state machine is theatre. State earns its place when transitions are constrained, several, and worth enforcing, which the order lifecycle plainly is.

State also ties forward. If you record each transition rather than just the current mode, the list of transitions *is* an event log, and you are one step from the Event Sourcing story in the persistence chapter, where `OrderPlaced`, `OrderConfirmed`, and the rest become the write model. State changes are events you have not written down yet.

> *A lifecycle modelled as states is a bug class deleted. The illegal transition stops compiling.*

> ***In the front-end.*** *The customer's order-tracking screen is the same machine on the client. A small state hook drives the "preparing → on its way → arriving" UI, and modelling it as explicit states stops the screen from showing "courier 2 minutes away" for an order that was cancelled.*

Proxy

The problem. You want to control access to an object: defer its creation, check a permission, throttle the call, or stand in for something that lives across a network, without the caller knowing any of that is happening. A restaurant's menu carries a photo for every item, and loading all of them up front is wasted bandwidth for images the customer may never scroll to.

```
public sealed class LazyMenuImage(string blobKey, IBlobStore store) : IMenuImage
{
    private byte[]? _bytes;

    public async Task<byte[]> Load(CancellationToken ct) =>
        _bytes ??= await store.Fetch(blobKey, ct);   // fetched once, on first use
}
```

The proxy presents the same `IMenuImage` as the real thing and defers the fetch until someone asks. The caller sees an image and nothing else.

What it buys you in production. Proxy puts a policy (lazy loading, auth, rate limiting) in front of an object without polluting that object. The same shape stands in for a remote restaurant service: a local proxy holds the address, opens the connection on first call, and the order code never knows whether the menu came from memory or a network hop. The structure looks like Decorator, and the code often does too; the intent is the difference. Decorator adds behaviour, Proxy controls access.

Skip-if. No access concern to enforce and no remote call to stand in for. A proxy that only forwards is dead weight. Reach for it when there is a real gate: a permission, a quota, a deferral, a network hop.

This is the same idea you will meet at the hosting altitude as the Ambassador and Sidecar. A proxy controls access to an object in-process; an ambassador controls access to a remote service out-of-process. One pattern, two altitudes.

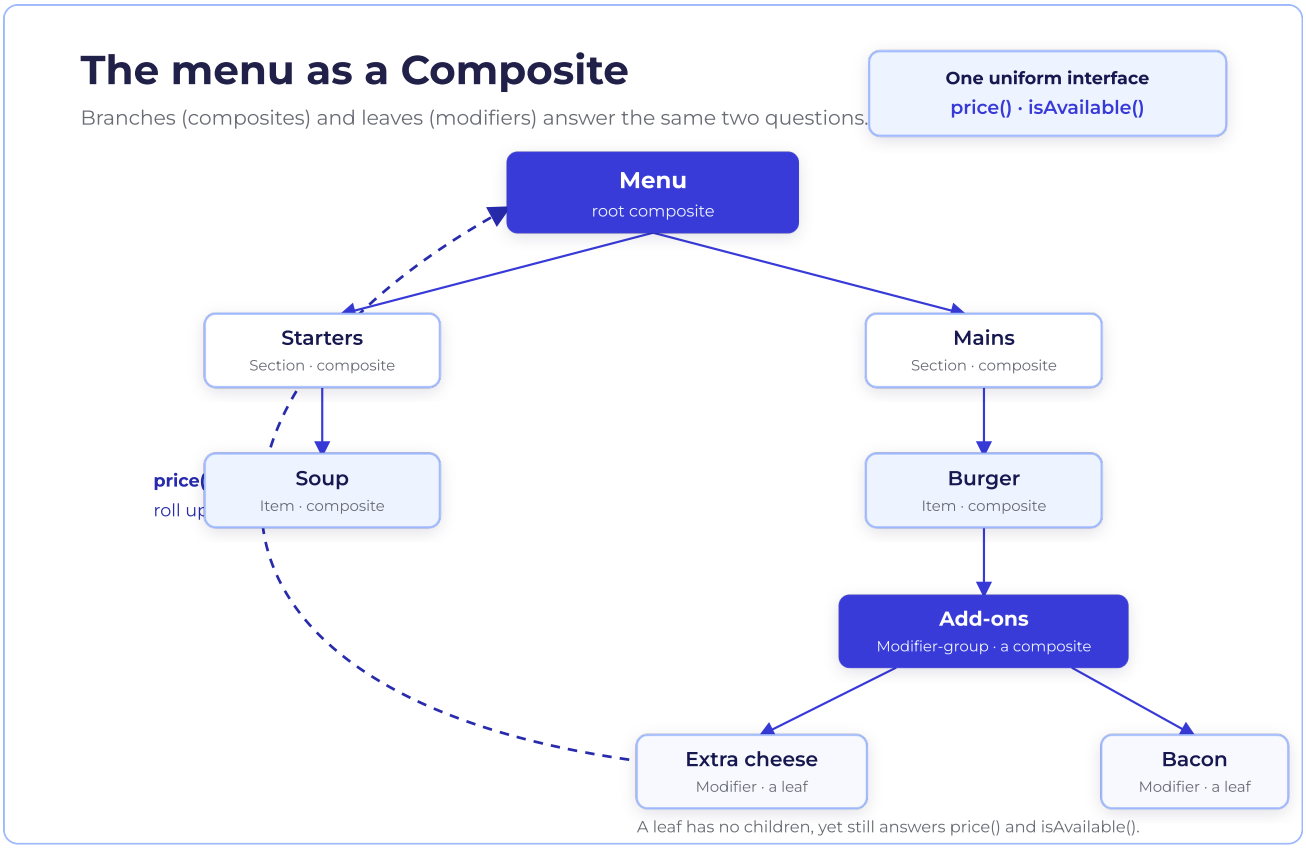
> ***In the front-end.*** *Menu images lazy-load the same way: an* `<img loading="lazy">` *or an intersection-observer wrapper fetches the photo only as it scrolls into view. Optimistic UI is a proxy too, standing in with a predicted result while the real call is in flight.*

Composite

The problem. You have a tree of objects where a single thing and a group of things should be treated the same way, and you do not want every caller writing "if it is a leaf do this, if it is a branch recurse." A menu is exactly this tree: sections hold items, items hold modifier-groups (size, extras), modifier-groups hold modifiers, and you want one question, "what does this cost and is it available?", answered uniformly whether you ask a single modifier or a whole section.

```
public interface IMenuComponent
{
    Money Price();
    bool  Available();
}

public sealed class Modifier(Money price, bool inStock) : IMenuComponent
{
    public Money Price()    => price;            // leaf
    public bool  Available() => inStock;
}

public sealed class MenuNode(IReadOnlyList<IMenuComponent> children) : IMenuComponent
{
    public Money Price()    => children.Aggregate(Money.Zero, (sum, c) => sum + c.Price());
    public bool  Available() => children.All(c => c.Available());   // branch rolls up
}
```

A leaf (`Modifier`) and a branch (`MenuNode`) share `IMenuComponent`, so price sums and availability rolls up the tree with no special-casing. A section is unavailable when any required child is out of stock, and its price is just the sum below it, computed by the same two methods at every level.



What it buys you in production. One interface for the whole menu means the pricing engine, the renderer, and the availability check each walk the tree without knowing its depth. Add a new node type, a "combo" that bundles items, and it slots in as another `IMenuComponent` with no edits to the walkers. The recursion lives in the structure, not smeared across every caller (GoF).

Skip-if. The structure is flat, or only ever one level deep. A list of items with no nesting does not need Composite; a `foreach` reads better than a tree. It earns its keep when the hierarchy is genuinely recursive and you want leaf and branch handled alike, which a real menu with nested modifiers is.

> ***In the front-end.*** *This is the component tree itself. Rendering the nested menu is a recursive component that draws a section or a single item with the same code, and the React reconciler is a Composite walking your elements. The pattern you reach for to render the menu is the pattern the framework already runs underneath it.*

Money

The problem. Every price in this chapter has a currency and a rounding rule, and the moment you store it as a bare `decimal` you have scattered both across the codebase. One method rounds half-up, another truncates, a third adds a euro subtotal to a dollar delivery fee and nobody notices until a customer who orders in two markets sees a total that does not add up. Money is not a number. It is an amount and a currency travelling together, and when you split them you get the bugs that surface only in production, in another timezone, in a currency your tests never touch.

```
public readonly record struct Money(long Cents, string Currency)
{
    public static readonly Money Zero = new(0, "");   // additive identity; adopts the other

    public static Money operator +(Money a, Money b)
    {
        if (a == Zero) return b;
        if (b == Zero) return a;
        if (a.Currency != b.Currency)
            throw new InvalidOperationException($"cannot add {a.Currency} to {b.Currency}");
        return a with { Cents = a.Cents + b.Cents };
    }

    public static Money operator *(Money m, decimal factor) =>
        m with { Cents = (long)Math.Round(m.Cents * factor, MidpointRounding.ToEven) };
}
```

This is the `Money` you have already been reading: the `amount.Cents` the payment adapter handed to Stripe, the `* clock.MultiplierAt(...)` in the surge decorator, the `Money.Zero` seed the menu Composite summed into. One immutable value type owns the cents, the currency, and the single rounding convention the whole system agrees on.

What it buys you in production. A currency mismatch stops being a silently wrong total three calls later and becomes a loud exception on the line that caused it. Rounding lives in one place, so the surge multiplier and the promo discount cannot disagree about a half-cent. The type also makes a whole bug class structurally hard: you cannot hand a raw `1995` to something expecting `Money`, so you can never lose track of whether that meant dollars, cents, or pounds.

Value objects (Fowler, *Patterns of Enterprise Application Architecture*, 2002, the Money pattern) pay off anywhere a primitive hides rules, and money hides the most expensive ones in the system.

Skip-if. A genuinely single-currency product that rounds in one obvious place can live with a `decimal` and a documented convention. But the type costs a dozen lines and the bugs it prevents reach an accountant, so the bar to skip it sits higher than it first looks. The day a second currency arrives, the `decimal` version is a refactor across every price you have; the `Money` version is a new string.

> *A bare `decimal` for money is a currency bug waiting for its second market. A type that carries its own currency never has that bug.*

Null Object

The problem. A collaborator is optional, so half your methods guard it before they use it. The customer who turned off push notifications, the order with no promo, the tenant with no custom branding: each leaves a reference that might be set or might be `null`, and every call site pays the tax of checking first. Scatter enough `if (notifier is not null)` through the order flow and the real work disappears behind the defending.

```
public interface ICustomerNotifier
{
    Task Notify(OrderId order, string message, CancellationToken ct);
}

public sealed class PushNotifier(IPushService push) : ICustomerNotifier
{
    public Task Notify(OrderId order, string message, CancellationToken ct) =>
        push.Send(order, message, ct);
}

public sealed class NullNotifier : ICustomerNotifier      // does nothing, on purpose
{
    public static readonly NullNotifier Instance = new();
    public Task Notify(OrderId order, string message, CancellationToken ct) => Task.Completed
}
```

A customer who opted out is handed a `NullNotifier`; everyone else gets the real one. The status-update code calls `notifier.Notify(...)` and never asks which it is holding.

What it buys you in production. The null check is gone, because there is no null. The do-nothing path becomes one named, tested class instead of a branch copied to every call site, and a reader sees intent: this customer is deliberately not notified, not a missing dependency someone forgot to wire. The Null Object (Woolf, *Pattern Languages of Program Design 3*, 1998) is really a Strategy whose algorithm is "do nothing," which is why it drops in exactly where a Strategy does: register it like any other implementation and the caller stays oblivious.

Skip-if. Absence that means something is not a job for a Null Object. When no courier is free, the matcher returning `null` is a real condition the caller has to handle, hold the order, try again, tell the customer, and a do-nothing stand-in would bury a decision you need to make out loud. Reach for it when the neutral behaviour genuinely is "nothing happens." Keep the null when "nothing here" is news.

> *A Null Object turns "might be there" into "always there, sometimes silent." Use it for neutral behaviour, never to hide a real nothing.*

The Singleton skip: global state wearing a pattern's name

Singleton is in the GoF catalogue. It is also the clearest over-engineering example at this altitude, and it is worth walking the trap because so many codebases still fall into it.

The classic shape is a class with a private constructor and a static `Instance` property, guaranteeing exactly one instance for the lifetime of the process. The intent sounds reasonable: some things should exist once. The cost is that you have created global mutable state and hidden a dependency. Every class that calls `Logger.Instance` now depends on `Logger` without saying so in its constructor, which means you cannot substitute it in a test, you cannot tell from the signature what a method touches, and two tests that share the instance can poison each other. Misko Hevery's line, that singletons are pathological liars, is about exactly this: the dependency is real but invisible.

The fix is not a cleverer Singleton. It is the dependency container you will meet in the next chapter, and the principle behind it is the Dependency Inversion Principle, the "D" in Robert C. Martin's SOLID (*Clean Architecture*, 2017): depend on an abstraction, hand it in, do not reach out to a global. You want one instance, so you register the type with a singleton *lifetime* and inject it:

```
services.AddSingleton<IClock, SystemClock>();
// Consumers ask for IClock in their constructor.
// One instance, an honest dependency, a trivial test double.
```

Same guarantee (one instance), opposite properties. The dependency is declared in the constructor, the lifetime is the container's job rather than the class's, and the test swaps in a fake clock without touching global state. The pattern you wanted was never "one instance." It was "one instance, injected." The container gives you that and the GoF Singleton does not.

> *"One of these" is a lifetime, not a pattern. Let the container own it.*

Honorable mentions

Four more GoF patterns sit just outside this set, useful enough to name and not quite frequent enough to teach in full.

Factory Method centralises which concrete type to create from runtime input, the `switch` that picks the right courier-matching strategy or order-state handler, kept in one place. It used to earn a top slot, but in a DI-driven codebase the container absorbs most creation: you register the types and resolve by key, or inject a `Func<MatchingMode, ICourierMatchingStrategy>`, and the hand-written factory mostly disappears. Reach for the explicit version only when the choice is too involved for a registration.

Facade puts a simple interface over a messy subsystem. You will reach for it the day a third-party library or a tangle of internal services needs a single, sane entry point, and the component-level Anti-Corruption Layer in the next chapter is Facade grown up with a domain motive.

Template Method defines the skeleton of an algorithm in a base class and lets subclasses fill in the steps. It is quietly everywhere in framework code (think a base controller or a base handler with hooks), but in application code the same job is often cleaner as a Strategy passed in, because composition outlasts inheritance.

Chain of Responsibility passes a request along a line of handlers until one of them takes it. You rarely write it by hand at the object level, because its real home is one altitude up as the request pipeline, which is exactly the Middleware pattern in the next chapter.

These patterns organise code inside a single boundary. Next we climb a rung and organise the service around them.

Component Level: Structuring One Service

The object patterns of Chapter 3 organise code inside a boundary. These organise the boundary itself: how one service stays testable, swappable, and changeable a year after you wrote it.

A service is the unit you deploy, page on, and rewrite when it rots. Its inside is where most of your maintenance time goes, so the patterns at this altitude buy you something concrete: the ability to change one corner without the whole thing shifting under you. Several earn the slot. The spine is Dependency Injection. The rest hang off it.

The selection rule holds here as everywhere. Each pattern is worth what it buys you in production, and each has a carrying cost worth naming out loud. Reach for the ones that keep a small team moving; skip the ones that just add ceremony you have to maintain forever.

Dependency Injection

The problem: a class that builds its own collaborators is welded to them, and you can't test it or swap them without surgery.

Dependency Injection inverts that. A class declares what it needs in its constructor and receives it; something else decides what to hand over. The "something else" is usually a container. In .NET that's `Microsoft.Extensions.DependencyInjection`, and you register the wiring once at startup.

```
services.AddScoped<IOrderGateway, OrderGateway>();
services.AddScoped<IMenuGateway, MenuGateway>();
services.AddSingleton<IPaymentAdapter, StripePaymentAdapter>();

// The consumer just asks for what it needs.
public sealed class OrderService(IOrderGateway orders, IPaymentAdapter payments)
{
    public Task<Order> Place(PlaceOrder cmd) => /* ... */;
}
```

What it buys you in production: every dependency becomes a seam. You swap `StripePaymentAdapter` for the provider a new market needs without touching `OrderService`. You pass a fake gateway in a unit test and place an order without ever hitting the database. Lifetimes (singleton, scoped, transient) become a deliberate decision instead of an accident,

which matters the day a singleton accidentally captures a per-request `tenant_id` and serves one restaurant's orders to another.

DI is also where the Singleton skip from Chapter 3 lands properly. You almost never need the global-access Singleton, because a container-managed singleton lifetime gives you one instance without the static field and the testing pain that comes with it.

Inversion of Control and the DI pattern are Martin Fowler's framing, set out in *"Inversion of Control Containers and the Dependency Injection pattern"* (2004), and the "D" in Robert C. Martin's SOLID, the Dependency Inversion Principle. This is the one pattern in the book with no real skip-if; below a trivial script, you want it.

> *Every dependency you inject is a seam. Every seam is a place you can change your mind later.*

Data Gateway (and stored procedures)

The problem: SQL scattered through your service couples business logic to table shapes, and the first schema change sends you hunting for query strings in twelve files.

A Data Gateway is a thin object that owns the SQL for a slice of data and exposes it behind an interface (Fowler, *Patterns of Enterprise Application Architecture*, 2002; Table Data Gateway). Callers see methods. The gateway sees connections, parameters, and stored procedures. Nothing leaks.

This is where the book states its data stance plainly. **SQL-first: a thin Data Gateway over stored procedures, not a heavy ORM.** You write the SQL, the database owns the query plan, and the gateway is fifty lines you can read in one sitting. We use Dapper for the mapping and ADO.NET underneath. We do not use Entity Framework, and the reason is the rest of this section.

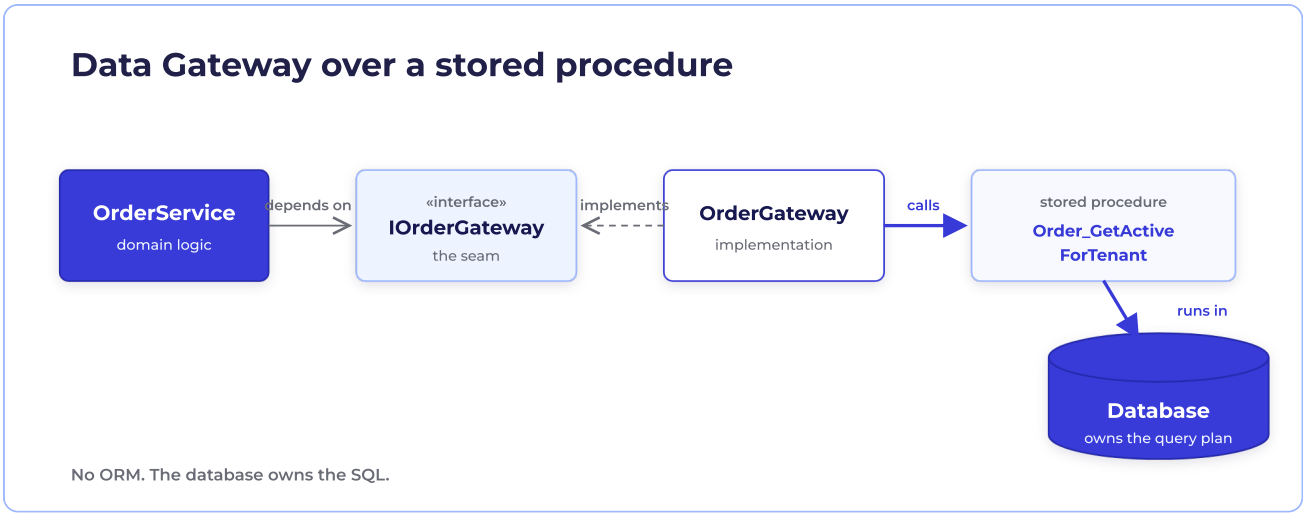
This is one of three fuller anchor listings in the book, so here it is properly: a stored function, the interface a caller depends on, and the Dapper implementation behind it. This is the `OrderGateway`, the same one the reference service in Chapter 10 wires up.

```
-- Migration, checked into source control alongside the gateway.
CREATE OR REPLACE FUNCTION order_get_active_for_tenant(
    p_tenant_id uuid,
    p_as_of     timestamptz)
RETURNS TABLE (
    order_id    uuid,
    tenant_id   uuid,
    customer_id uuid,
    status      text,
    total_minor bigint,
    currency    text,
    placed_at   timestamptz)
LANGUAGE sql STABLE AS $$
    SELECT o.order_id, o.tenant_id, o.customer_id,
           o.status, o.total_minor, o.currency, o.placed_at
    FROM   "order" AS o
    WHERE  o.tenant_id = p_tenant_id
      AND  o.status NOT IN ('delivered', 'cancelled')
      AND  o.placed_at <= p_as_of
    ORDER BY o.placed_at;
$$;
```

```
public interface IOrderGateway
{
    Task<IReadOnlyList<Order>> GetActive(Guid tenantId, DateTime asOf, CancellationToken ct);
}

public sealed class OrderGateway(IDbConnectionFactory factory) : IOrderGateway
{
    public async Task<IReadOnlyList<Order>> GetActive(
        Guid tenantId, DateTime asOf, CancellationToken ct)
    {
        await using var conn = await factory.OpenAsync(ct);

        var command = new CommandDefinition(
            "SELECT * FROM order_get_active_for_tenant(@TenantId, @AsOf)",
            new { TenantId = tenantId, AsOf = asOf },
            cancellationToken: ct);

        var rows = await conn.QueryAsync<Order>(command);
        return rows.AsList();
    }
}
```

The function returns snake_case columns, so the gateway relies on Dapper's `DefaultTypeMap.MatchNamesWithUnderscores = true`, set once at startup, to bind `total_minor` to `TotalMinor` and the rest.

What it buys you in production: the gateway is the only place that knows the `Order` table exists. Change a column and you change one file and one procedure, both of which a reviewer can read end to end. The query plan lives in the database where a DBA can tune it without a redeploy, which matters when the live-orders board reloads every few seconds at dinner rush. The interface keeps the rest of the service ignorant of SQL, so it stays unit-testable with a fake. And there is no translation layer between what you wrote and what the database runs, so what you read in the procedure is what executes.



The skip-if, stated as a worked over-engineering example: reach for a heavy ORM, and Entity Framework is the canonical one. EF buys you scaffolding speed on day one and a maintenance tax forever after. Change-tracking decides on its own what to write and when, so a stray assignment to `Order.Status` becomes an `UPDATE` you didn't ask for. LINQ-to-SQL translation is a guessing game, where a query that reads fine in C# silently materialises the whole `Order` table into memory and filters it there, or emits a join no one would write by hand. Migration drift creeps in when the model and the database disagree, and a junior staring at a generated migration cannot reason about what it will do to a table full of live orders. None of that is wrong because EF is bad software; it's wrong because it hides the database from the small team that has to operate it. A framework you can't reason about is a tax, not a shortcut.

> *The gateway is fifty lines you can read in one sitting. That is the whole point.*

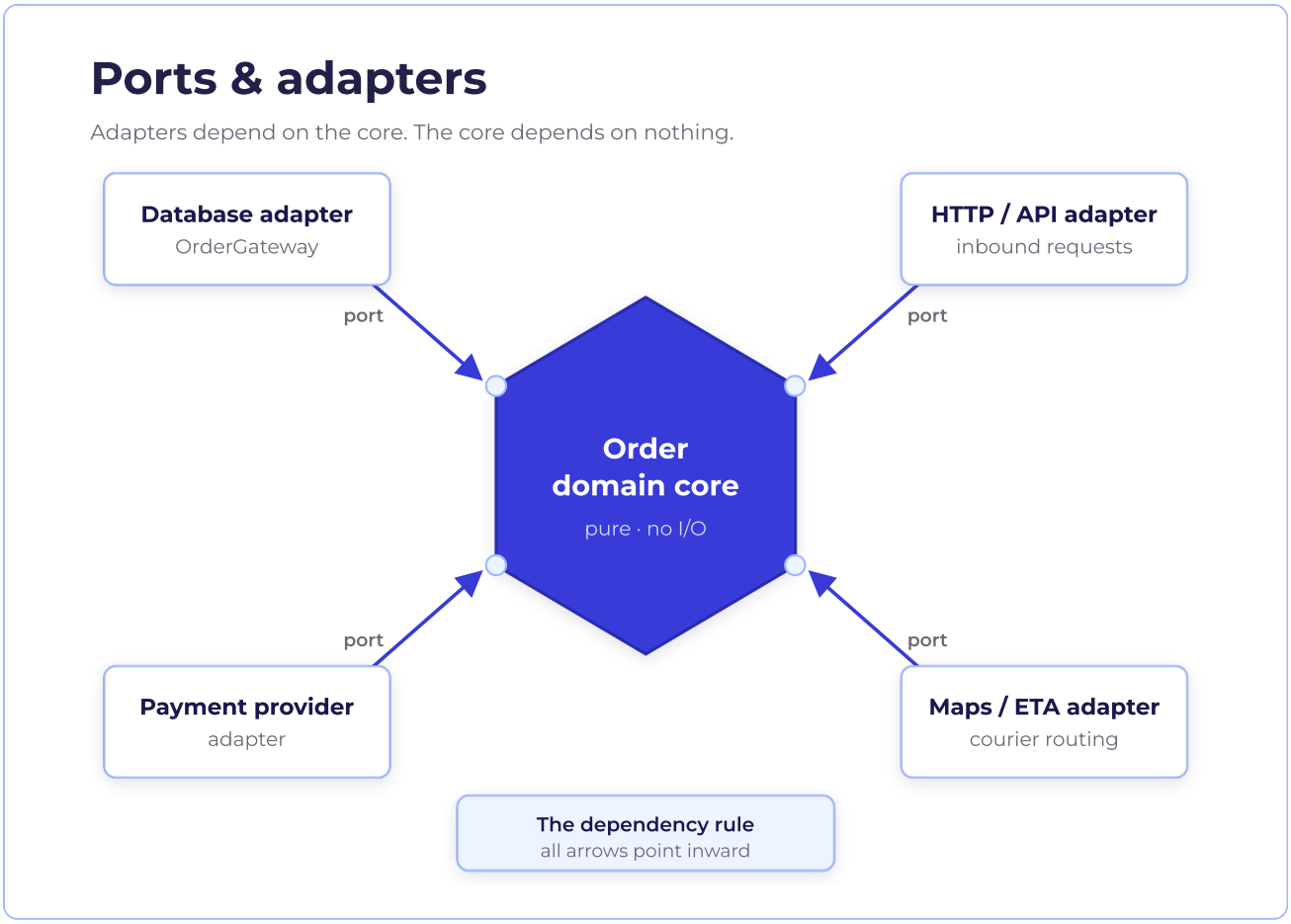
Ports and Adapters (Hexagonal)

The problem: business logic that imports the database, the message broker, and the HTTP client is a knot you can't test in isolation or move to a new dependency.

Ports and Adapters (Cockburn, 2005, also called Hexagonal) draws one line. Inside it: your order domain, which defines *ports*, the interfaces it needs (`IOrderGateway` , `IPaymentAdapter` , `IMapsService`). Outside it: *adapters*, the concrete implementations that talk to the database, the payment provider, and the mapping API. The domain depends only on its own ports. Everything technical points inward, which is Uncle Bob's Dependency Inversion again at the level of a whole service.

```
// Inside the hexagon: a port the order domain owns.
public interface IPaymentAdapter { Task<PaymentResult> Charge(Money amount, CardToken card, Ca

// Outside: an adapter the infrastructure layer provides.
public sealed class StripePaymentAdapter(StripeClient client) : IPaymentAdapter
{
    public Task<PaymentResult> Charge(Money amount, CardToken card, CancellationToken ct) => /
}
```

What it buys you in production: the order domain has no idea whether a charge goes to Stripe, Adyen, or an in-memory fake. You test order placement with fakes and never spin up a payment sandbox or a maps API. The day a new market needs a different payment provider, you write a new adapter and the domain doesn't notice.



Keep it minimal. The skip-if here is the ceremony, not the idea. You do not need the full Onion or Clean Architecture stack of four concentric layers, a separate assembly per ring, and a mapping class between every pair. For one service, the pattern is a folder of interfaces the domain owns and a folder of adapters that implement them. Add a ring only when a ring earns its keep.

Pipeline and Middleware

The problem: every request needs the same cross-cutting work (logging, auth, validation, error shaping), and copying it into each handler guarantees one of them drifts.

A pipeline composes that work into ordered stages, each wrapping the next. It's Chain of Responsibility (GoF) at the request level, and in .NET it's ASP.NET Core middleware: each component does its bit, then calls `next`. In the food-delivery API the ordered stages are tenant-resolution, auth, then logging, and order matters: you resolve the restaurant from the request before you check that the caller is allowed to act on it.

```
// Resolve the restaurant tenant from the subdomain or header, before anything else runs.
app.Use(async (ctx, next) =>
{
    var tenantId = TenantResolver.From(ctx.Request);   // e.g. host header -> tenant_id
    ctx.Items["tenant_id"] = tenantId;
    using (LogContext.PushProperty("tenant_id", tenantId))
        await next();                                  // auth and the rest run inside this s
});
```

What it buys you in production: cross-cutting concerns live in one ordered place. Tenant resolution, auth, correlation IDs, and exception-to-problem-detail mapping become stages you add once and every request inherits. Order is explicit, so you can reason about what runs before auth and what runs after, and every log line downstream already carries the `tenant_id`.

The skip-if: don't build your own pipeline abstraction for two stages you could write as two method calls. The pattern pays off when the cross-cutting list is long enough that order and reuse matter. Below that, it's indirection you maintain for no gain.

Mediator (the command/query split)

The problem: a controller that news up handlers, orchestrates them, and shapes results becomes a god-object that knows about everything downstream.

A Mediator sits between the caller and the handlers. The caller sends a *command* (do something) or a *query* (ask something) as a message; the mediator routes it to the one handler that knows how. The controller takes a `PlaceOrder` command off the wire and hands it to the mediator, and a single handler owns placing the order. In .NET this is commonly the MediatR library (Jimmy Bogard). Note the two senses of the word: GoF's Mediator is an object-level pattern for decoupling colleagues, and the application-level Mediator here is the request-routing version. Same idea, larger altitude.

```
public record PlaceOrder(Guid TenantId, Guid CustomerId, IReadOnlyList<LineItem> Items)
    : IRequest<OrderId>;

public sealed class PlaceOrderHandler(IOrderGateway orders, IPaymentAdapter payments)
    : IRequestHandler<PlaceOrder, OrderId>
{
    public Task<OrderId> Handle(PlaceOrder cmd, CancellationToken ct)
        => /* validate, charge, persist, return the new OrderId */;
}
```

What it buys you in production: handlers become small, single-purpose, and independently testable, and the command/query split makes intent obvious in the type. The `PlaceOrder` write and the live-orders read stop sharing a fat service. Pipeline behaviours (validation, logging, transactions) can wrap every handler in one place, which is the same composition idea as middleware applied to messages rather than HTTP.

The skip-if: a small service with a handful of endpoints doesn't need a message bus between its own classes. Reaching for Mediator there adds a layer of indirection and a stack trace that's harder to follow, for the price of decoupling you don't have a problem with yet. It earns its place when the handler count grows or when you want cross-cutting behaviour around every operation.

Anti-Corruption Layer

The problem: a messy external API or a legacy system leaks its model into yours, and soon your clean domain is shaped by someone else's bad decisions.

An Anti-Corruption Layer (Evans, *Domain-Driven Design*, 2003) is a translation boundary. It's the higher-altitude cousin of the Adapter from Chapter 3: where an Adapter reshapes one interface, an ACL insulates your whole model from a foreign one, translating their concepts into yours at the edge and never letting their vocabulary inside. The food-delivery case is a restaurant still running a decade-old point-of-sale system: you have to send it orders, but you do not want its `TicketRecord` and its single-character status codes leaking into your domain.

```
public sealed class LegacyPosTickets(ILegacyPosClient pos) : IKitchenTickets
{
    public async Task<KitchenTicket> Fetch(OrderId id, CancellationToken ct)
    {
        var rec = await pos.GetTicket(id.Value.ToString(), ct);  // their model: TicketRecord
        return new KitchenTicket(id, rec.LineText,               // ours: a clean domain type
            accepted: rec.State == "P");
    }
}
```

What it buys you in production: the POS vendor's breaking changes stop at the boundary. When the legacy system renames a field or returns a new status code, you fix one translator, not every call site that places or tracks an order. Your domain stays expressed in your language, which keeps it readable to the people who maintain it.

The skip-if: don't build an ACL around a dependency you trust and control. If the external model is clean and stable, a plain Adapter or a direct call is enough. The ACL earns its cost specifically when the other side is messy, legacy, or out of your control, like a restaurant's ageing POS, and the translation is worth maintaining to keep that mess contained.

Domain Events

The problem: placing an order grows a tail of side effects (notify the kitchen, reserve stock, send the customer a confirmation), and stuffing them all into the place-order method couples things that have nothing to do with each other.

A Domain Event records that something meaningful happened in the domain (`OrderPlaced` , `OrderCancelled`), and interested handlers react to it (Evans, *Domain-Driven Design*; Martin Fowler's bliki, "Domain Event"). The originating code raises the event and moves on; it doesn't know or care who's listening. In .NET these are often dispatched in-process as MediatR notifications.

```
public sealed record OrderPlaced(OrderId OrderId, Guid TenantId, DateTime PlacedAt) : INotifi

// One handler among several; each is independent.
public sealed class NotifyKitchenOnOrderPlaced(IKitchenTickets kitchen) : INotificationHandle
{
    public Task Handle(OrderPlaced e, CancellationToken ct) => kitchen.Open(e.OrderId, ct);
}
```

What it buys you in production: the side effects of an order decouple from the act of placing it. Notifying the kitchen, reserving stock, and sending the confirmation become three independent handlers, and you add a fourth (a courier pre-alert, an analytics event) by adding a handler, without editing the code that places the order. It also plants the seed for the messaging altitude: an in-process `OrderPlaced` is one small step from a published integration event once kitchen and courier-assignment genuinely live in other services.

The skip-if: in-process events are still synchronous code with a less obvious call graph. If a side effect always runs and always belongs to the operation, just call it directly; the indirection only pays off when the list of reactions is open-ended or genuinely independent. Save the cross-process version for Chapter 6, where the network and the `OrderPlaced` Pub/Sub fan-out make the decoupling real.

Specification

The problem: a business rule like "this restaurant is open and within delivery range" gets retyped wherever it is needed, in the search query, in the validation that rejects an out-of-range order, in the UI that greys a restaurant out, and the three copies drift until they disagree about what "deliverable" even means.

A Specification packages that rule as a small object with one method, `IsSatisfiedBy` , that you can name, test, and pass around (Evans, *Domain-Driven Design*, 2003; Evans & Fowler, *"Specifications,"* 2002). The payoff is that specifications combine: `And` , `Or` , and `Not` are

themselves specifications that hold other specifications, which is the Composite from Chapter 3 doing the same job one altitude up, a tree of rules handled through one interface.

```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T candidate);
}

public sealed class IsOpen : ISpecification<Restaurant>
{
    public bool IsSatisfiedBy(Restaurant r) => r.OpenNow();
}

public sealed class WithinRange(Location to) : ISpecification<Restaurant>
{
    public bool IsSatisfiedBy(Restaurant r) => r.DistanceTo(to) <= r.MaxDeliveryKm;
}

public sealed class And<T>(ISpecification<T> left, ISpecification<T> right) : ISpecification<
{
    public bool IsSatisfiedBy(T c) => left.IsSatisfiedBy(c) && right.IsSatisfiedBy(c);
}
```

One rule, composed once and reused: `var deliverable = new And<Restaurant>(new IsOpen(), new WithinRange(here));` now drives the search filter and the order validation from the same object.

What it buys you in production: the definition of "deliverable" lives in one place, tested on its own, and the search, the validation, and the UI all ask the same object instead of three predicates that quietly diverge. A new rule is a new small class; a new combination is an `And` of two you already have, with no edit to either. Because the combinators are a Composite, the rule tree reads the same whether it is one clause or six.

The skip-if: for a SQL-first service, a rule that only ever runs as a database query is usually better written as the predicate in the stored procedure, where the query planner can use it. Specification earns its keep when the same rule has to run in more than one place, an in-memory check and a query and a validation, or when rules combine at runtime. When a rule has a single home and runs once, skip the object and write the `if`.

Honorable mentions

Two patterns just missed the cut, useful when you want to go further.

Result / ErrorOr error handling. Returning an explicit `Result<T>` or `ErrorOr<T>` instead of throwing makes the failure path a value you have to handle, which reads well in a `PlaceOrder` handler and keeps control flow honest. It's a strong default; it missed the cut only because exceptions plus a pipeline error-handler cover most small services.

Plain Layered architecture. The oldest structure there is, presentation over application over domain over data. It's the honest default when Ports and Adapters would be ceremony. Plenty of solid services are just three well-named folders, and there's no shame in that.

A single service is the easy case. Production stores a great deal of state and has to move it without losing or corrupting it, which is the next altitude: data and persistence.

Data & Persistence

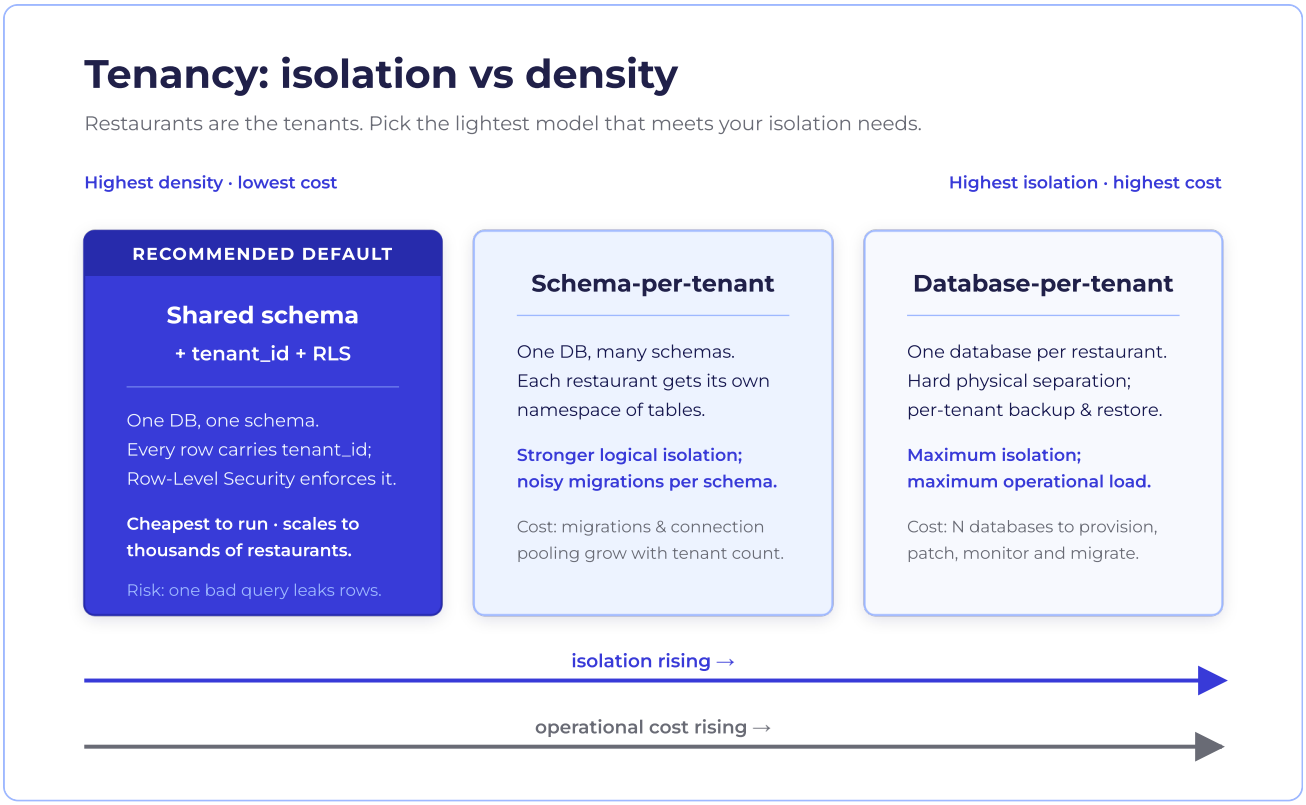
The catalogues spend a thousand pages on objects and a paragraph on the database. This is where the data actually lives, where the hard tradeoffs are, and where a small team usually over-builds first.

A single service is the easy case. You hold its state in one schema, wrap your SQL in a thin Data Gateway, and move on. Then a second restaurant signs, the menu reads outpace the order writes, someone cancels an order they shouldn't have, and the column you named `status` last quarter no longer fits the order lifecycle. The persistence altitude is where those pressures show up, and where the over-engineering tax is steepest. A handful of patterns cover almost all of it. Most of the work is knowing which one the moment actually calls for, and which one is a trap dressed as foresight.

Multitenancy

The problem: one running food-delivery platform, many restaurants, and each restaurant's orders and menu must stay theirs. **The shape:** every tenant-scoped table carries a `tenant_id` (the restaurant), and the database enforces the boundary so a missing `WHERE` clause can't leak one brand's orders into another's live-board.

The honest framing is a single axis: isolation against density. At one end, every restaurant gets its own database (maximum isolation, maximum operational cost, a migration to run N times). At the other, every brand shares one schema and a discriminator column (maximum density, one migration, the leak you have to prevent). Schema-per-tenant sits in the middle. Microsoft's *Multitenant SaaS database tenancy patterns* lays out the spectrum cleanly; the choice is which end of it your actual constraints push you toward.



Start dense. The default is shared schema, a `tenant_id` on every order and menu row, and **Postgres Row-Level Security** doing the enforcement. RLS turns "remember the `WHERE` clause" from a discipline you hope holds into a guarantee the database keeps. You set the restaurant on the connection, and every query the `OrderGateway` runs is silently filtered to that tenant.

```
ALTER TABLE "order" ENABLE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation ON "order"
    USING (tenant_id = current_setting('app.tenant_id')::uuid);
```

```
// Set once per connection, before any query runs.
await conn.ExecuteAsync(
    "SELECT set_config('app.tenant_id', @TenantId, false)",
    new { TenantId = tenantId.ToString() });
```

What it buys you in production: one database to back up, patch and migrate, and a leak that's a `CREATE POLICY` away from impossible rather than one forgotten clause away from one

restaurant reading another's orders. You escalate when a real constraint forces it. A national brand that needs menu-schema customisation, or one busy enough at dinner rush to starve the smaller restaurants, earns schema-per-tenant. Database-per-tenant is for when compliance or an enterprise contract demands physical separation, not for when it feels safer. The density you give up is paid for every day in operations.

> *Density is the default. Isolation is the exception you justify, not the posture you assume.*

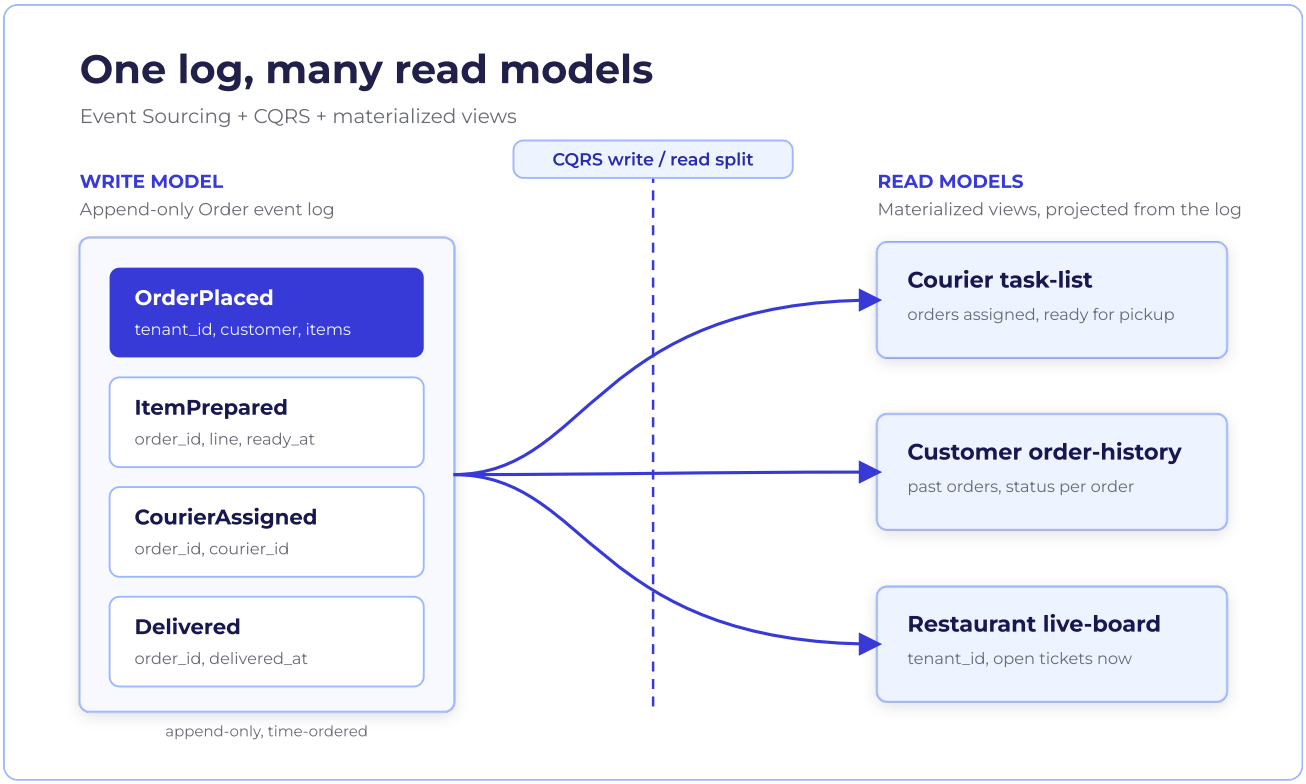
Skip-if: you have one restaurant. A single-tenant product needs none of this. No `tenant_id`, no RLS, no spectrum. Build it the day the second brand is real, not the day you imagine them.

Event Sourcing, CQRS and the Materialized View

The problem: an order's reads and writes want different shapes, and at rush they fight over the same tables. The courier wants a task-list, the customer wants order-history, the restaurant wants a live-board, and all three are projections of the same order. **The shape:** stop storing the order's current state and start storing the events that produced it, then build whatever read models you need from the log.

These three patterns are usually taught apart. Treat them as one story. An order is a lifecycle, not a row: `OrderPlaced`, `ItemPrepared`, `CourierAssigned`, `Delivered`. **Event Sourcing** is the write model. Instead of updating a row to `status = 'delivered'`, you append a `Delivered` event to an immutable log, and the order's current state is the log folded up (Fowler; popularised by Greg Young). **CQRS** is the umbrella, separating the write model from the read model so neither has to compromise for the other (Greg Young, Udi Dahan). The **Materialized View** is the read model: a denormalised projection built from the events and optimised purely for queries (Fowler; Azure Cloud Design Patterns). You build one per audience. The courier task-list, the customer order-history, the restaurant live-board.

```
// Write side: append, never update.
await _events.AppendAsync(orderId, new CourierAssigned(orderId, courierId, assignedAt));

// Read side: each projection updated as events arrive, queried directly.
// SELECT order_id, status, eta FROM courier_tasklist_view WHERE courier_id = @courierId
```



What it buys you in production: you scale reads without touching the write path, rebuild the live-board by replaying the log, and get a complete audit trail of every order for free because the log *is* the history. Add the restaurant live-board a year after the courier task-list and you replay events you already have. The write side never knows the read side exists.

Now the honest part. This is the heaviest skip-if in the book, and it is the one teams reach for most often by mistake. Event sourcing means giving up the thing every developer already knows how to do, the `UPDATE` statement, in exchange for eventual consistency between write and read, projection rebuild logic, event versioning as the order schema evolves, and a debugging story most of your team hasn't lived. The catalogue makes it sound like the mature choice. For the overwhelming majority of systems it is not.

> *Most CRUD apps must not reach for this. If a row update and a `SELECT` describe your problem honestly, that is your architecture. Say so and move on.*

Skip-if: you can't name the specific read load or audit requirement that a single normalised `order` table fails to meet. "We might need it later" is not that requirement. Reach for this when

you have a genuine read/write asymmetry across the three sides, or a hard append-only audit mandate, and not one paragraph sooner.

Optimistic Concurrency

The problem: two kitchen staff open the same order, both edit it (one marks an item out of stock, the other bumps the prep time), and the second save silently erases the first. **The shape:** carry a version with the order row, and make the write fail loudly when the version it read no longer matches.

This is Fowler's Optimistic Offline Lock (*PoEAA*). You don't hold a lock across the staff member's think-time, which would tie up the database waiting for a human mid-rush. You bet that conflicts are rare, check that bet at write time, and reject the loser. Postgres gives you a system column for it in `xmin`; a `rowversion`, an explicit integer, or an ETag work the same way.

```
UPDATE "order"
SET    status = @Status, version = version + 1
WHERE  id = @Id AND version = @ExpectedVersion;
-- rows affected = 0 → the other staff member won; reload and retry or surface the conflict
```

What it buys you in production: no lost updates when two staff edit one order, no database locks held across a slow client, and a clean conflict signal you can turn into a "this order changed since you opened it" message. It costs almost nothing to add and saves you from a class of bug that is invisible until a customer's order goes out with the wrong items.

Skip-if: an order is only ever written by one actor at a time, the way a single courier-assignment worker draining its own queue is. No concurrent writers, no lost update, no version column.

Evolutionary Schema Migrations

The problem: the menu and order schema has to change while the platform is live and taking orders, repeatably, across every environment, without a human running ad-hoc SQL against production. **The shape:** versioned, forward-only migration scripts, applied in order, tracked in a table the tool owns.

This is Fowler and Sadalage's evolutionary database design. Each change is a numbered script checked into the repo. A runner (DbUp-style, the same idea as Flyway) applies any script that

hasn't run yet and records it. The schema's state is whatever the ordered scripts produce, and it's identical in every environment because the same scripts ran in the same order.

```
Migrations/
  0001_create_order.sql
  0002_add_tenant_id.sql
  0003_add_menu_modifier_groups.sql
```

Forward-only is the discipline that makes this safe. You don't write down-scripts you'll never test under load; you roll forward with a new migration that corrects the last one. To rename the order's `status` column without downtime you expand, migrate, then contract: add the new column, backfill it, ship code that writes both, then drop the old one in a later migration once nothing reads it.

What it buys you in production: a schema you can reproduce from an empty database to current state by running the folder, a clear audit of every change, and deploys that don't depend on someone remembering to run a script. This is the SQL-first stance held all the way down. The migrations are plain SQL you can read in a code review, not artifacts a framework generates from your model and resolves by magic. EF-style migrations, inferred from a changed C# class and prone to drift the moment two branches touch the model, are exactly the over-engineering tax the component chapter warned about.

Skip-if: there isn't one. If your schema changes after it ships, and it will, you need versioned migrations from the first deploy. The cheapest moment to adopt this is migration 0001.

Sharding and Partitioning

The problem: the `order` table has grown past what a single node serves well. **The shape:** split the data by a key, by range, hash or tenant, so each piece lives on its own and queries hit one piece instead of all of them (Azure Cloud Design Patterns). For a delivery marketplace the natural key is the city: an order is served, tracked and delivered within one metro, so orders shard cleanly by city.

Partitioning splits a table within one database; sharding spreads it across several. Both buy you headroom past a single node's limits. Both also cost you: cross-city queries (a national restaurant brand's daily totals) get expensive or impossible, transactions stop spanning the boundary cleanly, and your shard key becomes a decision you can barely change once orders are distributed by it.

What it buys you in production: scale past the point where one machine, well-indexed and with a read replica or two, runs out of room. That is a real ceiling, and the platforms that hit it are real. Sharding orders by city also keeps a busy metro's dinner rush off the same node as everyone else's.

Premature sharding is one of the classic taxes. Teams shard for a load they project rather than one they have, and inherit the operational weight of a distributed dataset to serve a table that would fit comfortably on one node for years. The order to exhaust first: index properly, then cache the hot reads, then add a read replica, then consider partitioning within one database. Sharding across nodes comes after all of that.

> *You are almost certainly not at the scale that needs this. The platforms that genuinely are know it from their metrics, not from a capacity-planning daydream.*

Skip-if: a single well-tuned instance with proper indexes still has headroom. Measure the ceiling before you build for it. Most platforms never reach it.

Cache-Aside

The problem: the menu is read on every customer visit and changes a few times a day, yet you re-query it from the database thousands of times an hour. **The shape:** check the cache first; on a miss, load the menu from the database, populate the cache, and return (Azure Cloud Design Patterns).

This is the cheapest read-latency win on the altitude, and the menu is the textbook case for it: read-heavy, rarely written, the same `Composite` tree served to everyone browsing a restaurant. The application owns the logic, not a black box: look in the cache, fall through to the `MenuGateway` on a miss, write the result back with a time-to-live.

```
var cached = await _cache.GetAsync(menuKey);
if (cached is not null) return cached;

var menu = await _menuGateway.LoadAsync(restaurantId);   // miss: hit the source
await _cache.SetAsync(menuKey, menu, _ttl);
return menu;
```

What it buys you in production: a large cut in read latency and database load on the hottest read in the app, with very little code and no new persistence model. For a menu hammered at lunch and dinner it's often the biggest win you can ship in an afternoon.

The whole difficulty is invalidation. A cache is a second copy of the truth, and the moment a restaurant edits a price or marks an item sold out, your cached menu is stale until the TTL expires or you evict it. The pattern's quiet cost is reasoning about how wrong a read is allowed to be. A short TTL bounds staleness with minimal logic; explicit eviction when the restaurant saves the menu is tighter but more code and more ways to get it wrong. Pick the staleness window deliberately rather than discovering it through a customer charged the old price.

Skip-if: your reads are already fast enough, or the data changes so often that the cache would miss as much as it hits. Caching live courier GPS positions, which move every few seconds, is invalidation complexity you're paying for nothing.

Soft Delete and Temporal

The problem: "delete" almost never means "destroy." A restaurant removes a menu item but you still need it on last month's orders, or you need to know what an item cost on the day a customer was charged. **The shape:** mark rows deleted instead of removing them, and keep a history of changes rather than overwriting in place.

Soft delete is a flag, a `deleted_at` timestamp, and a default filter that hides marked rows, so a delisted dish disappears from the live menu but still resolves on the orders that bought it. Temporal goes further and keeps the full history of every version of a row, which is what SQL:2011 system-versioned temporal tables and Fowler's temporal patterns formalise. Menu prices are the case that earns it: when a customer disputes a charge, you need the price that was live at the moment they ordered, not today's.

```
-- Soft delete: delist a menu item, don't destroy it.
UPDATE menu_item SET deleted_at = now() WHERE id = @Id;

-- Reads exclude delisted items by default.
SELECT * FROM menu_item WHERE deleted_at IS NULL;
```

What it buys you in production: undo for an accidentally delisted item, a recovery path that doesn't involve a backup restore, and a price history that answers "what did this cost when the order was placed?" without a forensic dig. When the lighter event-sourcing payoff you want is just menu-price history, this gets you most of it without the rest of the machinery.

The cost is that nothing is ever really gone, so every query has to remember the filter (push it into the `MenuGateway` or a view so callers can't forget), tables grow without a purge policy, and "delete my account" requests under privacy law mean a real delete of customer data, not a flag. Soft delete is recoverability, not erasure. Keep the two distinct.

Skip-if: the data is genuinely disposable and nobody will ever ask for it back or audit it. A transient courier-location ping or a throwaway cache row doesn't need a tombstone.

Honorable Mentions

Three patterns missed the cut but are worth a line for the team that wants to go further.

Lightweight Unit of Work (Fowler, *PoEAA*) is a single transaction boundary spanning several gateway calls, so placing an order and decrementing its stock commit or roll back together. Use it as exactly that, a `using` scope around a transaction, and nothing more. The full ORM-style Unit of Work that tracks every changed object is the change-tracking magic the SQL-first stance rejects.

Read Replicas route read traffic to copies of the primary, scaling reads without the weight of sharding. They're the step you reach for before partitioning, and they pair naturally with the menu cache-aside. The catch is replication lag, so route reads that must be current (a customer's live order status) to the primary.

Connection Pooling reuses database connections instead of opening one per request, which is the difference between a service that holds steady under load and one that exhausts the database's connection limit. Your data library and driver give it to you; the honorable-mention work is tuning the pool size, not building it.

Persistence holds state still. Production also has to move it between services, reliably and without dropping it on the floor. That is the next altitude.

Messaging & Scale

Persistence holds state still. Production has to move it between services, absorb spikes without falling over, and never lose a message it promised to deliver. A handful of patterns do most of that work.

The last altitude kept your state safe. This one keeps it moving. The moment the order service hands work to the kitchen, the courier-matcher, or the payment provider over a network, you inherit a new class of problem. The producer is fast and bursty. The consumer is slow and steady. And the connection between them can drop at the worst possible second. The patterns here put a buffer in that gap, scale the slow side out, and turn "I'll process this `OrderPlaced` later" into a promise you can actually keep. We carry the food-delivery marketplace through them all. A customer places an `Order`, a restaurant (the tenant) cooks it, a courier delivers it, and at lunch every one of those steps is on fire at once.

A broker sits at the centre of all of it. The book stays neutral on which one. Examples name **GCP Pub/Sub** first, with **AWS (SQS + SNS)** and **Azure Service Bus** as the counterparts; the patterns outlive any of them.

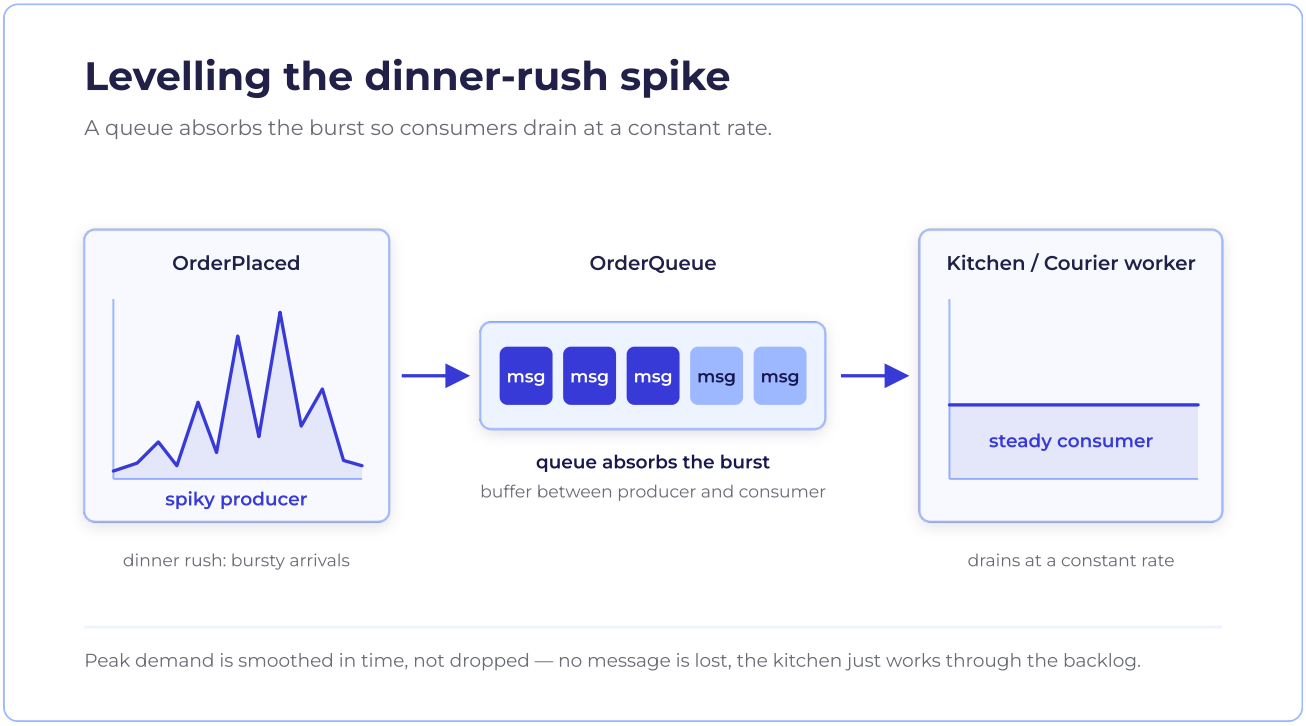
Queue-Based Load Levelling

The lunch rush hits and a fixed-capacity consumer falls over at the peak (Azure Cloud Design Patterns). Noon and 7pm are not the average; they are five minutes of orders arriving at ten times the baseline rate. The fix is to put a queue between the order intake and the work behind it, so customers place orders at their own pace and the kitchen pipeline drains at its own.

```
// Order intake: accept fast, return immediately.
public async Task<IActionResult> PlaceOrder(Order order)
{
    await _publisher.PublishAsync("orders.placed", order);  // enqueue, don't cook
    return Accepted();                                      // 202 — order is on the queue
}
```

What this buys you in production: the order pipeline is sized for the *average* lunch, not the noon spike. A rush that would have taken the synchronous version down instead becomes a queue that grows for a few minutes and drains. Your database connection pool sees a steady trickle instead of a thousand orders landing in the same ten seconds. The endpoint returns in milliseconds because it does almost nothing but accept the order.

The skip-if: if the caller needs the answer *now* (a card authorisation, a "is this restaurant still open" check), a queue is the wrong tool. Asynchronous processing means the customer's app has to come back later to ask how it went. Don't level load that has to be synchronous.



Competing Consumers

One courier-assignment worker can't keep up with the queue, and the backlog of unassigned orders grows without bound (Azure Cloud Design Patterns; Hohpe & Woolf, *Enterprise Integration Patterns*). Point several identical assignment workers at the same queue and let the broker hand each order to whichever worker is free.

```
// N identical courier-assignment workers, each pulling from the same subscription.
await foreach (var msg in subscription.ReadAllAsync(ct))
{
    await _matcher.Assign(msg.Order);   // CourierMatchingStrategy picks a courier
    await msg.AckAsync();               // ack only after success — see DLQ below
}
```

What this buys you in production: horizontal scale with no code change. At lunch you run twenty assignment workers; at 3pm you run two. Throughput is a dial you turn by adding

workers, and the broker handles the distribution. When a worker dies mid-assignment, the broker redelivers to a sibling once the lease expires, so a crash costs you a retry, not an order that never finds a courier.

The skip-if: two things have to hold. The work must be parallelisable (no ordering dependency between orders), and each handler must be **idempotent**, because redelivery means an order can be assigned more than once. If assigning the same order twice books two couriers, fix that before you add the second worker. We come back to idempotency in the resilience chapter; here it's the precondition that makes competing consumers safe.

> *Adding workers is a dial. Making the assignment safe to run twice is the engineering.*

Publish/Subscribe

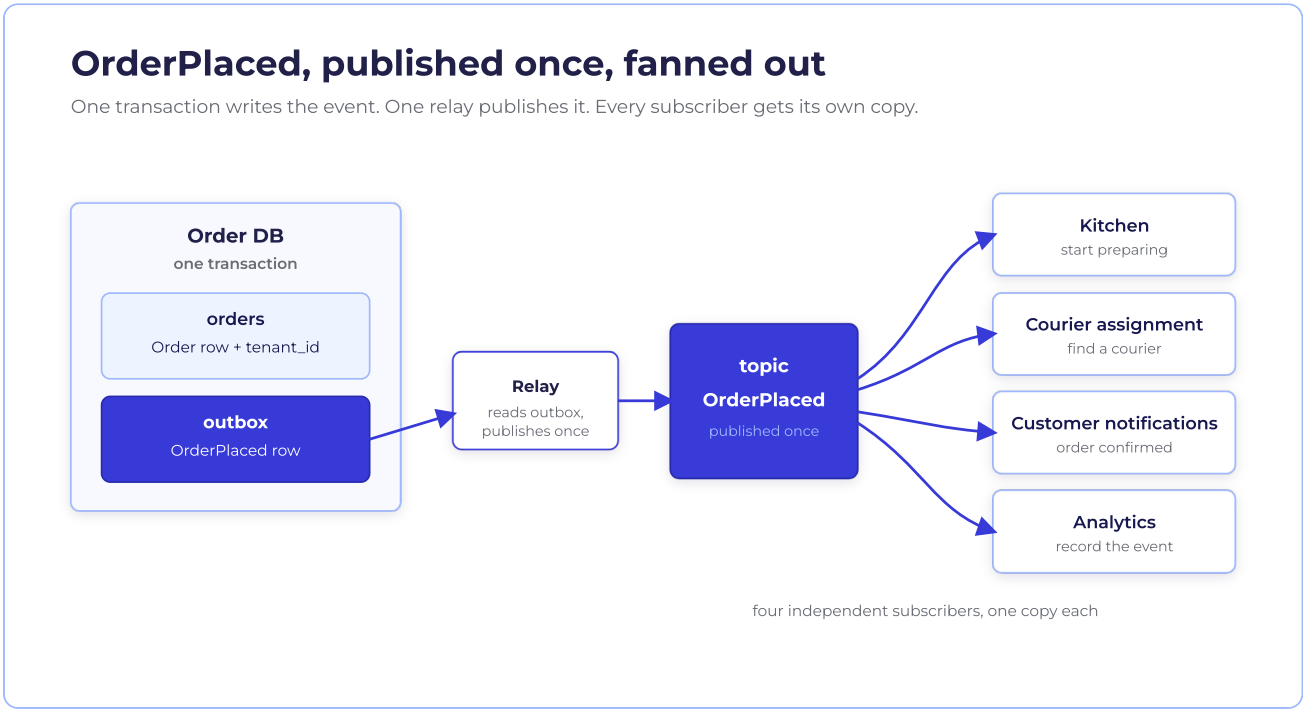
A queue delivers each message to exactly one consumer, but one placed order is news to four different parts of the business at once (Azure Cloud Design Patterns; EIP Publish-Subscribe Channel). The kitchen has to start cooking, courier-matching has to find a rider, the customer has to see "order confirmed," and analytics has to count it. Publish to a topic instead of a queue, and every subscriber gets its own copy.

The order service stops knowing who its consumers are. It announces `OrderPlaced`; the kitchen, courier-matching, the customer-notification service, and analytics each subscribe on their own. Adding a fraud-screening consumer later is a new subscription, not a change to the order service. That decoupling is the whole point.

```
// One publish, fanned out to every subscriber of the topic.
await _bus.PublishAsync(new OrderPlaced(orderId, tenantId, total));
// kitchen, courier-matching, customer notifications, analytics — each its own subscription
```

What this buys you in production: the sides of the marketplace evolve independently. The team that owns analytics adds a subscription without filing a ticket against the order service. On most brokers a topic with one subscription is just a queue, so you can start point-to-point (order service → kitchen only) and fan out to couriers and analytics later without rebuilding.

The skip-if: pub/sub is for *facts that already happened*, broadcast to whoever cares. `OrderPlaced` is a fact. If you instead need a specific worker to assign a specific courier and tell you the result, that's a command on a queue, not an event on a topic. Broadcasting a command to nobody-in-particular is how an order quietly never gets a courier.



Outbox

Here is the hole the persistence chapter left open. You commit the `Order` row to the database, then publish `OrderPlaced` to the broker. Two systems, two network calls, no shared transaction. If the process dies between them, you've either saved the order without telling the kitchen, or announced an order that was never saved. Dual writes have no safe ordering.

The Transactional Outbox closes it (Chris Richardson, microservices.io). Write the `OrderPlaced` message into an `outbox` table *in the same database transaction* as the order itself. A separate relay reads unpublished rows and pushes them to the broker, marking each as sent. The DB commit is now the single source of truth: if the order row is there, `OrderPlaced` will go out; if the transaction rolled back, there's nothing to send.

```
-- place_order: both writes commit together, atomic by construction.
CREATE OR REPLACE PROCEDURE place_order(
    p_id uuid, p_tenant uuid, p_total bigint,
    p_msg_id uuid, p_payload text)
LANGUAGE sql AS $$
    INSERT INTO orders (id, tenant_id, total)
    VALUES (p_id, p_tenant, p_total);

    INSERT INTO outbox (id, topic, payload, created_utc)
    VALUES (p_msg_id, 'order.placed', p_payload::jsonb, now());
$$;
```

```
// The gateway calls the proc through Dapper: one transaction, no ORM.
await _db.ExecuteAsync("place_order",
    new { id, tenant, total, msgId, payload },
    commandType: CommandType.StoredProcedure);

// Relay: drain unsent OrderPlaced rows, publish, mark sent. Runs on its own loop.
foreach (var row in await _outbox.ReadUnsentAsync(batch: 100))
{
    await _bus.PublishAsync(row.Topic, row.Payload);
    await _outbox.MarkSentAsync(row.Id);
}
```

What this buys you in production: at-least-once delivery of `OrderPlaced` you can actually trust, without a distributed transaction across the DB and the broker. The relay can crash and resume; it just re-sends anything it didn't get to mark. That re-send is exactly why the kitchen and courier consumers had to be idempotent. The outbox is the same stored-procedure write the persistence chapter (Ch 5) built, with one extra `INSERT` riding inside the transaction.

The skip-if: if you aren't publishing as a result of a database write, you don't have the dual-write problem, so you don't need an outbox. A "courier moved" GPS ping isn't tied to a transactional state change; publish it directly. The outbox earns its keep only when a state change and a message have to happen together or not at all, as `OrderPlaced` does.

Backpressure / Dead-Letter Queue

An order whose payment keeps failing, or one the matcher can't assign because no courier is online, will be retried forever, blocking the queue behind it and burning CPU on a failure that will never succeed this minute. Two patterns keep a struggling system honest instead of letting it thrash or silently drop an order.

Backpressure is the upstream signal: when the assignment queue is deep or the workers are saturated, slow intake down rather than buffer orders without limit (reactive-streams). A bounded queue that rejects or pauses is telling you the truth about capacity, which at the height of the dinner rush is exactly when you need the truth. An unbounded one just moves the moment you fall over to later, and makes it worse.

A Dead-Letter Queue is the downstream escape hatch (EIP Dead Letter Channel). After a `ProcessPayment` or `AssignCourier` message fails N times, the broker moves it to a side queue instead of redelivering it forever. The main queue keeps flowing for the orders that *can* be processed; the stuck order waits somewhere you can inspect it, alert on it, and replay it once the card is re-authorised or a courier comes online.

```
// Most brokers do this for you: set max-delivery-attempts + a DLQ target.
// In handler code, the contract is simple — assign the order, or let it throw.
try { await _matcher.Assign(msg.Order); await msg.AckAsync(); }
catch (Exception ex)
{
    _log.Error(ex, "Assignment failed for {OrderId}", msg.Order.Id);
    await msg.NackAsync();   // broker counts the attempt, DLQs at the limit
}
```

What this buys you in production: failure becomes visible and bounded. One un-assignable order doesn't wedge the whole rush, and you never silently lose an order; it lands in the DLQ with its history, waiting for a human or a replay. An alert on DLQ depth is one of the highest-signal alarms you can wire up, and "orders we couldn't deliver" is a number the business will want anyway.

The skip-if: none, really. If you run consumers, configure a dead-letter target. The skip is in the *backpressure* tuning, not the DLQ: don't hand-build elaborate flow-control before you've measured an actual saturation problem. Start with a bounded queue and a DLQ, then tune.

> *Never silently drop an order. One you can't process belongs in a queue you can see, not a log line nobody reads.*

Saga / Process Manager

This is the heaviest pattern in the chapter, and the one most teams reach for too early. Fulfilling an order spans several services, each with its own database, and you need them to agree on the outcome (Garcia-Molina & Salem, *Sagas*, 1987; Chris Richardson, microservices.io). Charge the customer, assign a courier, confirm with the restaurant: three services, three databases, no distributed transaction to roll them all back. So a Saga sequences the steps and, when one fails, runs a **compensating action** to undo the ones that already succeeded.

A Process Manager is the orchestrated form: a single component that holds the state of the in-flight order, reacts to each step's outcome, and decides the next move or the rollback. It walks the order through payment, then courier assignment, then restaurant confirmation. If the restaurant won't accept the order at the end of that, the manager refunds the customer and releases the courier it already booked. Every forward step needs a defined way back.

```
// Order-fulfilment process manager: react to each event, advance or compensate.
public async Task On(PaymentTaken e)               // charged → now assign
{
    var assigned = await _matcher.TryAssign(e.OrderId);   // CourierMatchingStrategy
    if (assigned) await _bus.PublishAsync(new CourierAssigned(e.OrderId));
    else          await _bus.PublishAsync(new RefundRequested(e.OrderId)); // compensate
}
```

What this buys you in production: eventual consistency across payment, courier-matching, and the restaurant without a two-phase commit you can't get anyway. The order reaches a defined end state (delivered, or fully refunded and released) even when a step fails halfway, and the saga's state tells you exactly where any stuck order is sitting.

The skip-if, and it's a big one: **you need an actual multi-step distributed transaction before any of this earns its cost.** If charge, assign, and confirm all lived in one database, a single local transaction would do the same job with none of the moving parts. Sagas bring orchestration code, a compensating action for every step (refund the charge, release the courier, cancel the

restaurant ticket), and a whole new failure mode: the compensation itself failing. Most teams that build a saga were one well-placed `BEGIN TRANSACTION` away from not needing it. Reach for it when an order genuinely spans services; until then, the local transaction is simpler and you should keep it.

Claim-Check

Some order messages carry a large payload (a generated PDF receipt, a "proof of delivery" photo from the courier, a full itemised invoice), and big messages clog the broker, blow past message-size limits, and make every consumer pay to move bytes most of them ignore (EIP; Azure Cloud Design Patterns). Store the payload in object storage and put only a reference on the bus.

```
// Producer: stash the receipt PDF, send the token.
var uri = await _storage.UploadAsync(bucket: "receipts", receiptPdf);
await _bus.PublishAsync(new ReceiptReady(orderId, uri));   // small message

// Consumer: fetch only when it actually needs the bytes.
var bytes = await _storage.DownloadAsync(msg.Uri);
```

What this buys you in production: the bus stays fast and cheap, because it moves order IDs and a storage URI, not megabytes of PDF. You sidestep broker size limits without splitting messages by hand, and the analytics consumer that only needs the order total never downloads the receipt. Object storage (GCP Cloud Storage; AWS S3, Azure Blob Storage) is already the right home for large, immutable payloads like a receipt or a delivery photo.

The skip-if: if your messages are small (an order ID, a status, a courier location), the indirection is pure overhead. A claim-check adds a storage write, a storage read, and a lifecycle question (when does the old receipt get cleaned up?). Reach for it when payload size is an actual problem, not by default. `OrderPlaced` is a handful of fields; it does not need a claim-check.

Honorable Mentions

Three patterns that just missed the cut, for when you need them.

Idempotent Consumer is arguably the most important pattern in this chapter, and it lives in the next one. At-least-once delivery means every consumer will eventually see a duplicate `OrderPlaced`, and a kitchen handler that cooks the same order twice corrupts state. The fix (dedup on the `order_id`, or design the operation to be naturally repeatable) is the **Idempotency** pattern in Resilience, where it belongs alongside the retries that make duplicates inevitable.

Event-Carried State Transfer (Fowler) puts enough state *inside* the event that subscribers don't have to call back to the source to act on it. Fattening `OrderPlaced` with the line items and delivery address means the kitchen consumer never calls the order service back to render the ticket. It cuts coupling and read load, at the cost of fatter messages and data that can be stale by the time it's read. Useful when a chatty callback pattern is hurting you.

Priority Queue (Azure Cloud Design Patterns) lets urgent work jump ahead of routine work when one queue serves both. A "courier waiting at the counter" re-assignment should beat a fresh order in the assignment queue. Most teams approximate it with two queues and more workers on the fast one, which is simpler and usually enough.

Moving an order across a network means the payment provider, the courier service, and the kitchen now fail independently of each other. Surviving that, without taking the rest of the marketplace down with the broken part, is the next altitude.

Resilience: Staying Up When Dependencies Don't

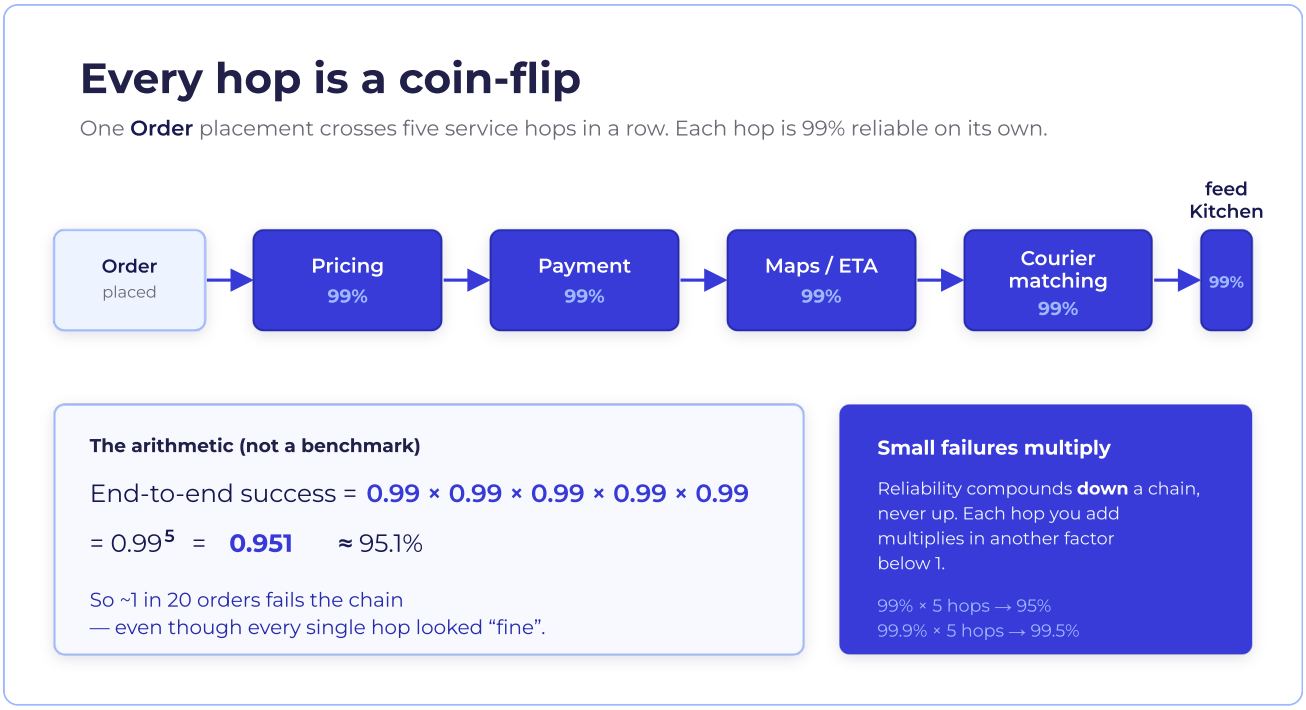
Every network call is a coin-flip that sometimes lands wrong. These patterns turn "the dependency hiccuped" into a non-event nobody notices.

The moment you split work across a network, you inherit a hard truth: the parts fail independently, and they fail at the worst time. The payment gateway returns a 503 for forty seconds in the middle of the Friday dinner rush. The maps service that quotes delivery ETAs slows to a crawl and drags your threads down with it. A database connection drops mid-query while a customer is placing an order. None of these are exotic. They are Tuesday.

Here is the arithmetic that should change how you think about a call chain. Placing one order touches several services in turn: pricing, payment, the maps/ETA quote, courier matching, the restaurant's kitchen feed. Suppose each hop succeeds 95% of the time, which is generous for a real network. Chain ten of them and your end-to-end success rate is 0.95^10, about 60%. Stretch it to twenty hops and you are at 0.95^20, roughly 36%. This is not a measured failure rate from some study; it is multiplication. But it tells you the thing worth knowing: in a distributed system, individual reliability compounds *downward*, and the only way back up is to make each hop forgive the others.

> *Reliability isn't the absence of failure. It's the presence of recovery.*

The resilience altitude is where you stop pretending dependencies are always there and start designing for the moments they aren't. The patterns here are drawn mostly from Nygard's *Release It!*, with the retry maths from AWS. We use **Polly** for the C#, but the concepts outlive any library. Throughout, the call we keep coming back to is the one that hurts most when it breaks: placing an order and charging the customer for it.



Retry (Exponential Backoff + Jitter)

The problem: a transient fault, a dropped connection, a brief timeout, a momentary 503 from the payment gateway, fails a charge that would have succeeded a half-second later.

The naive fix is to try again immediately. That makes things worse. When the payment gateway wobbles, every order in flight retries at once, and the synchronised stampede keeps it down. The fix is *exponential backoff*: wait longer after each failure (200ms, 400ms, 800ms). The second fix is *jitter*: randomise the delay so a thousand orders don't all retry on the same tick. AWS's own guidance on this is the canonical reference, and it lands on full jitter as the default (Marc Brooker, *"Exponential Backoff And Jitter,"* AWS Architecture Blog).

```
var retry = new ResiliencePipelineBuilder()
    .AddRetry(new RetryStrategyOptions
    {
        MaxRetryAttempts = 3,
        BackoffType = DelayBackoffType.Exponential,
        UseJitter = true,
        Delay = TimeSpan.FromMilliseconds(200),
        ShouldHandle = new PredicateBuilder().Handle<PaymentGatewayException>()
    })
    .Build();
```

What it buys you in production: the most return for the least code of any pattern here. Most faults are transient, and a retry with backoff turns the majority of them into a non-event the customer never sees.

Skip-if: the operation isn't safe to repeat. A retry on a non-idempotent charge can bill the card twice or place two orders for the same basket. Don't add retry until you've read the idempotency section below; the two patterns are a pair, not a choice.

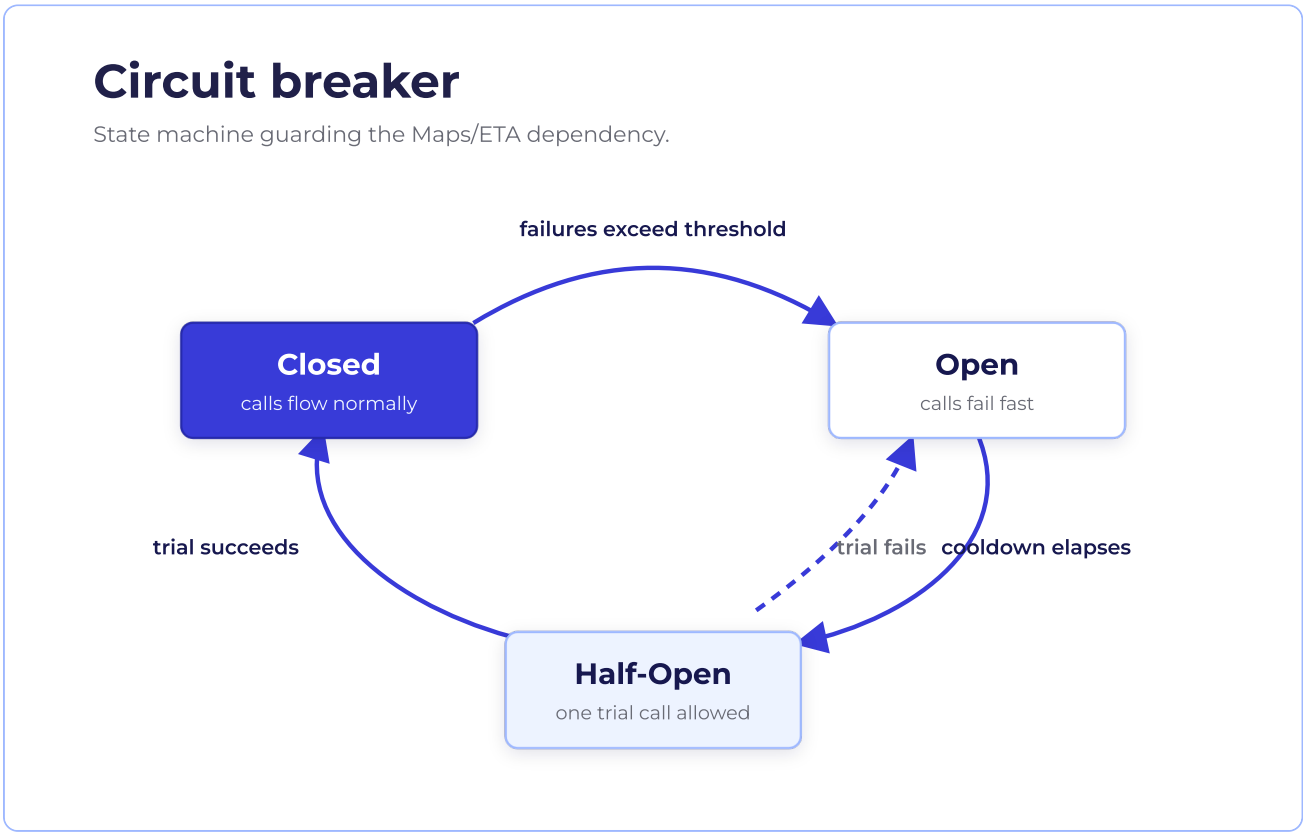
> ***In the front-end.*** *The customer app retries its own fetches too. A flaky mobile connection drops the "place order" request; the client retries with backoff over the same idempotency key (so the server never double-places), and queues the order locally if the device is offline, replaying it when the signal returns.*

Circuit Breaker

The problem: the maps/ETA service is genuinely down, not hiccuping, and every order you place waits for its quote to time out, holds a thread, and pointlessly hammers a service that can't answer.

Retrying a corpse is cruel and slow. The Circuit Breaker (Nygard, *Release It!*; Martin Fowler's bliki, *"CircuitBreaker"*) wraps the call in a state machine. While calls succeed it stays **closed**. After a run of failures it trips **open** and fails fast for a cooldown, no call, no wait. After the cooldown it goes **half-open**, lets a trial call through, and either recovers to closed or trips open

again. While the breaker on the ETA service is open, order placement keeps running; it just shows a default ETA instead of a live one (the Timeout + Fallback section makes that concrete).




```
var breaker = new ResiliencePipelineBuilder()
    .AddCircuitBreaker(new CircuitBreakerStrategyOptions
    {
        FailureRatio = 0.5,
        MinimumThroughput = 10,
        SamplingDuration = TimeSpan.FromSeconds(30),
        BreakDuration = TimeSpan.FromSeconds(15)
    })
    .Build();
```

What it buys you in production: it stops a struggling dependency from taking you down with it, and it gives that dependency room to recover instead of a fresh wave of traffic. A dead ETA service no longer blocks every order; it just degrades the ETA. Fail fast beats fail slow every time.

Skip-if: you're calling a single in-process component or a local cache with no real failure mode. A breaker around your own menu Composite, resolved in memory, is moving parts you have to reason about for no payoff.

Rate Limiting / Throttling

The problem: demand on the public order API exceeds what you can serve, and unbounded traffic, a rush, a scraper, a buggy partner integration hammering the menu endpoint, turns a busy system into a fallen one.

Rate limiting caps how much work you accept; throttling caps how much you send. Both protect a finite resource by refusing or deferring excess rather than collapsing under it (Azure Cloud Design Patterns, Throttling). .NET ships this in the box with `System.Threading.RateLimiting`, so you rarely roll your own. On a multi-tenant marketplace you usually key the limiter by `tenant_id`, so one busy restaurant can't crowd the API for the rest.

```
var limiter = new SlidingWindowRateLimiter(new SlidingWindowRateLimiterOptions
{
    PermitLimit = 100,
    Window = TimeSpan.FromSeconds(1),
    SegmentsPerWindow = 4,
    QueueLimit = 20
});

using var lease = await limiter.AcquireAsync(permitCount: 1);
if (!lease.IsAcquired) return Results.StatusCode(429);
```

What it buys you in production: a predictable ceiling. You'd rather reject the excess with a clean 429 than let every order request degrade into a timeout. It also keeps you a good citizen against the payment gateway or maps API, both of which enforce quotas of their own.

Skip-if: your traffic is comfortably below any limit and there's no shared resource to protect. A limiter on an internal courier-status endpoint that sees ten requests a minute is ceremony.

Bulkhead

The problem: one slow dependency consumes every thread, connection, or task slot in the pool, and a failure in one corner of the system starves all the others.

The name comes from a ship's hull (Nygard, *Release It!*): seal it into compartments so one flooded section doesn't sink the vessel. In software you isolate resources per dependency. The surge-pricing service is the obvious candidate: it calls out to demand and weather models, it's slow under load, and it's exactly when load is high that you can least afford it to drag order placement down with it. Give surge pricing its own bounded pool, and when it backs up it exhausts *its* compartment, not the threads placing orders.

```
var surgePricing = new ResiliencePipelineBuilder()
    .AddConcurrencyLimiter(permitLimit: 10, queueLimit: 5)
    .Build();
// Surge calls beyond 10 concurrent wait in a queue of 5, then reject.
```

What it buys you in production: fault isolation. Order placement keeps serving even while the surge-pricing model is timing out, because the two never share a pool. Worst case, an order goes through at the base price instead of failing.

Skip-if: you have one dependency and one workload. Compartments only help when there's something to wall off; a single-purpose courier-location worker doesn't need them.

Timeout + Fallback

The problem: the ETA call hangs. Not fails, *hangs*, holding a thread indefinitely while the customer stares at a spinner and the next order piles up behind it.

A hang is worse than an error, because an error you can handle and a hang just consumes you. Every outbound call needs a timeout (Nygard, *Release It!*). Pair it with a *fallback*: when the ETA call times out or fails, return something useful instead of an exception, a default "30–45 min" estimate, a cached quote, a degraded-but-honest answer.

```
var resilientEta = new ResiliencePipelineBuilder<DeliveryEta>()
    .AddTimeout(TimeSpan.FromSeconds(2))
    .AddFallback(new FallbackStrategyOptions<DeliveryEta>
    {
        FallbackAction = _ => Outcome.FromResultAsValueTask(DeliveryEta.Default),
        ShouldHandle = new PredicateBuilder<DeliveryEta>()
            .Handle<TimeoutRejectedException>()
    })
    .Build();
```

What it buys you in production: bounded latency and a graceful answer. The customer gets a default ETA in two seconds instead of a blank screen in thirty, and the order still goes through. A bounded wait is the difference between a slow checkout and a dead one.

Skip-if: you genuinely have no acceptable fallback and a wrong answer would be worse than no answer. The charge itself is like that, you can't fall back to a "default" payment. Keep the timeout regardless; it's the fallback that's optional, not the bound on waiting.

Idempotency (What Makes Retries Safe)

The problem: you retried the charge, but you don't know whether the first attempt actually landed before the connection dropped, so retrying might bill the customer twice and place two `Order`s for one basket.

This is the pattern that licenses the first one. An operation is *idempotent* if doing it twice has the same effect as doing it once. Reads are idempotent for free. Writes are not, until you make them so, usually with an **idempotency key**: the customer app sends a unique token with the basket, you record it on first execution, and a retry carrying the same token returns the original `Order` instead of placing a second one (Stripe's API documents this pattern well, which is no accident; it's a payments problem first). The same key flows through to the payment gateway, so the charge is deduplicated end to end.

```
// On the order-placement path, keyed by the basket's token.
var existing = await _orderGateway.FindByIdempotencyKeyAsync(key);
if (existing is not null) return existing;        // replay, don't re-place

var order = await _orderGateway.PlaceAndRecordAsync(key, request);
return order;
```

What it buys you in production: safe retries. Without idempotency, retry is a loaded gun pointed at the customer's card; with it, the worst a duplicate request can do is return the same `Order` twice. Every at-least-once message broker (Ch 6) leans on this too, which matters once `OrderPlaced` fans out to the kitchen and courier-matching consumers.

Skip-if: the operation is already a pure read, or naturally idempotent (setting an order's status to a fixed value). Don't bolt a key store onto something that can't double-fire.

> *Retry is the promise. Idempotency is the thing that lets you keep it.*

The Composed Pipeline

The patterns above are rarely used alone. In production you *layer* them, and the order matters. A timeout bounds each individual attempt. A retry wraps that, trying a few times with backoff. A circuit breaker sits across all of it, so a dependency that's truly down trips the breaker instead of grinding through retries forever.

Polly composes these as a single pipeline. This is the listing to copy: retry, breaker, and timeout assembled the way you'd actually run them on the payment gateway call, the most consequential outbound dependency the order service has.

```
using Polly;
using Polly.CircuitBreaker;
using Polly.Retry;
using Polly.Timeout;

// The pipeline wrapping every charge against the payment gateway.
public static ResiliencePipeline<PaymentResult> BuildPaymentPipeline()
{
    return new ResiliencePipelineBuilder<PaymentResult>()

        // Outermost: trip after sustained failure, fail fast during cooldown.
        .AddCircuitBreaker(new CircuitBreakerStrategyOptions<PaymentResult>
        {
            FailureRatio = 0.5,
            MinimumThroughput = 10,
            SamplingDuration = TimeSpan.FromSeconds(30),
            BreakDuration = TimeSpan.FromSeconds(15),
            ShouldHandle = new PredicateBuilder<PaymentResult>()
                .Handle<PaymentGatewayException>()
                .HandleResult(r => r.IsTransientGatewayError)
        })

        // Middle: retry transient faults with exponential backoff + jitter.
        // Safe only because every charge carries the basket's idempotency key.
        .AddRetry(new RetryStrategyOptions<PaymentResult>
        {
            MaxRetryAttempts = 3,
            BackoffType = DelayBackoffType.Exponential,
            UseJitter = true,
            Delay = TimeSpan.FromMilliseconds(200),
            ShouldHandle = new PredicateBuilder<PaymentResult>()
                .Handle<PaymentGatewayException>()
                .Handle<TimeoutRejectedException>()
                .HandleResult(r => r.IsTransientGatewayError)
        })

        // Innermost: bound each individual attempt.
        .AddTimeout(TimeSpan.FromSeconds(2))
```

```
        .Build();
}
```

Read it from the outside in. The breaker is the gatekeeper; when it's open, nothing below it even runs. Inside, each retry attempt gets its own two-second timeout, and the backoff spaces the attempts out. Register the pipeline once and apply it to every charge against the payment gateway. Other dependencies get their own pipelines: the maps/ETA service from earlier wants a shorter timeout and a fallback rather than three retries, because a slow ETA should degrade instead of blocking. A flaky payment gateway and a slow maps service have different tolerances.

The one rule that ties the room together: the inner timeout must be shorter than the outer ones, or the layers fight each other. A retry budget of three attempts at two seconds each means a worst case near six seconds, so size the breaker and the customer-facing checkout deadline with that in mind.

Steady State

The problem: nothing here failed, but something *grew*. The couriers stream a GPS ping every few seconds; the live-location cache and the ping table grow without bound until they fill the disk or exhaust the pool at 3am, when no one was watching.

This is the pattern that catches the failures the other six don't. Nygard's **Steady State** rule (*Release It!*): for every mechanism that accumulates a resource, there must be a counter-mechanism that reclaims it. Purge GPS pings older than the day's deliveries on a schedule. Cap the courier-location cache with an eviction policy so it holds the latest fix per courier and nothing older. Rotate and expire logs; bound the connection pool with a hard ceiling. A system in steady state can run for months untouched; one without a reclaim path is a time bomb with a slow fuse.

```
services.AddStackExchangeRedisCache(o =>
    o.Configuration = connectionString);
// Each courier's latest fix expires on its own; stale pings never accumulate.
await _cache.SetAsync($"courier:{courierId}:location", fix,
    new DistributedCacheEntryOptions
    {
        AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5)
    });
```

What it buys you in production: the quiet kind of uptime, the system that doesn't page you at 3am because the ping table filled the disk. Most of the worst outages aren't dramatic crashes; they're a resource that grew unbounded until it couldn't.

Skip-if: nothing about this one. Every growing resource needs a bound. The skip is only on *which* mechanism, you might not cap the courier-location cache in Redis this month, but you always need the discipline.

> *The outage that fells you isn't the one that crashes. It's the one that fills up while you sleep.*

Honorable Mentions

Three more that earn a look once the core is in place.

Graceful Degradation turns off non-essential features under load so the core keeps working, hide the "recommended for you" carousel, keep order placement. It's Fallback's strategy applied at the feature level.

Load Shedding drops low-priority requests on purpose when you're saturated, the deliberate twin of Rate Limiting: better to refuse a restaurant's analytics refresh than to fail a customer placing an order.

Failover / Redundancy keeps a standby ready so a dead instance or zone hands off to a live one. It's the infrastructure answer to resilience, and it pairs with the hosting altitude in Ch 9, where redundancy is a deployment property, not application code.

What This Altitude Gives You

Resilience is the altitude where you accept that dependencies fail and design so it doesn't matter. Retry forgives the payment gateway's transient 503; idempotency makes that forgiveness safe, so a retry never double-charges the customer. The breaker fails fast when the maps service is genuinely down. Timeouts bound the wait and a default ETA fills the gap, the bulkhead keeps surge pricing from starving order placement, rate limits cap the public order API, and steady state purges the GPS pings before they fill the disk. Layered through one Polly pipeline on the payment call, they cost you a few dozen lines and buy you the thing that separates a prototype from a product: orders keep going through.

There's a catch, and it's the next chapter's whole job. A resilient system fails *quietly*. The retry that saved you, the breaker that tripped, the fallback that served stale data, all of it happens silently by design. That's dangerous unless you can see it.

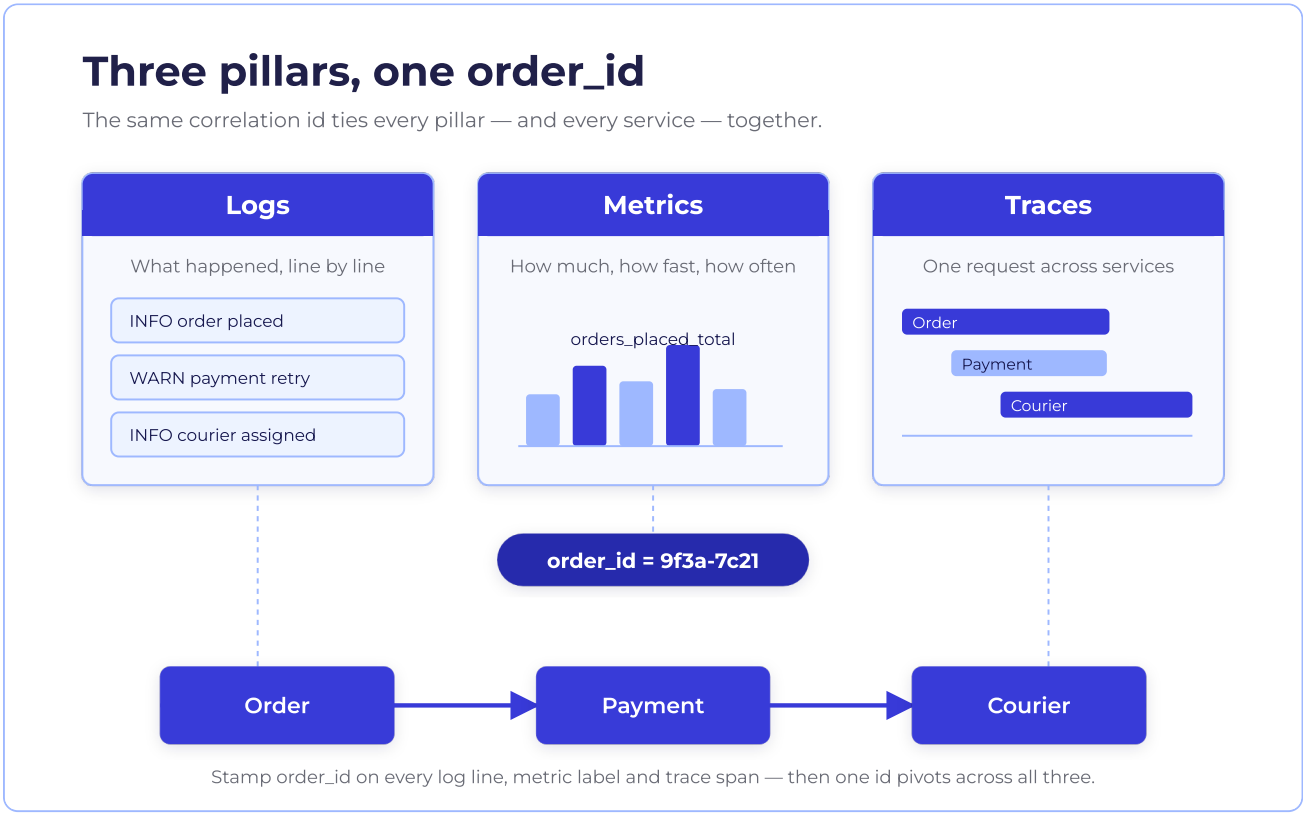
A system that recovers without telling you is a system that's hiding how close it came; next we make it observable.

Observability & Diagnostics: Seeing Inside Production

Resilient systems fail quietly, which is exactly what makes them dangerous. This altitude is how you see inside a running service before a customer does, and how you wake the right person when it matters.

The order service can swallow a failure so gracefully that nobody notices for hours. The payment retry succeeded on the third attempt. The breaker tripped and the fallback held while courier assignment drained a little slower than usual. No single event registers as an outage, yet together they are a service degrading toward the dinner rush it won't survive. None of that reaches you unless the service tells you, and it only tells you what you thought to make it say.

The vocabulary here is the three pillars: logs, metrics, and traces. Logs are the discrete events ("order 4471 failed payment for tenant 12"). Metrics are the aggregates ("p99 confirmation latency, last five minutes"). Traces are a single order's path across services. A handful of patterns turn those three pillars into something a small team can actually run, plus the one thing that makes all of it worth the effort: an alert that pages a human only when an objective is at risk.



Health Endpoint Monitoring

The problem: an orchestrator needs to know whether your order service container is alive and whether it's ready for traffic, and it can't tell from the outside (Azure Cloud Design Patterns). A process that's booted but still warming its connection pool will accept orders it can't write. A process wedged on a deadlock looks identical to a healthy one until you ask it.

You expose two endpoints. Liveness answers "am I running at all" and, if it fails, the orchestrator restarts you. Readiness answers "should I receive traffic right now" and, if it fails, the load balancer routes around you without a restart. Keep them separate. A slow downstream dependency should fail readiness, not liveness, or you'll get a restart loop that solves nothing.

```
builder.Services.AddHealthChecks()
    .AddCheck("self", () => HealthCheckResult.Healthy(), tags: ["live"])
    .AddNpgSql(ordersConnString, tags: ["ready"]);

app.MapHealthChecks("/health/live",  new() { Predicate = c => c.Tags.Contains("live") });
app.MapHealthChecks("/health/ready", new() { Predicate = c => c.Tags.Contains("ready") });
```

What it buys you in production: the platform heals you without a human. A wedged instance gets recycled. A warming one stops taking orders until it's ready, and a deploy that boots broken never receives a request at all. This is the contract between the order service and Cloud Run (AWS App Runner/ECS, Azure Container Apps), and it's nearly free to hold up your end.

Skip-if: you're not behind anything that reads the probes. A box you SSH into and restart by hand gets nothing from a readiness endpoint. The moment an orchestrator is in the picture, you need both.

Structured Logging

The problem: a log line written as a sentence is readable by one human and queryable by none. `"Order 4471 failed payment for restaurant 12 after 2 retries"` cannot be filtered, counted, or grouped without a regular expression you'll regret. At dinner rush, grep is not a search engine.

Log events, not strings. Each line carries a message template and its values as named properties, so the aggregator can index them. With Serilog the shape barely changes from what you'd write anyway, but the output is structured JSON your backend can query.

```
Log.Information("Order {order_id} for tenant {tenant_id} failed payment after {retries} retri
    order.Id, order.TenantId, retryCount);
```

That renders as a readable line locally and as `{ order_id: 4471, tenant_id: 12, retries: 2 }` to the sink, where you can ask "show every failed payment for tenant 12 during tonight's rush" and get an answer in seconds.

What it buys you in production: your logs become a dataset. You can filter by restaurant, count failures by type, and follow one order by its `order_id` instead of reading a wall of text. Push the common fields (`tenant_id`, `order_id`, environment) into the log context once and every line inside that scope carries them automatically.

> *A log you can query is a tool. A log you can only read is a diary.*

Skip-if: nothing. Structured logging costs you a logger configuration and the discipline to use message templates instead of string interpolation. There's no version of a production service where unstructured logs are the right call. Just don't put a customer's card number or home address in a property; the audit-logging discipline below applies to ordinary logs too.

Metrics: The Four Golden Signals

The problem: you can emit a thousand metrics and still not know whether the order service is healthy. A wall of dials buys you nothing during a rush except more dashboards no one has time to read. You need a small set that actually predicts user pain.

Google's SRE book names four, and they're enough to start (Google, *Site Reliability Engineering*, 2016). **Latency** is how long requests take, split by success and failure so a fast stream of failed orders doesn't hide in the average. **Traffic** is demand: orders per second, the assignment queue depth as the dinner peak hits. **Errors** is the rate of requests that fail, the orders that never confirm. **Saturation** is how full your most constrained resource is: CPU, memory, the database connection pool, the thing that gives out first when every city orders at once. Watch those four on the order service and you'll catch most problems before a customer reports them.

```
var meter = new Meter("Orders.Api");
var latency = meter.CreateHistogram<double>("order.confirm.duration", unit: "ms");
var errors  = meter.CreateCounter<long>("order.confirm.errors");

latency.Record(elapsedMs, new("route", route), new("status", statusCode));
if (statusCode >= 500) errors.Add(1, new("route", route));
```

What it buys you in production: a four-line health story you can read at a glance and alert on with confidence. Latency and errors tell you the customer's experience; traffic and saturation tell you whether you're about to run out of headroom when the rush lands. Instrument with the .NET metrics API and OpenTelemetry exports the same counters to Cloud Monitoring (CloudWatch, Azure Monitor) without rewriting them.

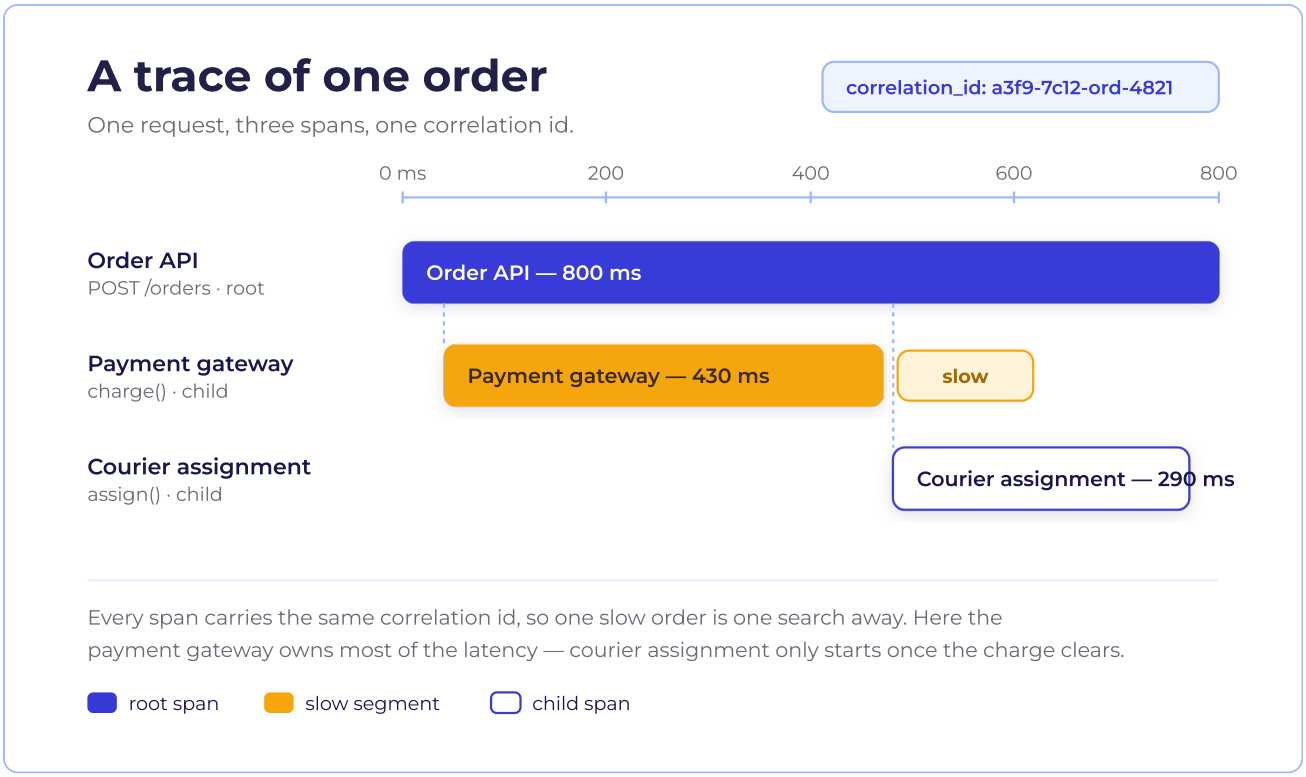
Skip-if: you can't skip metrics on a service that takes orders, but you can skip the elaborate ones. Resist the custom-metric sprawl. Four signals on the order service beats forty dials you never look at during a rush.

Distributed Tracing + Correlation IDs

The problem: placing one order now fans out across the order API, the payment gateway, and the courier-assignment worker, and when it fails you have three disconnected log streams and no way to stitch them into one story. "It's slow somewhere" is not a diagnosis you can act on at 8pm with the kitchen waiting.

A correlation ID is the cheap half: stamp the `order_id` at the edge, attach it to every log line and every `OrderPlaced` message you publish, and one order becomes greppable across every service it touched. Distributed tracing is the structured half: OpenTelemetry propagates a trace context automatically, so each step records a timed span and the backend assembles them into a waterfall showing exactly where the 800ms went between order, payment, and courier. In .NET the trace context rides on `Activity`, and the instrumentation is mostly wiring.

```
builder.Services.AddOpenTelemetry()
    .WithTracing(t => t
        .AddAspNetCoreInstrumentation()
        .AddHttpClientInstrumentation()
        .AddOtlpExporter());
```



What it buys you in production: an order you can follow end to end. The `order_id` lets you pull every line for one failed order across the order, payment, and courier services; the trace shows you which hop was slow, the payment charge or the courier match, not just that something was. Because you're exporting OTLP, the standard wire format, you can send the same telemetry to Cloud Trace (X-Ray, Azure Monitor) and switch backends without touching the code.

Skip-if: you're a single process with no outbound calls. If the order service did everything in-process with no payment gateway and no courier worker, a correlation ID on your logs would be plenty and full tracing would be pure overhead. The moment payment and courier assignment became separate hops, the spans earned their keep.

Externalised Configuration

The problem: configuration baked into the image means the same artifact can't move between environments, and a config change forces a rebuild and redeploy. The surge-pricing threshold that kicks in during a rush is exactly this kind of setting: you want to raise it on a busy Friday without shipping a new build, and a connection string compiled into a binary is a connection string in your source history.

Read config from the environment, not the image (12-Factor, "Config"). The container is identical in staging and production; what differs is what the platform injects at start: environment variables, mounted secrets, a config service. .NET's configuration system layers these for you, so the surge threshold can come from `appsettings.json` in development and an environment variable in production with no code change.

```
// Environment wins over the file; surge thresholds and secrets arrive as env vars, never in t
builder.Configuration
    .AddJsonFile("appsettings.json", optional: true)
    .AddEnvironmentVariables();

var ordersConnString = builder.Configuration.GetConnectionString("Orders");
var surgeThreshold   = builder.Configuration.GetValue<double>("Pricing:SurgeThreshold");
```

What it buys you in production: one image you can promote unchanged from staging to production, a surge threshold you can dial during a rush without a redeploy, and secrets that live in a secret manager (GCP Secret Manager, AWS Secrets Manager, Azure Key Vault) rather than in a layer anyone who pulls the image can read. It also makes diagnostics honest: when pricing behaves differently between environments, the difference is in config you can inspect, not in a binary you have to decompile.

Skip-if: there's no skipping this one for anything you deploy more than once. The closest thing to a skip is a throwaway prototype that only ever runs on your laptop. Everything else reads its config from the outside. This habit gets reinforced as a first-class hosting concern in the next chapter.

Alerting & SLOs

The problem: alerts wired to raw thresholds page you for things that don't matter, and an on-call who's been woken six times by noise will sleep through the seventh page that was real. Alert fatigue is how teams miss the outage they were paged about.

Page on objectives, not noise (Google SRE). An SLO is a target you commit to: "99% of orders are confirmed within 10 seconds over a rolling 28 days." That gives you an error budget, the small allowance of slow or failed confirmations you're permitted, and you alert when you're burning through it fast enough to miss the objective. A single order that takes twelve seconds is not a page. A burn rate that will exhaust the month's budget by Tuesday is.

The mechanics matter less than the discipline. Tie every page to a customer-facing objective, route everything else to a dashboard or a ticket, and make the rule simple: if a page doesn't demand a human act now, it isn't a page.

> *If an alert doesn't require someone to do something right now, it's a dashboard, not a page.*

What it buys you in production: an on-call rotation a small team can actually sustain through a dinner rush. People trust the pager because it only fires when the confirm-within-10-seconds objective is genuinely at risk, and the error budget turns "are we fast enough" into a number you can point at instead of an argument you keep having.

Skip-if: you have no on-call and no confirmation commitment yet. A pre-launch pilot in one neighbourhood doesn't need an error budget. The moment someone is carrying a pager, define the objective first, then wire the alert to it, never the other way around.

Audit Logging

The problem: a customer disputes a charge, a restaurant swears it never cancelled the order, and a chargeback review asks "who refunded this order, and when." Ordinary application logs rotate away before you can answer (OWASP Logging Cheat Sheet). An operational log and an audit trail have different lifetimes and different consumers.

Keep a separate, append-only record of consequential actions: who, what, when, against which entity. Make it immutable; you write to it, you never update or delete from it. And store references, not payloads. Echoing book #1's rule, store decisions, not documents: record that agent 88 refunded order 4471 for tenant 12 at a timestamp, not a copy of the order's contents and certainly not the customer's card number or address sitting in your audit table forever.

```
await audit.Record(new AuditEntry(
    Actor:   currentUser.Id,
    Action:  "order.refunded",
    Subject: order.Id,          // a reference to the order, not its body
    Tenant:  order.TenantId,
    At:      DateTimeOffset.UtcNow));
```

What it buys you in production: an answer to the "who refunded this" or "who cancelled that" question that holds up in a chargeback dispute or an incident, without turning your audit store into a second copy of your orders and a fresh pile of customer PII to protect. The append-only shape means the trail itself is trustworthy: nobody can quietly edit history after a disputed cancellation.

Skip-if: nothing you do is consequential or regulated. A read-only menu browser with no privileged actions has nothing to audit. The day your service lets someone refund a charge, cancel another party's order, or change access, you need the trail, and you need it from the first such action, not retrofitted after the first dispute.

Honorable Mentions

Log Aggregation is the piece that makes structured logging pay off at scale: ship every container's logs to one searchable place (Cloud Logging, CloudWatch Logs, Azure Monitor) so you're not SSH-ing into instances that scale to zero. On most platforms it's a default you turn on, which is why it sits just outside the core.

Synthetic Monitoring runs a scripted order against production on a schedule and alerts when the real customer journey breaks, catching the failures that your internal metrics miss because no real customer has hit them yet. Worth adding once you have a critical path, place-an-order being the obvious one, that you can't afford to discover is broken from a support ticket.

RED and USE are two lenses on the golden signals worth knowing by name. RED (Rate, Errors, Duration, from Tom Wilkie) is the request-centric view for services; USE (Utilisation, Saturation, Errors, from Brendan Gregg) is the resource-centric view for machines and pools. They're complements to the four signals, not replacements, and naming them helps a team agree on what to watch.

All of this has to run somewhere, and the next altitude is how you host it without marrying one cloud.

Hosting: Cloud-Agnostic by Default

The top altitude. The patterns that make the same container run on any cloud, scale to nothing when idle, and roll out without taking the service down.

You have an order service that is testable, resilient, and observable. Now it has to run somewhere. The trap at this altitude is the one the cloud vendors quietly want you to fall into: build for their managed runtime, wire in their proprietary glue, and discover two years later that "moving clouds" means a rewrite. A small team cannot afford that bet. The defence is older and duller than any vendor's roadmap, and it fits on a single page.

The stance for the whole chapter is **cloud-agnostic, container-first**. The container is the portability boundary. Everything specific to one cloud lives outside it, in configuration and infrastructure code you own. Get that boundary right and the same order-service image runs on Cloud Run (AWS App Runner/ECS, Azure Container Apps) without you noticing which one you picked this morning.

The Container as the Unit

Your build artifact is one thing, and it is an OCI image. Not a zip of DLLs that the ops person unpacks onto a VM, not a deployment script that assumes a particular host. The unit you ship, test, and run in every environment is the same image, built once.

The discipline that makes this work is the **Twelve-Factor App** (Wiggins/Heroku, 2011). Treat config as environment, the build as a one-time step, the run as a separate one, and the process as disposable. The Dockerfile is where most of that lands.

```
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src
COPY . .
RUN dotnet publish -c Release -o /app

FROM mcr.microsoft.com/dotnet/aspnet:9.0
WORKDIR /app
COPY --from=build /app .
EXPOSE 8080
ENTRYPOINT ["dotnet", "OrderService.dll"]
```

What it buys you in production: one artifact, identical from your laptop to staging to live. The "works on my machine" failure mode disappears because the machine travels with the code. The image is also the contract every later pattern in this chapter leans on. Scale-to-zero, orchestrator-agnostic deploy, blue-green rollout, all of them assume the unit is an image you can start, stop, and replace at will.

> *The container is the only thing you ship. If the cloud knows anything about you that the image doesn't, you have a lock-in bug.*

Skip-if: you genuinely have one long-lived server and no second environment, and you never will. Then a published binary and a service manager is less to carry. The moment a second environment or a second instance appears, the image earns its keep.

Stateless + Externalised State

Treat instances as cattle, not pets (the phrase is folklore, the discipline is twelve-factor's). Any one of them can vanish mid-request and be replaced by an identical one, and nothing important is lost. That only holds if the instance keeps no state worth keeping. A half-built cart, an `Order` still being placed, an in-memory menu cache the next request depends on: all of it has to live somewhere the instance doesn't.

So you push state out. The customer's cart and order state go to a shared store (Memorystore for Redis; ElastiCache, Azure Cache for Redis) or straight to the `OrderGateway`. Uploaded restaurant photos go to object storage (Cloud Storage; S3, Azure Blob). Anything you were tempted to hold in a static field gets a real home outside the process.

```
// Not this — lost the instant the instance is replaced
private static readonly Dictionary<string, Cart> Carts = new();

// This — state lives outside the process, any instance can serve any request
public async Task<Cart?> GetCart(string id)
{
    var bytes = await _cache.GetAsync($"cart:{id}");
    return bytes is null ? null : Deserialize<Cart>(bytes);
}
```

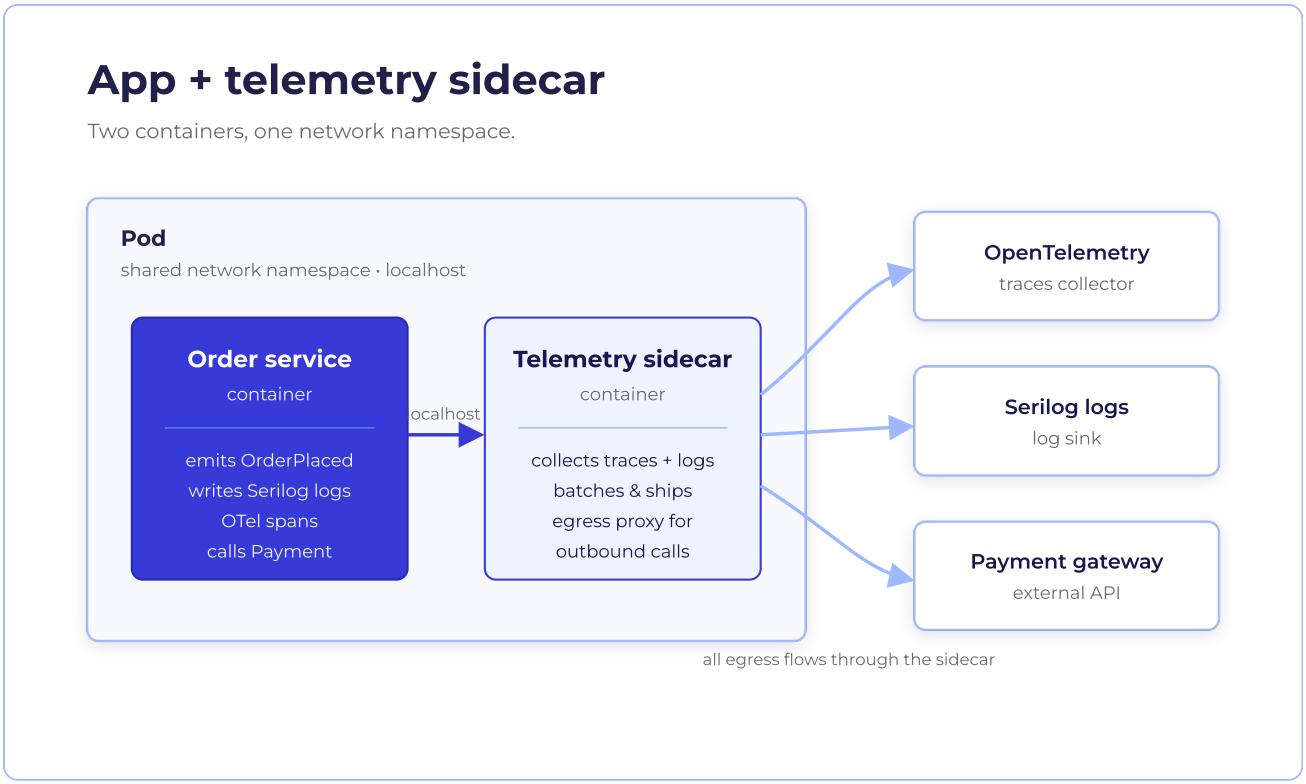
What it buys you in production: when the dinner rush hits you scale out by adding identical instances and scale in by killing them, with no sticky sessions and no "which box has my cart" debugging at 3am. It is the precondition for every elastic behaviour the cloud offers. Stateful instances quietly cap you at one.

Skip-if: nothing. This is the price of admission for the rest of the altitude. If an instance holds state that can't be rebuilt or refetched, fix that before you reach for autoscaling, because autoscaling will simply expose it as data loss.

Sidecar / Ambassador

Some concerns belong next to your order service but not inside it: a metrics exporter, a log shipper, a proxy that handles mTLS or retries to a flaky payment gateway. Bolting them into your application code couples your release cycle to theirs and bloats the image with things that have nothing to do with placing an order. The **Sidecar** pattern (Azure Cloud Design Patterns) runs them as a separate container in the same deployment unit, sharing the network and lifecycle but not the codebase. The OpenTelemetry traces and Serilog logs you wired at the observability altitude leave the order container and land in a telemetry sidecar that ships them onward.

The **Ambassador** is the outbound-facing variant: a local proxy your service talks to as if it were the remote dependency, while the proxy handles connection management, retries, and routing. This is the same idea as the Proxy pattern from the object altitude, lifted to the deployment level. A stand-in that controls access, except now it is a container, not a class.



What it buys you in production: cross-cutting infrastructure you can upgrade independently, written once and reused across the order, menu, and courier services in whatever language. Your application stays about the order. The sidecar handles the plumbing.

Skip-if: you run on a platform that already gives you the concern for free. Cloud Run and Container Apps inject telemetry and ingress without you managing a sidecar; the explicit pattern earns its place on Kubernetes, where the pod is the unit and the sidecar is idiomatic. Don't add a proxy container to chase a problem the platform already solved.

Scale-to-Zero (and Graceful Shutdown)

A worker that costs nothing when no one is using it is a small team's quiet advantage. Take the nightly analytics and report worker that rolls up the day's orders per restaurant: it does real work for an hour after close and sits idle the rest of the day. **Scale-to-Zero** (Knative, the model behind Cloud Run; AWS App Runner, Azure Container Apps) drops its running instance count to zero between runs and spins one back up on the next message or request. You pay for the rollup, not for idle CPU through the small hours. The tradeoff is the cold start: the first request after an idle period waits for an instance to boot.

The half of this pattern people skip is the half that bites. When the platform scales you down, reclaims an instance, or rolls a new revision, it sends **SIGTERM** and then waits a grace period before SIGKILL. If the order service ignores SIGTERM mid-rush, in-flight orders get cut before they are written and queued courier-assignment work is lost. Graceful shutdown means catching that signal, refusing new work, and draining what you already accepted.

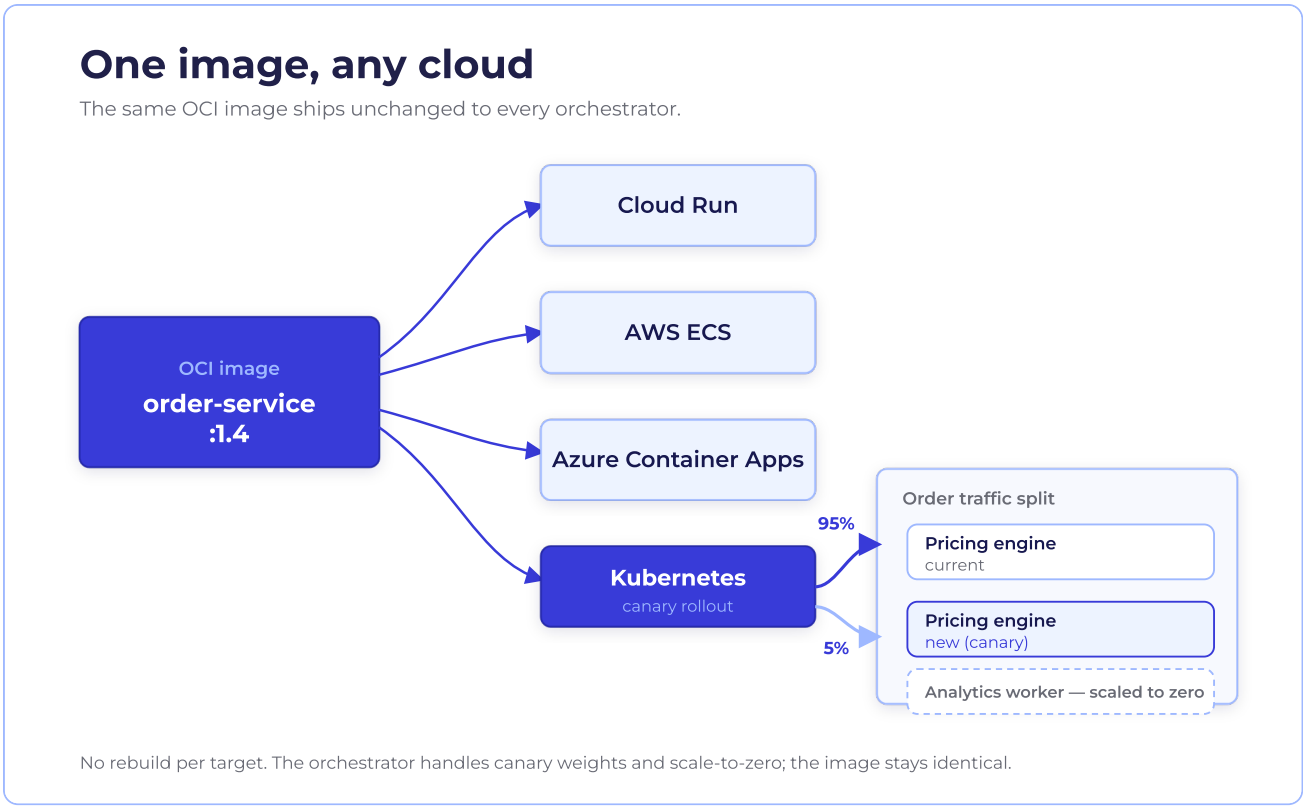
```
// ASP.NET Core honours SIGTERM through IHostApplicationLifetime.
// Give in-flight requests time to finish before the host exits.
builder.Services.Configure<HostOptions>(o =>
    o.ShutdownTimeout = TimeSpan.FromSeconds(25));

var app = builder.Build();
var lifetime = app.Services.GetRequiredService<IHostApplicationLifetime>();
lifetime.ApplicationStopping.Register(() =>
    app.Logger.LogInformation("SIGTERM received, draining in-flight work"));
```

What it buys you in production: near-zero cost for the off-peak worker, and clean reclaims that don't drop a single order when the platform decides to move you. The two halves are one pattern. Elasticity you can't shut down gracefully is just a faster way to lose orders.

Skip-if: the order API itself, which is latency-critical and never truly idle through the lunch-to-dinner stretch, where the cold-start penalty on a first request costs more than the idle instance saves. Keep a warm minimum there and let the analytics worker scale to zero. Drain on SIGTERM regardless, because reclaims and rollouts happen whether or not you ever scale to zero.

Orchestrator-Agnostic Deploy

Here is where cloud-agnostic stops being a slogan and becomes a property you can test. Because the unit is an OCI image and the cart, the `Order`, and the menu live outside it, the same order-service image runs unchanged on Cloud Run, on ECS, on Container Apps, or on Kubernetes. The orchestrator's job is to start your container, route traffic to it, restart it when it dies, and read its health endpoint. Those are standard verbs. Nothing in your code names the platform.



The deployment differences live in thin, swappable wrappers: a Cloud Run service YAML, an ECS task definition, a Container Apps manifest, a Kubernetes Deployment. Each says the same things in a different dialect: which image, which port, how much memory, which health path, how many instances. You write the wrapper once per target and the image is the constant.

What it buys you in production: a real exit. When the bill, the region story, or a client's compliance requirement pushes you to another cloud, you move the wrapper, not the application. For a fractional team with no platform group to run a migration project, that optionality is the whole point of the altitude.

> *Lock-in is rarely a decision. It's the accumulation of small assumptions that each cloud invited you to make. The container boundary is where you refuse them.*

Skip-if: you are certain of one cloud forever and the proprietary primitives genuinely buy you something the portable path can't. Be honest that "certain forever" is a strong claim, and that the portable path costs little to hold open.

Infrastructure as Code

A reproducible environment you can recreate from a file is how a small team stays cloud-agnostic without a platform team. **Infrastructure as Code** (Fowler, *InfrastructureAsCode*) means the order service, its orders database, its `OrderPlaced` queue, and its secrets are declared in version-controlled files (Terraform, Pulumi, Bicep), applied by a tool, and reviewed like any other change. The click-ops console is where reproducibility goes to die: nobody remembers which checkbox in which dropdown made staging different from production.

```
resource "google_cloud_run_v2_service" "order_service" {
  name     = "order-service"
  location = var.region
  template {
    containers {
      image = var.image
      ports { container_port = 8080 }
    }
    scaling { min_instance_count = 1 }
  }
}
```

What it buys you in production: a new environment in minutes instead of a day of console archaeology, a diff you can review before it touches anything, and a blast radius you can reason about. When the same code provisions staging and production, the gap between them stops being a source of 3am surprises. It is also where orchestrator-agnostic deploy becomes routine, because the per-target wrapper is just another file under review.

Skip-if: a weekend prototype with one manually clicked service that you will throw away. The instant a second environment or a second person appears, ad-hoc console state becomes the most expensive thing you own. Codify before then.

Blue-Green / Canary Deploy

Shipping a new version should not require a maintenance window, and a bad version should not require a panic. Take the change that scares you most: a rewritten pricing engine, the one that computes surge, promo, and loyalty on top of the menu price. Get it wrong and every customer sees a wrong total. **Blue-Green Deployment** (Fowler, *BlueGreenDeployment*) keeps two production environments: blue runs the current pricing engine, green holds the new one.

You deploy to green, smoke-test it against real menus and real tax rules, then switch traffic in one move. If green misprices, you switch back. The rollback is instant because the old version never went away.

Canary Release (Danilo Sato) is the gradual cousin: route the new pricing engine to 5% of orders, watch the golden-signal metrics from the observability altitude plus your own "totals matched" check, and ramp up only if error rates and latency hold. If they don't, 5% of orders saw the bug, not 100%. Most managed runtimes give you traffic splitting natively, so this is configuration, not a custom router.

```
# Cloud Run: deploy the new pricing engine without traffic, then send it 5% of orders
gcloud run deploy order-service --image $IMAGE --no-traffic
gcloud run services update-traffic order-service \
  --to-revisions LATEST=5
```

What it buys you in production: zero-downtime rollout and a rollback measured in seconds, not in how fast someone can rebuild the previous pricing engine while the totals are wrong and support is filling up. The canary turns a deploy from an act of faith into a measured one, watched by the metrics you already wired up at the observability altitude.

Skip-if: an internal restaurant-onboarding tool with a handful of staff users where a few seconds of restart costs nothing. A rolling restart is simpler and it's fine. The pattern earns its place when downtime has a real cost or a rollback has to be instant, which a customer-facing pricing change always is.

Sidebar: Secrets Management

Secrets are the one piece of externalised state you must not treat like ordinary config. Deep security was book #1's job; the hosting-altitude rule is short. The orders-database password, the payment-gateway API key, the token-signing key: none of them belongs in the image, in the repo, or in a plaintext environment variable baked into a manifest. Put each in a managed secret store (GCP Secret Manager; AWS Secrets Manager, Azure Key Vault) and let the runtime inject it at start, so the order-service image stays portable and the secret stays out of source control.

```
// Orders-DB connection pulled from the platform's secret store at startup, never committed
var connString = builder.Configuration["Secrets:OrdersDb"];
```

Two habits cover most of the risk: rotate secrets on a schedule so a leak has a short half-life, and grant each service read access only to the secrets it needs. The courier service has no business reading the payment key. That keeps secrets cloud-agnostic in the same way everything else here is: the application asks for a value by name and never knows which vault answered.

Honorable Mentions

Four patterns that just missed the cut, for the reader going further. **Feature Flags** (Fowler, *feature toggles*) decouple deploy from release: ship the new pricing engine dark, turn it on for one city, and kill it without a rollout. They pair naturally with canary and are the cheapest way to de-risk a change once you have more than a trickle of orders. **Gateway / Backend-for-Frontend** (Sam Newman) puts a tailored API edge in front of your services, one per client type. The customer app and the courier app want very different things from the same `Order`: the customer wants the menu, the running total, and a live map; the courier wants the pickup address, the route, and a single "picked up" button. A shared contract that serves both ends up bloated for each. A customer BFF and a courier BFF each shape the order data to one screen, and neither leaks the other's concerns into the core service.

In the front-end. This is where the client stack meets the hosting altitude. The customer app (React / TypeScript) talks only to its BFF and never to the order, menu, and courier services directly; the courier app talks only to its own. Each BFF is the seam where one client's view of the `Order` is assembled, so a change to the courier screen never risks the customer total.

Secrets Management sits here too as a full pattern, expanded just above into the sidebar it deserves at this altitude. **Service Discovery** lets the order, menu, and courier services find each other by name rather than by hardcoded address. On the managed runtimes in this book it's usually handled for you (Cloud Run URLs, internal DNS), which is exactly why it's a mention and not a slot. Reach for the explicit pattern on Kubernetes or a service mesh, and not before three services have turned into thirty.

Seven altitudes, one working vocabulary. Now watch it compose into one real service.

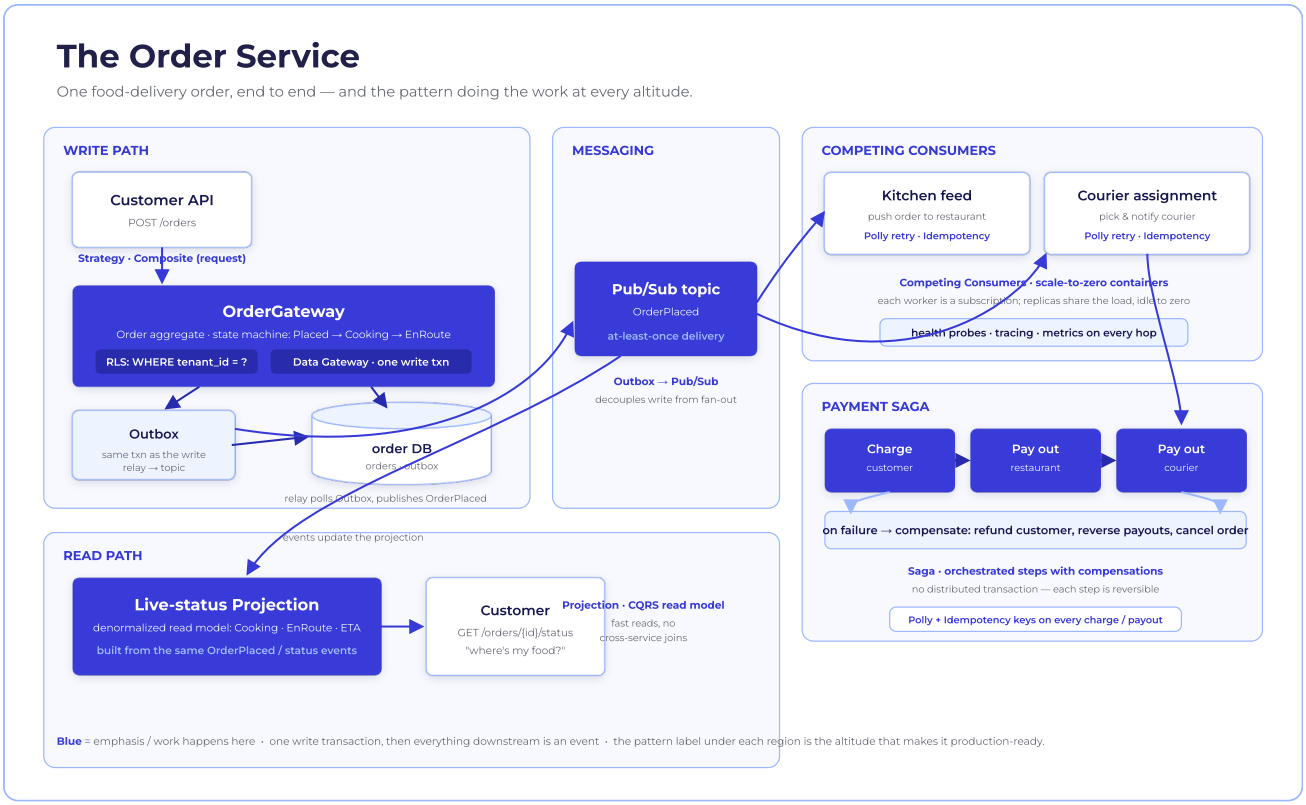
A Reference Service

Seven altitudes, one service. The food-delivery Order Service walked end to end, with every pattern annotated where it actually sits in the code.

You have climbed the ladder. Every pattern with its payoff and its skip-if. Read in isolation they are a vocabulary; the question a practitioner actually asks is how they fit together in one running thing. So here is one running thing, and it is the same app you have met in every chapter so far.

A customer opens the food-delivery app, builds an order from a restaurant's menu, and taps place. From there the work fans out. The order is written and an `OrderPlaced` event leaves through the Outbox. A kitchen worker and a courier-assignment worker pick it up. A payment Saga charges the customer and pays out the restaurant and the courier. The customer watches live status climb from placed to delivered. Three sides touch this: the customer, the restaurant (the tenant), and the courier. One slice of the marketplace, assembled.

We will walk it in the order an order flows: in through the API, down to the database, out to the queue, into the workers, through the payment Saga, and back to the customer as live status. At each stop, the pattern doing the work is named where it lives. Nothing here is a toy fragment. It is the shape you would copy.



The place-order path: API, gateway, outbox

An order comes in over HTTP. The endpoint does almost nothing itself; it hands the request to a mediator and returns. That thin edge is deliberate. The Mediator (MediatR, Jimmy Bogard) keeps the controller from knowing anything about how an order is stored or queued, and the command/query split means the read side never touches the write side.

```
app.MapPost("/orders", async (PlaceOrder cmd, IMediator mediator, CancellationToken ct) =>
{
    var id = await mediator.Send(cmd, ct);
    return Results.Accepted($"/orders/{id}", new { id });
});

app.MapGet("/orders/{id:guid}", async (Guid id, IMediator mediator, CancellationToken ct) =>
    await mediator.Send(new GetOrderStatus(id), ct) is { } status
        ? Results.Ok(status)
        : Results.NotFound());
```

The handler is where the persistence altitude shows up. Two patterns share one database transaction: the `OrderGateway`, a Data Gateway (Fowler, *Patterns of Enterprise Application Architecture*) that owns the SQL, and the Outbox (Richardson, microservices.io) that records the `OrderPlaced` intent-to-publish in the same commit. This is the part people get wrong. If you write the order row and then publish to the queue as a separate step, a crash between them either drops the order or duplicates it. The customer is charged for food no kitchen ever sees, or the kitchen cooks an order twice. The Outbox closes that gap by making the message part of the write.

```
public sealed class PlaceOrderHandler(IOrderGateway gateway) : IRequestHandler<PlaceOrder, Gu
{
    public async Task<Guid> Handle(PlaceOrder cmd, CancellationToken ct)
    {
        var order = Order.Place(cmd.RestaurantId, cmd.Lines, cmd.Total); // State: starts Pla
        await gateway.InsertWithOutbox(order, ct);                       // Data Gateway + O
        return order.Id;
    }
}
```

The menu those `Lines` came from is a Composite (GoF): sections hold items, items hold modifier-groups, modifier-groups hold modifiers, and a price or an availability flag rolls up the same way at every level. The customer priced the order by walking that tree on the client; the server stores the flat lines it produced. The Composite earns its keep on the read side of the menu, not here, but it is the structure the order references.

The gateway itself is hand-written SQL behind a thin interface, calling a stored procedure through Dapper. No ORM, no change-tracking. The procedure inserts the order and the outbox row inside a single transaction, so either both land or neither does. Tenancy is not passed as a parameter you might forget; the restaurant is the tenant, and `tenant_id` is set as a session variable that Row-Level Security reads.

```
public sealed class OrderGateway(IDbConnectionFactory factory, ITenantContext tenant) : IOrder
{
    public async Task InsertWithOutbox(Order order, CancellationToken ct)
    {
        using var conn = await factory.OpenAsync(ct);
        using var tx = await conn.BeginTransactionAsync(ct);
        await conn.ExecuteAsync(
            "select set_config('app.tenant_id', @value, true)",
            new { value = tenant.Id.ToString() }, tx);            // RLS sees this for the tx
        var msgId = Guid.NewGuid();
        var payload = JsonSerializer.Serialize(new OrderPlaced(order.Id, order.RestaurantId, 
        await conn.ExecuteAsync("place_order",
            new { order.Id, Tenant = order.RestaurantId, order.Total, MsgId = msgId, Payload =
            tx, commandType: CommandType.StoredProcedure);        // inserts order + outbox r
        await tx.CommitAsync(ct);
    }
}
```

> *If the `OrderPlaced` message isn't written in the same transaction as the order, you don't have a queue. You have a race.*

Multitenancy here is the dense default from the persistence chapter: one database, one schema, a `tenant_id` column on every restaurant-owned table, and a Row-Level Security policy (PostgreSQL RLS) that filters every query to the current tenant. The application sets `app.tenant_id` once per request; the database enforces isolation whether the developer remembers to filter or not. A query for one restaurant's orders can never return another's, even when the code is wrong. That is the whole point of putting the guarantee in the engine rather than the query.

The relay: outbox to queue

A row in an outbox table is not yet on a queue. A small relay polls the table, publishes each pending `OrderPlaced` message to the broker, and marks it sent. This is where Queue-Based Load Levelling (Azure Cloud Design Patterns) begins: the API accepts orders at whatever rate

the lunch rush throws at it, and the queue absorbs the spike so the workers downstream see a steady stream instead of a flood.

```
public async Task Relay(CancellationToken ct)
{
    foreach (var msg in await _outbox.PendingBatch(ct))
    {
        await _publisher.Publish(msg.Topic, msg.Body, ct);   // GCP Pub/Sub (SQS/SNS · Servic
        await _outbox.MarkSent(msg.Id, ct);
    }
}
```

`OrderPlaced` is published once and fanned out to several subscribers: the kitchen, courier assignment, the customer's notifications, and analytics. That is Publish/Subscribe (Azure Cloud Design Patterns) doing the fan-out the marketplace needs. The broker is named cloud-agnostically: GCP Pub/Sub first, with AWS SQS/SNS and Azure Service Bus as the parenthetical equivalents. The relay code does not care which; it talks to an `IMessagePublisher`. Swapping the broker is an adapter change, not a rewrite.

The kitchen and courier workers: competing consumers, strategy

The workers are separate processes. Run one courier-assignment worker, run twenty; they pull from the same subscription and split the work between them without coordinating. That is Competing Consumers (Azure Cloud Design Patterns), and it is the reason this design scales out instead of up. Dinner rush hits, add workers.

The kitchen worker confirms the order with the restaurant and advances its state. The courier-assignment worker decides who delivers it, and that decision is a Strategy (GoF). A `CourierMatchingStrategy` picks the courier; nearest, fastest, or cheapest are interchangeable implementations chosen per restaurant or per city. Adding a new matching rule is a new strategy and a registration, not a change to the assignment loop.

```
public interface ICourierMatchingStrategy
{
    Task<CourierId?> Match(Order order, CancellationToken ct);  // nearest / fastest / cheapes

}

public sealed class CourierAssignment(ICourierMatchingStrategy strategy)
{
    public async Task<CourierId?> Assign(Order order, CancellationToken ct) =>
        await strategy.Match(order, ct);   // swap the rule without touching this caller
}
```

The lifecycle of an order is a State machine (GoF). It moves Placed → Confirmed → Preparing → Ready → PickedUp → Delivered, with Cancelled reachable from the early states, and the transitions are explicit rather than a pile of boolean flags. Each transition is a fact worth recording, which is why this pattern ties so cleanly to the event log behind the live-board view. The kitchen worker drives the cooking transitions; the courier worker drives pickup and delivery.

```
public async Task Handle(OrderPlaced evt, CancellationToken ct)
{
    var order = await _gateway.Load(evt.OrderId, ct);
    await _gateway.Save(order.Confirm(), ct);     // State: Placed → Confirmed
    await _gateway.Save(order.StartPreparing(), ct); // State: Confirmed → Preparing
}
```

When a message fails past its retries, it goes to a dead-letter queue rather than vanishing or blocking the line behind it. Backpressure and the dead-letter queue (Enterprise Integration Patterns) are what stop one malformed order from stalling every kitchen worker. The failure is visible, parked, and replayable, which matters more than it sounds at the peak of a Friday-night rush.

The payment Saga: charge, pay out, compensate

Placing an order is not one database write. It is a distributed transaction across the payment provider, the restaurant's payout account, and the courier's. No single commit spans them, so

coordination falls to a Saga (Garcia-Molina & Salem; Richardson). The Saga charges the customer, pays out the restaurant, pays out the courier, and on any failure runs the compensations in reverse: refund the customer, claw back a payout. A process manager tracks which step the order is on so a crash resumes rather than restarts.

```
public async Task Run(Guid orderId, CancellationToken ct)
{
    var charge = await _payments.Charge(orderId, ct);          // step 1
    try
    {
        await _payouts.PayRestaurant(orderId, ct);             // step 2
        await _payouts.PayCourier(orderId, ct);                // step 3
    }
    catch
    {
        await _payments.Refund(charge.Id, ct);                 // compensate step 1
        throw;
    }
}
```

The charge call is the riskiest hop in the whole service, so it runs through a Polly resilience pipeline: a retry with exponential backoff and jitter (Brooker, AWS) wrapped by a circuit breaker (Nygard, *Release It!*) and a timeout. The retry rides out a payment-gateway hiccup; the breaker stops hammering a provider that is genuinely down; the timeout keeps a slow provider from holding an order hostage.

```
_pipeline = new ResiliencePipelineBuilder()
    .AddRetry(new RetryStrategyOptions
    {
        MaxRetryAttempts = 3,
        BackoffType = DelayBackoffType.Exponential,
        UseJitter = true
    })
    .AddCircuitBreaker(new CircuitBreakerStrategyOptions
    {
        FailureRatio = 0.5,
        BreakDuration = TimeSpan.FromSeconds(30)
    })
    .AddTimeout(TimeSpan.FromSeconds(10))
    .Build();
```

Retries are only safe because the charge is idempotent, keyed on the order id. A retry that lands after the provider already took the money returns the original charge instead of taking it twice. Without that key, every backoff is a chance to double-charge a customer for one dinner.

```
public async Task<Charge> Charge(Guid orderId, CancellationToken ct) =>
    await _pipeline.ExecuteAsync(t =>
        _provider.Charge(orderId, idempotencyKey: orderId.ToString(), t), ct);
```

> *A retry without an idempotency key is a coin-flip on whether the customer pays once or twice. Key the charge, or don't retry it.*

The read path: a projection, not the log

The customer does not wait on any of this synchronously. The app polls `GET /orders/{id}` and reads status from a projection: a live read model the workers keep current as they advance the order's state. The place-order path writes; the read path reads its own shape, the one the tracking screen wants. That separation is CQRS (Greg Young; Fowler's bliki), and the order-history and restaurant live-board views are further Materialized Views (Fowler; Azure Cloud Design Patterns) over the same event stream.

```
public sealed class GetOrderStatusHandler(IOrderReadModel reads)
    : IRequestHandler<GetOrderStatus, OrderStatusDto?>
{
    public Task<OrderStatusDto?> Handle(GetOrderStatus q, CancellationToken ct) =>
        reads.StatusById(q.OrderId, ct);   // a thin projection query, tenant-scoped by RLS
}
```

The same Row-Level Security policy that protected the write protects this read. A customer reads only their own order; a restaurant's live-board reads only its own tenant's orders, and no developer had to remember to add `WHERE tenant_id = @id`. The isolation is structural.

Seeing it run: health, traces, metrics

None of the above is operable unless you can see it. The observability altitude threads through every component at once. A health endpoint (Health Endpoint Monitoring, Azure) tells the orchestrator whether to send traffic and whether to restart. Liveness says the process is alive; readiness says it can reach its database and broker.

```
builder.Services.AddHealthChecks()
    .AddNpgSql(cfg.GetConnectionString("Orders")!, name: "db")
    .AddCheck<BrokerHealthCheck>("broker");

app.MapHealthChecks("/health/live",  new() { Predicate = _ => false });
app.MapHealthChecks("/health/ready", new() { Predicate = c => c.Tags.Contains("ready") });
```

Distributed tracing (OpenTelemetry) follows a single order from the HTTP place, through the queue, into the kitchen and courier workers, through the payment Saga, by carrying a correlation id across the broker boundary. Without it, a charge that failed in the Saga is unconnectable to the order that triggered it. With it, one trace tells the whole story: placed, confirmed, charged, assigned, delivered. The metrics are the four golden signals (Google SRE): latency, traffic, errors, saturation. Queue depth is the saturation signal that matters most here, because a queue that only grows during a rush is a worker fleet that cannot keep up.

```
builder.Services.AddOpenTelemetry()
    .WithTracing(t => t.AddAspNetCoreInstrumentation().AddNpgsql().AddOtlpExporter())
    .WithMetrics(m => m.AddMeter("orders.worker").AddOtlpExporter());

// in the worker: the saturation signal that actually predicts a backed-up kitchen
_queueDepth = meter.CreateObservableGauge("orders.queue.depth", () => _broker.ApproximateDeptl
```

Structured logs (Serilog) carry the same correlation id and the `tenant_id` on every line, keyed on `order_id`, so a support question about one customer's missing dinner is a filter, not an archaeology dig. The trail records references and decisions, not payloads, keeping it useful without turning the log into a copy of the orders table.

Where it lives: a stateless container that scales to zero

The whole thing ships as a container (the 12-Factor App). State lives outside the image, in the database and the broker, so any instance can serve any order and the orchestrator can kill and restart instances freely. Because the workers hold nothing local, the platform can scale them to zero between meal rushes and back up when the queue fills (Cloud Run; AWS App Runner, Azure Container Apps). The analytics worker, idle most of the afternoon, scales all the way down.

```
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base
WORKDIR /app
COPY --from=build /publish .
ENV ASPNETCORE_URLS=http://+:8080
EXPOSE 8080
ENTRYPOINT ["dotnet", "OrderService.dll"]
```

Scale-to-zero only works if the container shuts down cleanly. On SIGTERM the worker stops pulling new messages, lets the in-flight order finish its current step, and exits, so the platform never kills work mid-charge. The host's graceful-shutdown hook does exactly that, and the `IHostedService` honouring the cancellation token is what makes it safe.

```
public async Task StopAsync(CancellationToken ct)
{
    _accepting = false;                       // stop pulling new orders
    await _inFlight.WhenAllOrTimeout(ct);     // let the current step finish, then exit
}
```

The same image runs on any of the three clouds because nothing in it is tied to one. The broker, the database, and the secrets come from configuration, not from compile-time choices. That is the cloud-agnostic stance made concrete. You build one artifact and point it at whichever target you are deploying to, with no rewrite in between.

The ladder as one picture

Step back and the whole stack is visible in one service. Dependency Injection (Fowler's *Inversion of Control* article; the "D" of Robert C. Martin's SOLID) and the Mediator wire the inside, while the `OrderGateway` and RLS hold the data. The Outbox, Pub/Sub, and Competing Consumers move the work, the payment Saga coordinates the money, and Polly keeps the charge safe when the provider wobbles. Health checks, traces, and golden-signal metrics make all of it observable, and a stateless container that scales to zero hosts the lot. A Strategy decides which courier gets the order. A State machine runs that order's life from placed to delivered, against the menu a Composite priced in the first place.

> *A whole vocabulary of patterns, and one slice of a food-delivery app uses maybe fifteen of them well. That ratio is the book.*

Notice what is *still* missing. Full Event Sourcing never appears, because a projection over explicit state transitions is enough for live tracking; the event log stays a convenience here, not the system of record. Sharding is absent too, since one database holds these orders comfortably until the marketplace spans cities. You will also find no second tenant-isolation tier, no service mesh, and no Claim-Check for the small receipts these orders carry. The Saga earned its place because there genuinely is a distributed transaction, money moving across three parties. The heavier patterns did not, because the order does not yet need them. Every one of those absences was a deliberate skip, and the service is more maintainable for it.

The temptation now is to use all of it. Resist; the next chapter is about the tax you pay when you don't.

The Over-Engineering Tax

Every pattern you adopt is one you maintain forever. This chapter teaches the discipline the rest of the book is incomplete without: knowing when not to reach.

The Order Service in Chapter 10 uses a lot of the ladder. That was the point of it: show the patterns composing cleanly into one shape. The danger is that you read it as a starting template and build all of it on day one, for a food-delivery app with one restaurant signed up and a dozen orders a week.

You won't have over-built because you were careless. You'll have over-built because each pattern, on its own, looked obviously correct. Multitenancy is correct. So is an outbox, and so is a circuit breaker, and that defended judgement on each one is exactly the trap. No single decision is the tax. The sum of them is, taken before the problem that justified each one ever showed up.

Every pattern has a carrying cost. It's code a junior has to understand before they can change anything near it, a failure mode you now own, a line in the runbook, a thing that breaks in a way the boring version never could. A five-person team that adopts a pattern it doesn't yet need hasn't bought insurance. It's bought a second job.

> *A pattern you don't need yet is not free. It is a liability you chose to carry early.*

The deferring question

There's one question that does most of the work here, and you ask it of every pattern before you adopt it: *what breaks if I don't add this yet?*

If the honest answer is "nothing breaks; it would just be tidier" or "nothing breaks until we hit a scale we're nowhere near," you defer. Write down the trigger that would change the answer ("when a restaurant's order data has to live in its own database for compliance," "when the courier-assignment queue regularly backs up past a minute at dinner rush") and move on. The pattern isn't rejected. It's parked, with the condition that un-parks it written next to it.

This is the inverse of the skip-if from Part II. The skip-if tells you a pattern's adoption signal. The deferring question makes you say out loud what it costs you to *not* have it today. Most of the time the cost is zero, and zero is a price worth paying for a smaller system you can hold in your head.

What each pattern costs, and what un-parks it

The over-engineering tax

Pattern	Carrying cost	The trigger that un-parks it
Multitenancy	a second schema/migration path	a second restaurant signs
Event Sourcing	replay + projection machinery	order history becomes the product
Courier-assignment queue	a broker to run	the dinner-rush spike
Service mesh	a control plane to operate	more than a handful of services
Sharding	cross-shard complexity	one database stops coping

Park the pattern until its trigger fires. Until then, the cost is pure carrying tax.

The trap at each altitude

The over-engineering tax has a signature shape at every rung of the ladder. The pattern is real and so is its production payoff; the trap is reaching for that payoff before it exists. What follows is the premature-pattern trap altitude by altitude, each with the question that defers it.

OBJECT

The trap is the abstraction with one implementation. You introduce a `CourierMatchingStrategy` interface while you only ever match couriers one way, a Factory for an object you only ever construct one way, an interface on a class nobody else implements. The indirection costs a reader a jump to a second file to learn there was nothing there.

What breaks if I don't add this yet? Nothing, until a second implementation actually shows up. When you only assign the nearest free courier, a plain method that finds the nearest free courier is the correct amount of code, not technical debt. Add the Strategy seam the day "fastest" and "cheapest" become real options; the change is small, and the second case will tell you the right shape for the seam, which you'd have guessed wrong today. Robert C. Martin's open-closed principle (*Agile Software Development: Principles, Patterns, and Practices*) cuts both ways: a seam you open before there's a second case to vary is open for nothing.

COMPONENT

The trap is structure that outruns the domain: ports and adapters around an Order Service with one adapter, a mediator over three handlers, a full anti-corruption layer for a restaurant POS you call in two places. The cousin every team meets is the heavy ORM, sold as a shortcut and paid back as a maintenance tax you can't reason about. The book's stance is the smaller one: a thin `OrderGateway` over stored procedures, SQL you can read. Fowler's *Patterns of Enterprise Application Architecture* (PoEAA) names the contrast directly: a Table Data Gateway owns one table's SQL, where the Data Mapper an ORM gives you hides that SQL behind change-tracking you don't control.

What breaks if I don't add this yet? Your Order Service stays a service you can read top to bottom. Hexagonal architecture earns its keep when you have real adapters to swap or a domain core worth protecting from I/O. Until then it's ceremony, and ceremony is the thing juniors cargo-cult next.

DATA AND PERSISTENCE

This altitude carries the two heaviest taxes in the book. The first is multitenancy: shared schema, `tenant_id` on every table, row-level security, a filter you can never forget, before you have a second restaurant on the platform. The second is the golden combo, Event Sourcing with CQRS and a materialized read model, stood up for a menu admin that is plainly CRUD.

What breaks if I don't add this yet? For multitenancy: nothing, if you have one restaurant. Book #1 chose single-tenant on purpose and shipped. Onboard the first restaurant with its menu and orders in one place; the move to shared-schema-plus-RLS keyed on `tenant_id` is a migration you do once, when the second restaurant signs, not a tax you pay from commit one.

For Event Sourcing: you keep a database a new hire understands on their first afternoon. Event Sourcing is the book's loudest skip-if. It buys you an audit-grade history and independent read scale, and it charges you eventual consistency, projection rebuilds, versioned events, and a debugging story most teams underestimate. A menu admin that edits prices and toggles items in and out of stock is a handful of tables and a few stored procedures. A `status` column on the `Order` row answers "what state is this in" without an event log behind it. Fowler's own bliki write-up of Event Sourcing is candid about the complexity it adds; reach for events when the history *is* the product, not before.

> *CRUD is not the thing you graduate from. For a menu admin it's the thing you should still be running in year three.*

MESSAGING AND SCALE

The trap is the broker you don't have load for: a courier-assignment queue between two services that exchange a hundred orders a day, or competing consumers scaled out for a workload one process handles in its sleep. The worst version is a saga, the heaviest coordination pattern in the messaging chapter, written for an order flow that is two updates in one database and one transaction.

What breaks if I don't add this yet? Nothing, until a real dinner-rush spike knocks you over or a slow consumer falls behind. Handling `OrderPlaced` with an in-process call is simpler than a queue in every way that matters at low volume: no broker to run, no message to lose, no dead-letter queue to drain, no at-least-once semantics forcing idempotency on you. Add the queue the day the rush is real. Add the order-fulfilment saga when you genuinely have a distributed transaction across charge, courier, and restaurant, which is later than you think and possibly never.

RESILIENCE

The trap here is subtle because resilience patterns feel like pure virtue. A circuit breaker on a dependency that has never once failed. Three retry policies, two timeout budgets, and a bulkhead around a call to the orders database on the same machine. Resilience machinery is itself a source of incidents: a breaker tuned wrong trips under normal load, a retry storm turns a blip into an outage.

What breaks if I don't add this yet? For an in-process or same-region call that doesn't fail, nothing. Resilience is a tax you pay per network boundary, and you pay it where calls actually cross an unreliable boundary and actually fail. The payment gateway, a third party over the public internet, earns the full Polly pipeline of retry, breaker, and timeout. A timeout on every outbound call is cheap and you should have one. That same pipeline wrapped around the local `OrderGateway` call that can't fail is configuration nobody will dare touch in a year.

OBSERVABILITY

This is the altitude where under-building hurts more than over-building, so the trap is shaped differently. The mistake here is rarely too many patterns. It's standing up the platform-grade rig before you have a platform's problems. Think a full SLO regime with error budgets for an internal restaurant dashboard three staff use, custom dashboards nobody reads, distributed tracing wired through an order flow that is still one process.

What breaks if I don't add this yet? Less than at any other altitude, which is the point: get the cheap, high-value pieces in early and defer the rest. Structured logs keyed on `order_id` and

`tenant_id`, a health endpoint, and the four golden signals are not a tax; they're the floor, and you want them from the start. A formal "99% of orders confirmed within 10s" SLO with paging tied to an error budget is worth it once a real customer is paged by its absence. Buy the floor early. Buy the rig when the on-call rotation asks for it.

HOSTING

The trap is the platform-team architecture without the platform team: a service mesh for the three services your food-delivery app actually runs, or Kubernetes for a workload Cloud Run (AWS App Runner, Azure Container Apps) runs with a single command and scales to zero for free. Worst of all is a multi-region active-active deploy for a marketplace whose restaurants and couriers are all in one city and would not notice an hour of downtime a year.

What breaks if I don't add this yet? Nothing a small team can't absorb, and a great deal of operational surface you'd otherwise own. The Order Service as a 12-factor container behind a managed runtime is cloud-agnostic, scales, and rolls back, with no mesh to operate and no control plane to patch at 3am. A config file is a service mesh you don't have to run. One managed Postgres instance is sharding-by-city you've deferred until a single instance genuinely can't hold the orders. Reach up the hosting altitude when the managed boring option visibly runs out of room, not in anticipation of a scale you're modelling on a whiteboard.

The honest case for boring

Notice the pattern across the altitudes. The boring choice isn't a worse version of the sophisticated one. It's a different bet, and at small scale it's usually the better bet.

Single-tenant before multitenancy. CRUD before Event Sourcing. From there the same instinct keeps repeating: a monolith before microservices, a config file before a service mesh, one database before sharding, an in-process call before a queue. Each pair is the same trade. Less to build now, far less to maintain forever, and a clean migration path to the heavier option *if and when* the trigger fires. You are not betting against scale. You're refusing to pay for it before it arrives.

The small-team rule underneath all of this: you maintain everything you ship. There is no platform team to absorb the patterns you adopted speculatively. The breaker you wrapped round the local `OrderGateway` "to be safe" is yours to tune. The order event log you stood up "for flexibility" is yours to rebuild the courier and customer projections against at midnight.

Capacity spent maintaining a pattern you don't need yet is capacity not spent on the feature a restaurant is actually asking for.

> *Maintain what you ship. The corollary is the whole chapter: ship less, so there's less to maintain, so the small team stays fast.*

Production-readiness per unit of team capacity is the selection criterion the whole book runs on, and restraint is half of it. The patterns in Part II raise the numerator. The discipline in this chapter protects the denominator. A team that knows the whole catalogue and deploys six of them on purpose will out-ship a team that deploys everything it knows because it could.

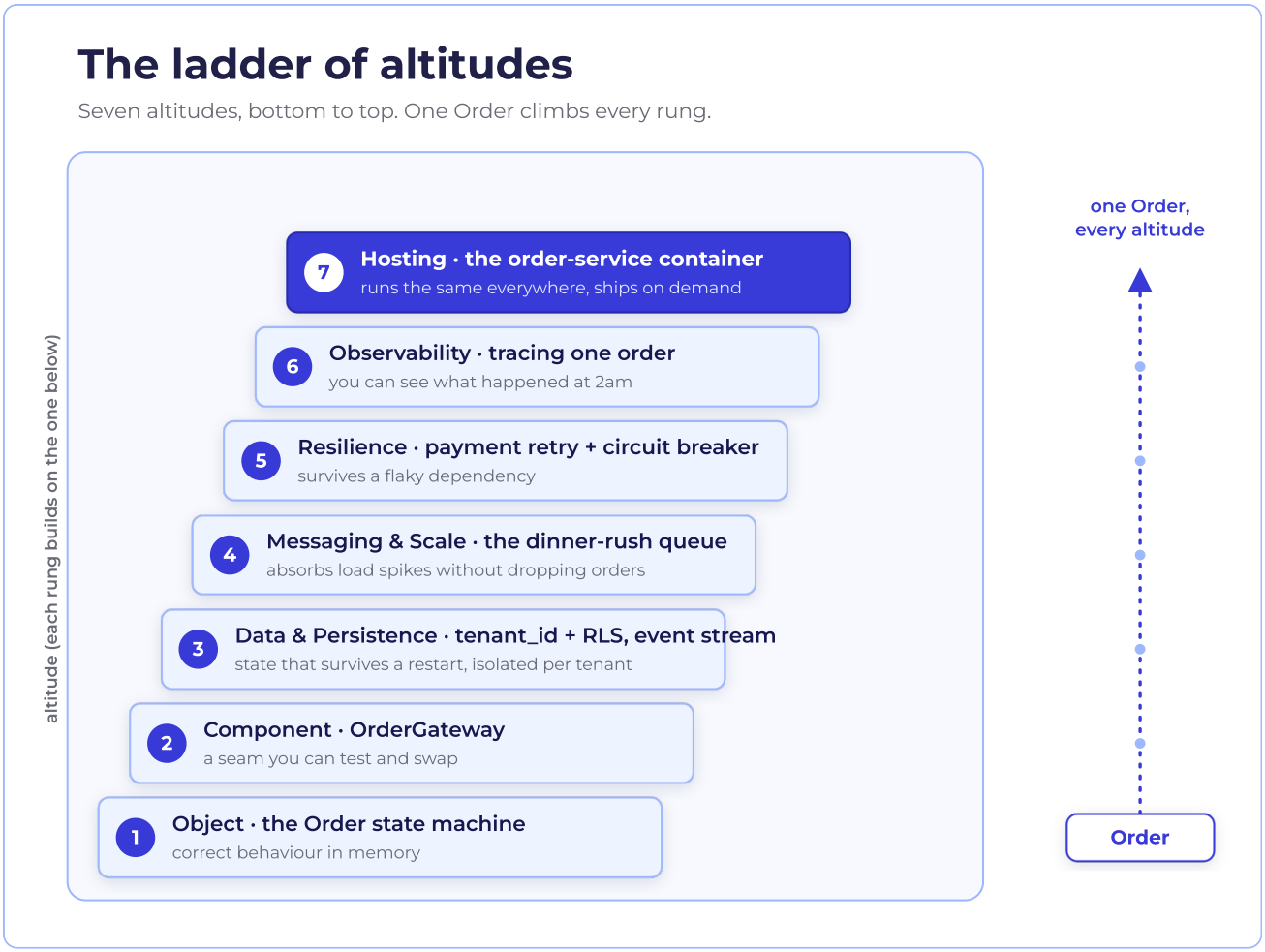
Production-ready is a deliberate bar you clear, not a maximum you chase. That's the note the conclusion lands on.

Conclusion: Production-Ready, Deliberately

Seven altitudes, one working vocabulary, and one food-delivery app that proves it composes. All of it aims at a single bar, and here is the whole spine in one pass.

You started with a catalogue problem. The Gang of Four named twenty-three patterns, the cloud catalogues add dozens more, and a small team that tries to learn all of them ships nothing or over-builds everything. The book's answer was never "learn more." It was "learn the right few, in order, and know what each one costs you to keep."

That few is the Pareto stack: the top patterns at each of seven altitudes, from the object up to the container. One app carried the whole climb so you could watch them line up. A food-delivery marketplace, the kind you have ordered from: a customer places an `Order`, a restaurant prepares it, a courier carries it. At the bottom, object patterns organise the code inside a boundary, and the `Order`'s lifecycle is a State machine while courier selection is a `CourierMatchingStrategy`. Component patterns shape one service so it stays testable, and an `OrderPlaced` domain event fans out to the kitchen. Data patterns hold state and let it evolve, with restaurants as tenants behind a `tenant_id` and Row-Level Security. Messaging moves that state between services without dropping it. A flaky payment gateway becomes a non-event once the resilience patterns sit in front of it. Higher up, observability lets you see inside a running system and wake someone when an order stalls. And the hosting patterns put the whole thing on a cloud without marrying one.



The vocabulary, climbed in order

The order is the point. You don't reach for a Saga before you have a distributed transaction, and you don't reach for Event Sourcing because a blog post made it sound clever. Each altitude assumes the one below it is solid. A Circuit Breaker around an `OrderGateway` that nobody can reason about just wraps a problem in a second problem.

Patterns are vocabulary, not architecture. They give a team a shared language for decisions it still has to make. As Robert C. Martin argues in *Clean Code*, the names carry the design intent. `CourierMatchingStrategy` doesn't decide which courier gets the order; it names the shape so two engineers can talk about nearest-versus-fastest without drawing on a whiteboard for an hour. That is most of the value: a small team moves faster when it argues in the same words.

> *Patterns don't design the system. They let you describe one fast enough to change your mind cheaply.*

Applied with judgement

Every pattern in Part II carried a skip-if, and that was the harder half of the book. The selection criterion ran underneath all of it: maximum production-readiness per unit of team capacity. A pattern a five-person team can't maintain is a liability the day after it merges, not an asset.

This is the over-engineering tax, and it is real money. Every pattern you adopt is one you carry forever. The honest default is boring: single-tenant before multitenancy, CRUD `Order` rows before an event-sourced stream, a monolith before microservices, a `tenant_id` and a Row-Level Security policy before a database per restaurant. You earn the heavier pattern by hitting the wall the simple one couldn't, and not a sprint sooner.

The deferring question is the whole discipline in one line. *What breaks if I don't add this yet?* If you can answer with a concrete failure you've actually seen, reach for the pattern. If the answer is "it might not scale someday," you've found a tax, not a requirement. A single restaurant on a single schema does not need a Saga; the day a real charge has to be refunded because a courier never showed, it does.

The code under all of it

The stances held for the whole book because they're the cheap ones to live with. SQL-first data access through a thin Data Gateway, the `OrderGateway` calling stored procedures behind a small interface, not a heavy ORM you can't reason about under load. Polly for the resilience pipeline wrapping the payment call. OpenTelemetry and Serilog for the things you have to see. A container as the portability boundary, state pushed outside it, the same image on GCP (AWS, Azure) without a rewrite.

```
var pipeline = new ResiliencePipelineBuilder()
    .AddRetry(new RetryStrategyOptions { BackoffType = DelayBackoffType.Exponential, UseJitter
    .AddCircuitBreaker(new CircuitBreakerStrategyOptions { FailureRatio = 0.5 })
    .AddTimeout(TimeSpan.FromSeconds(10))
    .Build();
```

None of that is exotic. It's the smallest set of choices that survives a year of production with nobody on call but you. The Order Service in Chapter 10 wired the lot together, and the surprising thing about it was how little surprised anyone. A customer places an order, the Outbox publishes `OrderPlaced`, the kitchen and courier workers pick it up, a payment Saga charges and pays out, and the customer reads live status from a projection. Each altitude doing one job, on the same `Order` you met in the first chapter.

What this leaves you with

Production-ready is a bar you clear, not a maximum you chase. The bar is finite and it is learnable. It never required a platform team standing behind you, whatever the industry likes to imply. A small team fluent in this vocabulary builds pretty much any modern, cloud-agnostic system, a food-delivery marketplace or whatever you actually ship, and then keeps it running without paging itself awake.

This is how we build internally. The foreword said the list stopped being private a while ago: it became the thing we hand a new engineer in their first week, the same `Order` climbing the same ladder, across the work we ship for clients and our own systems. There's a team that ships exactly this stack by default, because being fluent in the whole vocabulary is the day job. That's You-Source's Dev on Demand: a fractional engineer who has climbed every rung of this ladder before and knows which twenty per cent your system actually needs. If that's a shortcut you'd rather take than build the fluency yourself, the company is easy enough to find.

Either way, the bar is the same, and you now know where it sits.

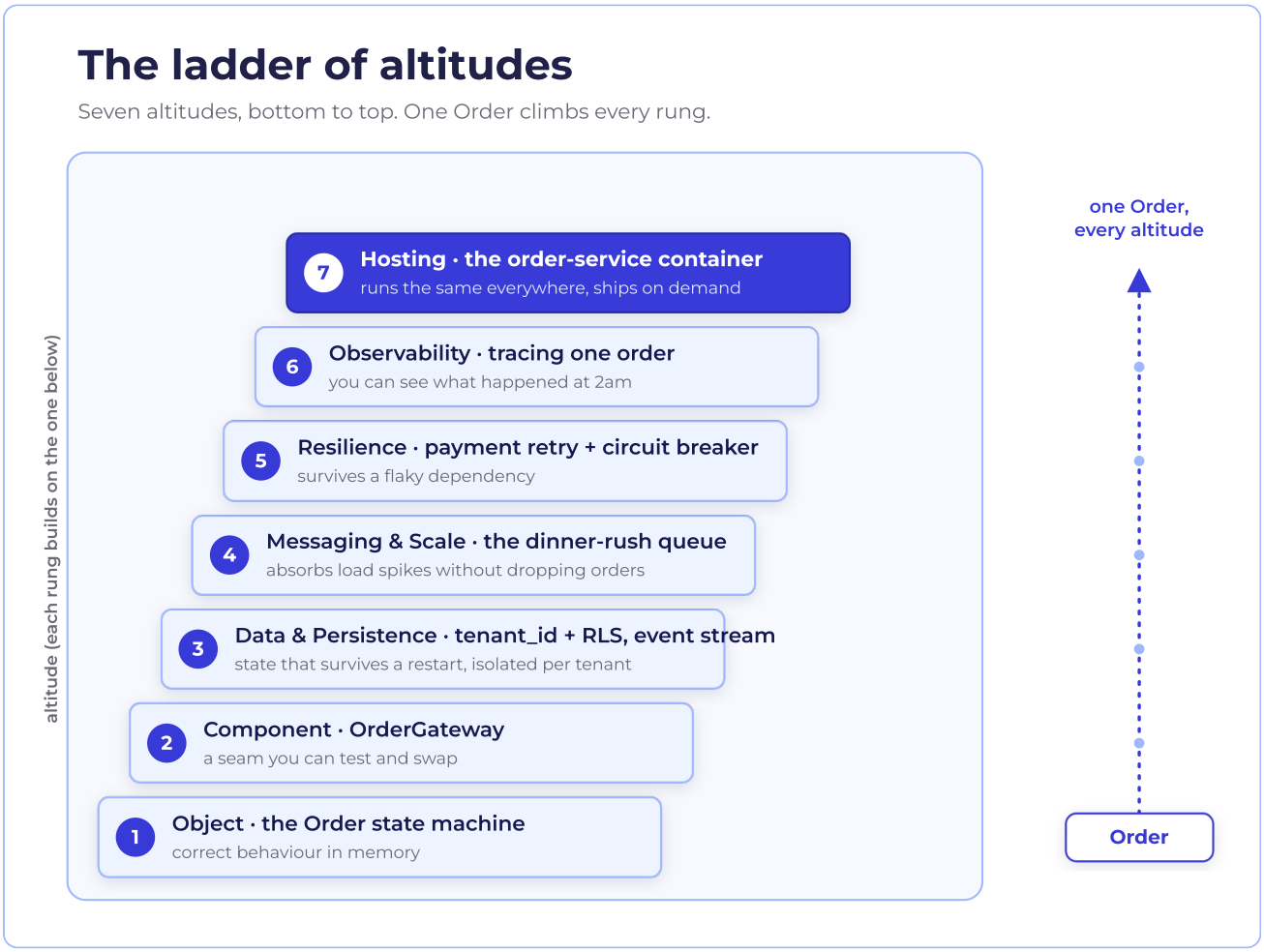
> *Maintain what you ship. The right twenty per cent.*

Appendix A — The Pattern Quick-Reference Card

Every pattern in the Pareto stack, grouped by altitude: what each one solves, its shape in a few words, and the skip-if that tells a small team to leave it on the shelf. The page to pin above your desk.

This is the whole ladder on one card. Each row is a pattern you met in Part II, compressed to the three things you need at a glance: the problem it answers, the shape it takes in code or config, and the honest signal to skip it. The skip-if column is the one most reference cards omit and the one a small team reads first. Every pattern you adopt is one you maintain forever, so read down the skip-if column before you reach for anything.

The attributions stay in the per-altitude chapters and in Appendix C. The chapters carry the worked snippets; this card carries the shape in a phrase. When two disagree, trust the chapter.



Altitude 1 — Object

The code-level vocabulary, mostly from the Gang of Four, with a value object (Money) and a null object reaching just past the catalogue. These organise behaviour inside a single boundary. The shapes are plain C#, no library required.

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Strategy	Swap an algorithm or behaviour at runtime	An interface with interchangeable implementations, chosen by the caller (`CourierMatchingStrategy`: nearest, fastest, cheapest)	Only one implementation will ever exist
Adapter	Make an incompatible interface fit yours	A wrapper that translates a third-party SDK to your own seam (the payment provider)	You control both sides of the interface
Decorator	Add behaviour without subclassing	A wrapper implementing the same interface, layering surge, promo or loyalty on a base price	DI interception already does it
Command	Encapsulate an action as an object	A request as data you can queue, retry, undo or audit (place, cancel, add-to-cart)	A direct method call is all you need
State	Make behaviour follow an explicit lifecycle	A state object per status, each owning its allowed transitions (the `Order` lifecycle, placed through delivered)	Two states, one transition: use a `bool`
Proxy	Control access to the real object	A stand-in for lazy loading, access control, caching or a remote call (the lazy-loaded menu image)	No access, laziness or remoting concern
Composite	Treat a tree and its parts through one interface	A recursive structure where a node and a leaf share a type (the menu tree: sections, items, modifier groups)	The data is flat, never nested
Money	Carry an amount and its currency as one value	An immutable value type owning cents, currency, and the one rounding rule; adding different currencies throws	A genuinely single-currency app that rounds in one place
Null Object	Kill scattered null checks for an optional collaborator	A do-nothing implementation of the interface (a silent `NullNotifier`), selected like any other	Absence that means something the caller must handle

Honorable mentions: Factory Method (DI absorbs most creation), Facade, Template Method, Chain of Responsibility.

Altitude 2 — Component (structuring one service)

How a single service stays testable and changeable. Dependency Injection is the spine; the rest keep the domain core away from I/O.

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Dependency Injection	Inject collaborators instead of newing them up	Constructor parameters resolved by `MS.Ext.DI` with explicit lifetimes	A one-file script
Data Gateway (+ stored procs)	Own a table's SQL in one thin object, no ORM magic	A focused class calling Dapper over stored procedures behind an interface	A genuine throwaway prototype
Ports & Adapters (Hexagonal)	Insulate the domain core from I/O	Domain logic depending on interfaces; adapters at the edge wire the DB and network	Tiny CRUD: plain layering is cheaper
Pipeline / Middleware	Compose cross-cutting concerns per request	An ordered chain handling auth, logging and validation before the handler	No cross-cutting concerns yet
Mediator (command/query split)	Decouple a request from its handler	One handler per use-case, dispatched through a mediator; thin controllers	A three-endpoint service
Anti-Corruption Layer	Keep a messy external model out of yours	A translation boundary that maps a third party's model to your domain	No external system, or one you fully control
Domain Events	Decouple in-process side effects from the core	An event raised on a change; handlers react (email, audit, projection)	One straightforward side effect: just call it
Specification	Reuse one business rule across query, validation and UI	A rule object with `IsSatisfiedBy`; `And` / `Or` / `Not` compose as a Composite ("open and in range")	The rule only ever runs as one SQL query: write the predicate

Honorable mentions: Result/ErrorOr error handling, plain Layered architecture. *(Worked over-engineering example for this altitude: heavy ORM / EF.)*

Altitude 3 — Data & Persistence

The database layer the catalogues under-serve. Start dense, escalate only when forced. The combo slot is the heaviest carrying cost in the book.

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Multitenancy (shared schema + RLS)	Isolate tenants in one database	One DB, a `tenant_id` column, Row-Level Security enforcing the boundary	A single-tenant product: build none of it
Event Sourcing + CQRS + Materialized View	Audit, replay and read-scale without locking the write model	Events as the write log, materialized views as the read model, CQRS the umbrella	Most CRUD apps: the heaviest skip-if in the book
Optimistic Concurrency	Detect conflicting writes without locking	A version or `rowversion` checked on update; the loser retries	Single-writer data
Evolutionary Schema Migrations	Change the schema safely and reproducibly	Versioned, forward-only scripts (DbUp-style, not EF migrations)	Never, but keep it lightweight
Sharding / Partitioning	Scale data past one box	Data split across stores or partitions by a key	Nowhere near a single DB's limits: a classic tax
Cache-Aside	Cut read latency and DB load	Load on miss, populate the cache, serve from it next time	Writes dominate, or staleness can't be tolerated and you can't invalidate cleanly
Soft Delete / Temporal	Keep recoverable data and a change history	A flag or system-versioned rows; reads exclude deleted	No recovery or audit value (and mind the per-read query cost)

Honorable mentions: lightweight Unit of Work (transaction boundary across gateways), Read Replicas, Connection Pooling.

Altitude 4 — Messaging & Scale

Put a broker between the parts that produce work and the parts that handle it. A queue absorbs spikes, lets you add workers when volume grows, and parks failed messages instead of losing them. Broker-neutral: GCP Pub/Sub (AWS SQS+SNS, Azure Service Bus).

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Queue-Based Load Levelling	Absorb a spiky producer	A queue between producer and a steady-rate consumer	Synchronous, low-volume work
Competing Consumers	Scale processing horizontally	Many workers pulling from one queue	Ordering matters more than throughput
Publish/Subscribe	Fan one event out to many consumers	A topic; independent subscribers add themselves without touching the producer	Exactly one consumer, forever
Outbox	Stop lost or double events between DB and broker	The event written in the same DB transaction, published after commit	No cross-system consistency need
Backpressure / Dead-Letter Queue	Never silently drop failed work	Signal upstream to slow; park unprocessable messages for inspection	In-memory, single-process work
Saga / Process Manager	Keep consistency across services with no 2PC	A multi-step distributed transaction with compensating actions	You don't yet have a distributed transaction: a heavy skip-if
Claim-Check	Keep messages small and fast	The large payload stored externally; only a token rides the bus	Payloads are already small

Honorable mentions: Idempotent Consumer (see Altitude 5), Event-Carried State Transfer, Priority Queue.

Altitude 5 — Resilience

Every network call is a coin-flip that sometimes lands wrong. These turn a dependency hiccup into a non-event. Library: Polly, without locking the concepts to it.

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Retry (backoff + jitter)	Ride out a transient failure	Re-issue with exponential backoff and jitter; pair with Idempotency	The op isn't safe to repeat and can't be made idempotent
Circuit Breaker	Stop hammering a failing dependency	Trip open after a failure threshold, fail fast, probe before closing	A single in-process call
Rate Limiting / Throttling	Cap request rate, in and out	A limiter that sheds or queues traffic above a ceiling	Trusted, low-volume internal traffic
Bulkhead	Stop one slow dependency draining everything	Isolated resource pools per dependency	One dependency, low concurrency
Timeout + Fallback	Never wait forever; degrade instead of erroring	A bounded wait with a sensible degraded answer on expiry	No sensible fallback: then fail fast and loud
Idempotency	Make retries and at-least-once delivery safe	The same request applied twice has one effect, keyed by an idempotency token	Naturally idempotent reads
Steady State	Stop a 3am failure from something filling up	Bound every growing resource: rotate logs, purge old data, cap caches and pools	Nothing in the process grows unbounded (rare: check first)

Honorable mentions: Graceful Degradation, Load Shedding, Failover/Redundancy.

Altitude 6 — Observability & Diagnostics

You can't operate what you can't see, and seeing is worthless without something that wakes someone. Vendor-neutral: OpenTelemetry plus Serilog.

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Health Endpoint Monitoring	Let an orchestrator restart or drain bad instances	Liveness and readiness endpoints it can probe	Nothing's orchestrating it
Structured Logging	Make "what happened at 2am" answerable	Machine-parseable logs with context (Serilog), not string soup	Never
Metrics (Four Golden Signals)	See degradation before customers do	Latency, traffic, errors, saturation: start with four, not fifty	Never
Distributed Tracing + Correlation IDs	Root-cause across a call graph in minutes	One request followed across services via a propagated trace context (OpenTelemetry)	A single process, no fan-out
Externalised Configuration	Run the same image in every environment	Config read from the environment, not baked into the image	Never
Alerting & SLOs	Page a human before the customer notices	Thresholds tied to objectives, not raw noise	No on-call or SLA yet (but wire it before you have users)
Audit Logging	Keep a defensible who-did-what trail	An immutable log of actions, references not PII, separate from diagnostics	No regulatory or security need, no sensitive actions

Honorable mentions: Log Aggregation, Synthetic Monitoring, the RED/USE methods.

Altitude 7 — Hosting (cloud-agnostic, container-first)

The container is the portability boundary; state lives outside it. The same image runs on GCP, AWS or Azure. 12-factor is the backbone.

PATTERN	WHAT IT SOLVES	THE SHAPE	SKIP-IF
Container as the Unit (12-factor)	Run the same artifact everywhere	One OCI image as the deployable, portable boundary	A managed-runtime function where a container adds nothing
Stateless + Externalised State	Scale out and restart freely	No state in process memory; instances are cattle, not pets	A genuinely single-instance tool
Sidecar / Ambassador	Add cross-cutting infra without app changes	A helper container (proxy, agent) co-deployed beside the app	The helper's job is a library call away
Scale-to-Zero (+ graceful shutdown)	Pay for use, lose no in-flight work	Idle instances drop to zero; SIGTERM drains work on reclaim	Latency-critical, always-warm workloads
Orchestrator-Agnostic Deploy	Avoid cloud lock-in	The same OCI image to Cloud Run, ECS, Container Apps or k8s	You've deliberately committed to one platform's primitives
Infrastructure as Code	Reproduce environments without a platform team	Version-controlled, reviewable infra definitions	A single hand-clicked environment you'll never rebuild (usually a false economy)
Blue-Green / Canary Deploy	Release with zero downtime and instant rollback	Traffic shifted to a parallel or partial slice, rolled back fast on trouble	A low-traffic internal tool where a few seconds of downtime is fine

Honorable mentions: Feature Flags, Gateway/Backend-for-Frontend, Secrets Management, Service Discovery.

> *One vocabulary, seven rungs. The skip-if column is the half of the card most teams need most.*

Next: Appendix B — The Skip List, for the patterns deliberately left off the ladder and the reason each is a tax a small team rarely needs.

Appendix B — The Skip List

The patterns this book leaves out on purpose, and the one honest line that explains why a small team rarely needs each. The quick-reference card tells you what to reach for; this is the other half of the judgement, the things you can walk past.

A curated set is defined as much by what it excludes as what it keeps. Every pattern below is real, useful, and earns its place in some system somewhere. The question is never whether a pattern is good. It's whether the production-readiness it buys you is worth the capacity it costs a team that has to maintain what it ships. For each one here, the answer is usually no until something specific forces your hand. That something is the skip-if, read in reverse.

> *A pattern you carry without a reason isn't architecture. It's debt with a fancier name.*

Singleton-as-global

The Gang of Four gave us Singleton (GoF, 1994) as a way to guarantee one instance. In practice most teams use it as a respectable-looking global variable, and a global variable is a global variable however you dress it. It hides dependencies and defeats your tests. Worse, it turns object lifetime into something nobody on the team can reason about.

You already have a better tool. A DI container manages a single instance for you when you register it, and does it without the static coupling. This is the Dependency Inversion Principle doing its job (Robert C. Martin, the "D" in SOLID, *Agile Software Development: Principles, Patterns, and Practices*): depend on an abstraction the container supplies, not on a static you reach for by name. Register the thing as a singleton lifetime and inject it.

```
// Not this: a static no test can replace.
public static class Clock { public static DateTime Now => DateTime.UtcNow; }

// This: one instance, injected, swappable in a test.
services.AddSingleton<IClock, SystemClock>();
```

The skip-if reads in reverse: if you find yourself writing `Whatever.Instance`, you wanted a DI lifetime.

Observer, Iterator, Builder

These three won so completely that C# absorbed them into the language. Observer is what `event` and `IObservable<T>` already do. Iterator is `IEnumerable<T>` and `yield return`. Builder is the object initialiser and, increasingly, the `with` expression on a record. The concepts are still correct, all three of them GoF. As hand-rolled patterns they're now redundant.

Writing them out by hand in 2026 reinvents a compiler feature you already paid for, and there's nothing disciplined about that. We teach what the language doesn't give you for free and skip what it does.

> *Don't hand-roll a pattern your compiler already ships.*

Heavy ORM / Entity Framework

This is the worked over-engineering example in the component chapter, so the verdict is short here. An ORM trades hand-written SQL for change-tracking magic, LINQ that compiles to queries you didn't write, and a migration history a junior can't read. The book's stance is SQL-first data access: a thin Data Gateway over stored procedures (Fowler, *PoEAA*), where the query you ship is the query you wrote.

Skip the ORM until you genuinely have a domain so large that mapping by hand is the bottleneck, and a team big enough to own the framework's behaviour when it surprises you. Most small teams never reach either.

Saga, before you have a distributed transaction

A Saga (Garcia-Molina & Salem, 1987; Richardson) coordinates a multi-step transaction across services using compensating actions, because you can't hold a single database lock across a network. It is the right answer to a real and painful problem. The trouble is that most systems billed as needing one have a transaction that still fits inside a single database.

If your work commits in one transaction against one store, you don't have a distributed transaction, and a Saga is pure overhead: orchestration code, compensation logic, and a state machine to debug at 3am. Reach for it the day a business operation truly spans two systems that can each fail on their own. Not before.

Full Event Sourcing, for CRUD

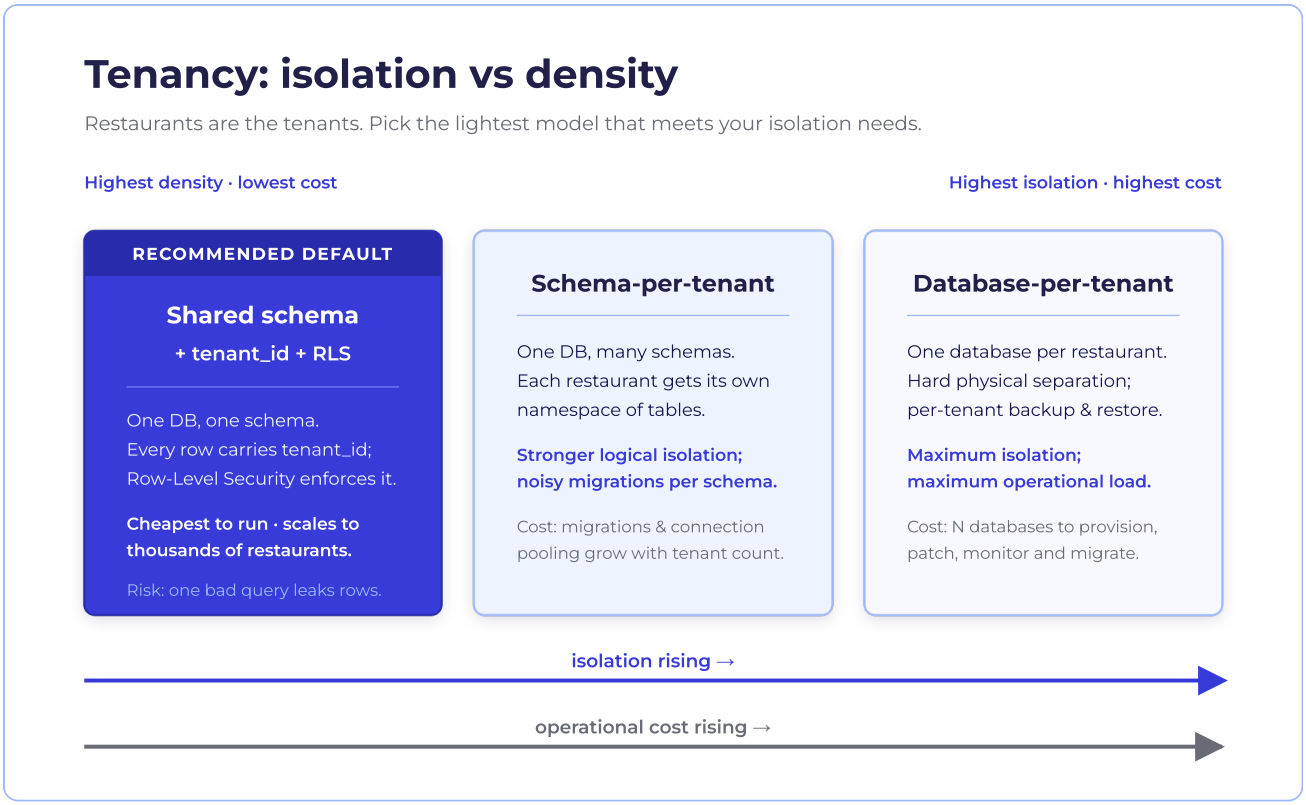
Event Sourcing plus CQRS plus Materialized View (Fowler's bliki; Event Sourcing popularised by Greg Young, CQRS by Greg Young and Udi Dahan) is the book's golden combo for read-scale, and its heaviest skip-if. The write model becomes an append-only log of events; the read model is projected from it. For the right workload, read-heavy, audit-hungry, with real contention on the write side, it's the pattern that pays for itself.

For an application that creates, reads, updates, and deletes rows, it's a tax with no return. You inherit event versioning, projection rebuilds, eventual consistency between write and read, and a mental model every new hire has to absorb before they can ship a form. A table and a few stored procedures do the same job and fit in your head. Most CRUD apps should stay CRUD.

> *Event Sourcing is a loan against future read-scale. Don't take it out to store a contact form.*

Database-per-tenant, before compliance forces it

The book's multitenancy default is the dense end of the spectrum: one database, one schema, a `tenant_id` column, and Row-Level Security so the database itself enforces isolation (PostgreSQL RLS). It's the cheapest tenancy that's still a real guarantee.



Database-per-tenant sits at the far end: maximum isolation, and a multiplied operational cost. Every migration, backup, connection pool, and monitoring dashboard now exists once per customer. You take that on when a contract or a regulator requires physical isolation, or when one tenant's load genuinely poisons the others. Escalate to schema-per-tenant first if you only need customisation. Walk the spectrum as far as you're forced, and no further.

Service mesh, for three services

A service mesh (Istio, Linkerd, and the like) gives you mTLS, traffic shaping, retries, and per-call telemetry across a fleet of services, applied uniformly without touching application code. At fleet scale, that uniformity is the whole point. At three services, it's a control plane, a sidecar per pod, and a new operational surface for problems you didn't have.

You can get retries and timeouts from a resilience library (Polly), tracing from OpenTelemetry, and TLS from your ingress, all without standing up a mesh. The honest threshold is dozens of services and a platform team to run the thing. If you're counting your services on one hand, a sidecar per pod is the Ambassador pattern doing one job, not a mesh.

Premature sharding

Sharding (Azure Cloud Design Patterns) splits one logical dataset across many physical stores to get past the limits of a single one. It is also one of the hardest things to undo once it's in, because every query and every join now has to know which shard its data lives on.

The trap is sharding for a scale you're forecasting rather than feeling. A single well-indexed Postgres instance, with a read replica when reads outpace it, carries most applications comfortably long past the point teams assume they've outgrown it. Shard when one node is measurably your ceiling and vertical scaling and replicas are exhausted, not when a slide deck predicts you'll get there.

Abstract Factory sprawl

Factory Method is not on this list. It earns its honorable-mention slot in the object chapter: it's the creational glue that picks the right `CourierMatchingStrategy` at runtime, nearest or fastest or cheapest, without the caller knowing which. Abstract Factory, its bigger sibling (GoF, 1994), builds whole families of related objects behind an interface. It's the right tool when you genuinely swap an entire family at once, a full set of platform-specific widgets, say.

Most of the time it shows up as speculative generality: a factory of factories standing in front of a problem that had exactly one implementation. The cost is layers of indirection a reader has to walk through to find the one concrete class that ever runs. Add the abstraction the day you have the second family. Until then, `new` is allowed.

> *Indirection is not a virtue. It's a bet that the future will need it, and most of those bets lose.*

Appendix C lists every source behind these calls, and behind the patterns we did keep.

Appendix C — Sources & Further Reading

Every pattern in this book traces to a canonical origin. This appendix names it, in the same shorthand the chapters use, and points you at the primary text or page so you can go deeper at any altitude.

The book leans on provenance, not statistics. Where a chapter cites "(Fowler, PoEAA)" or "(GoF)" or "(Nygard, *Release It!*)", this is where that shorthand resolves. Two authors run through the whole ladder, so they lead: read the Fowler and the Robert C. Martin sections first, then the rest, grouped by altitude in the order you met them. URLs are included where the canonical reference lives online.

> *Get every attribution right; keep every snippet idiomatic. A wrong origin loses an engineer reader on the spot.*

Martin Fowler — the house authority for the middle altitudes

Fowler is the primary source for the component, data, and messaging altitudes. Most of the food-delivery patterns trace back to one of the works below: the `OrderGateway` over stored procedures, the `OrderPlaced` domain event, the order event stream feeding read models, the optimistic check on a restaurant editing its own menu.

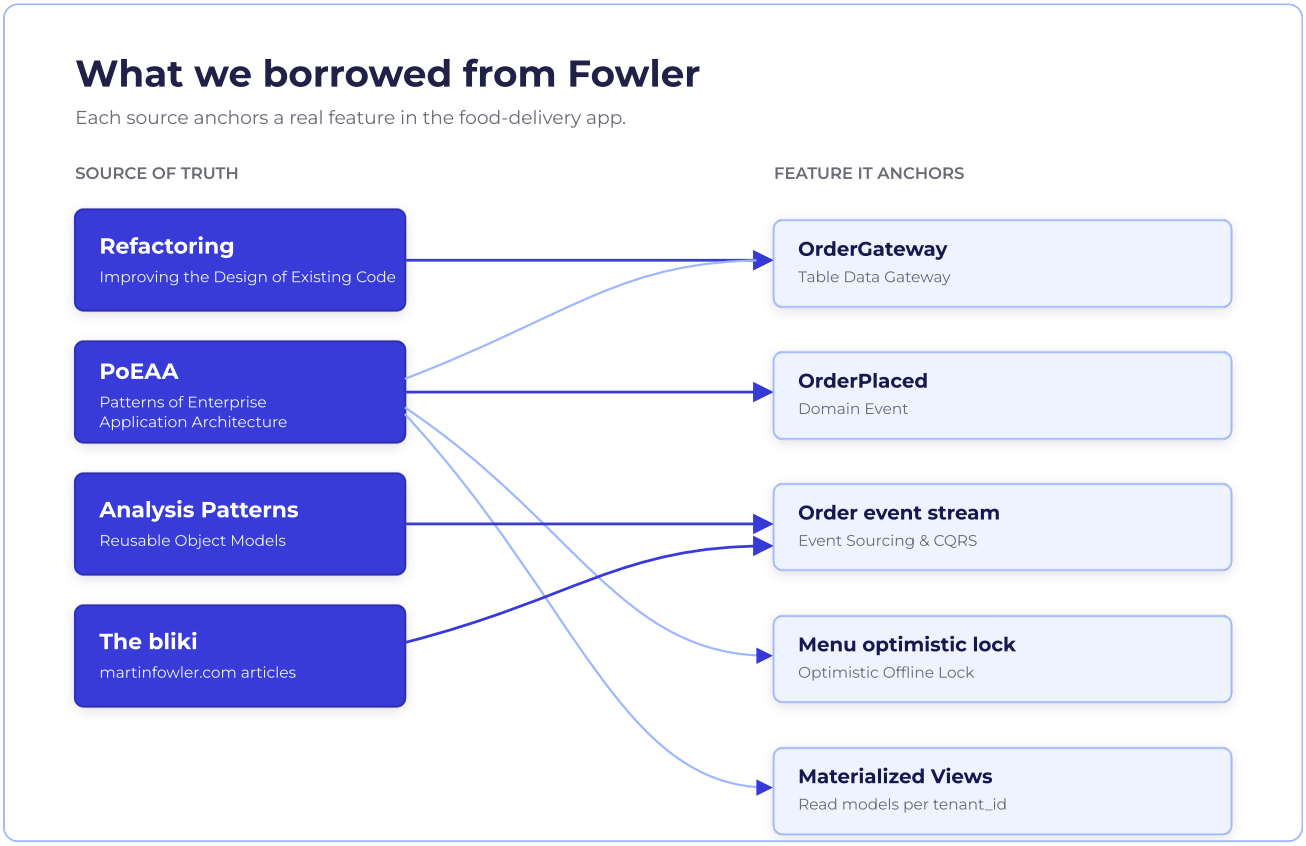
- **Fowler, Martin.** *Refactoring: Improving the Design of Existing Code.* 2nd ed., Addison-Wesley, 2018. The case for changing a working `Order` lifecycle in small, safe steps. Where the book argues you should still be able to change code in a year, this is the anchor.
- **Fowler, Martin.** *Patterns of Enterprise Application Architecture* (PoEAA). Addison-Wesley, 2002. The enterprise-application catalogue. Online catalogue: https://martinfowler.com/eaaCatalog/
- **Fowler, Martin.** *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997. The earlier modelling work behind the temporal and accountability patterns the data altitude borrows from.
- **Fowler, Martin.** *bliki* — an ongoing wiki/blog of pattern and architecture entries. https://martinfowler.com/ The "(Fowler bliki)" citations throughout resolve to the

individual entries below.

The specific patterns this book draws from Fowler, with the food-delivery feature each anchors:

- **Data Gateway (Table Data Gateway / Row Data Gateway)** — PoEAA. https://martinfowler.com/eaaCatalog/tableDataGateway.html The anchor for the book's SQL-first, anti-ORM stance: `OrderGateway` and `MenuGateway` are thin objects owning a table's SQL and stored procedures. Contrast with Data Mapper (https://martinfowler.com/eaaCatalog/dataMapper.html), which is what an ORM like EF implements.
- **Unit of Work** — PoEAA. https://martinfowler.com/eaaCatalog/unitOfWork.html Used here only as the transaction boundary that places an order and writes its outbox row in one commit, never as an ORM.
- **Optimistic Offline Lock (Optimistic Concurrency)** — PoEAA. https://martinfowler.com/eaaCatalog/optimisticOfflineLock.html Rowversion / `xmin` / ETag; the check that stops two managers editing the same restaurant's menu from clobbering each other.
- **Materialized View** — Azure Cloud Design Patterns (the canonical catalogue entry); Fowler references the concept on the bliki. https://learn.microsoft.com/azure/architecture/patterns/materialized-view The read model in the golden combo: courier task-list, customer order-history, restaurant live-board.
- **CQRS** — Fowler bliki. https://martinfowler.com/bliki/CQRS.html The umbrella separating the order write model from those read views. The book's heaviest skip-if.
- **Event Sourcing** — Fowler bliki. https://martinfowler.com/eaaDev/EventSourcing.html Popularised by Greg Young. The order as an append-only event stream.
- **Domain Event** — Fowler bliki. https://martinfowler.com/eaaDev/DomainEvent.html `OrderPlaced` notifying kitchen, stock, and confirmation. In C#, often dispatched via MediatR `INotification`.
- **Circuit Breaker** — Fowler bliki. https://martinfowler.com/bliki/CircuitBreaker.html Stop hammering a failing payment provider; covered in depth in *Release It!* below.
- **Evolutionary Database Design** — Fowler & Sadalage. https://martinfowler.com/articles/evodb.html Versioned, forward-only migrations for the orders schema (DbUp / Flyway-style; not EF migrations).
- **Infrastructure as Code** — Fowler bliki. https://martinfowler.com/bliki/InfrastructureAsCode.html The whole food-delivery stack as reproducible, reviewable, cloud-portable environments.

- **Feature Toggles** — Fowler. https://martinfowler.com/articles/feature-toggles.html Ship the new pricing engine dark, then turn it on per tenant.
- **Backend-for-Frontend (BFF)** — Fowler bliki (after Sam Newman). https://martinfowler.com/articles/gateway-pattern.html A separate API surface for the customer app and the courier app.
- **Inversion of Control & Dependency Injection** — Fowler, *"Inversion of Control Containers and the Dependency Injection pattern"* (2004). https://martinfowler.com/articles/injection.html Pairs with the "D" in SOLID below; in C#, `Microsoft.Extensions.DependencyInjection`.
- **Event-Carried State Transfer** — Fowler. https://martinfowler.com/articles/201701-event-driven.html An honorable mention in the messaging altitude.



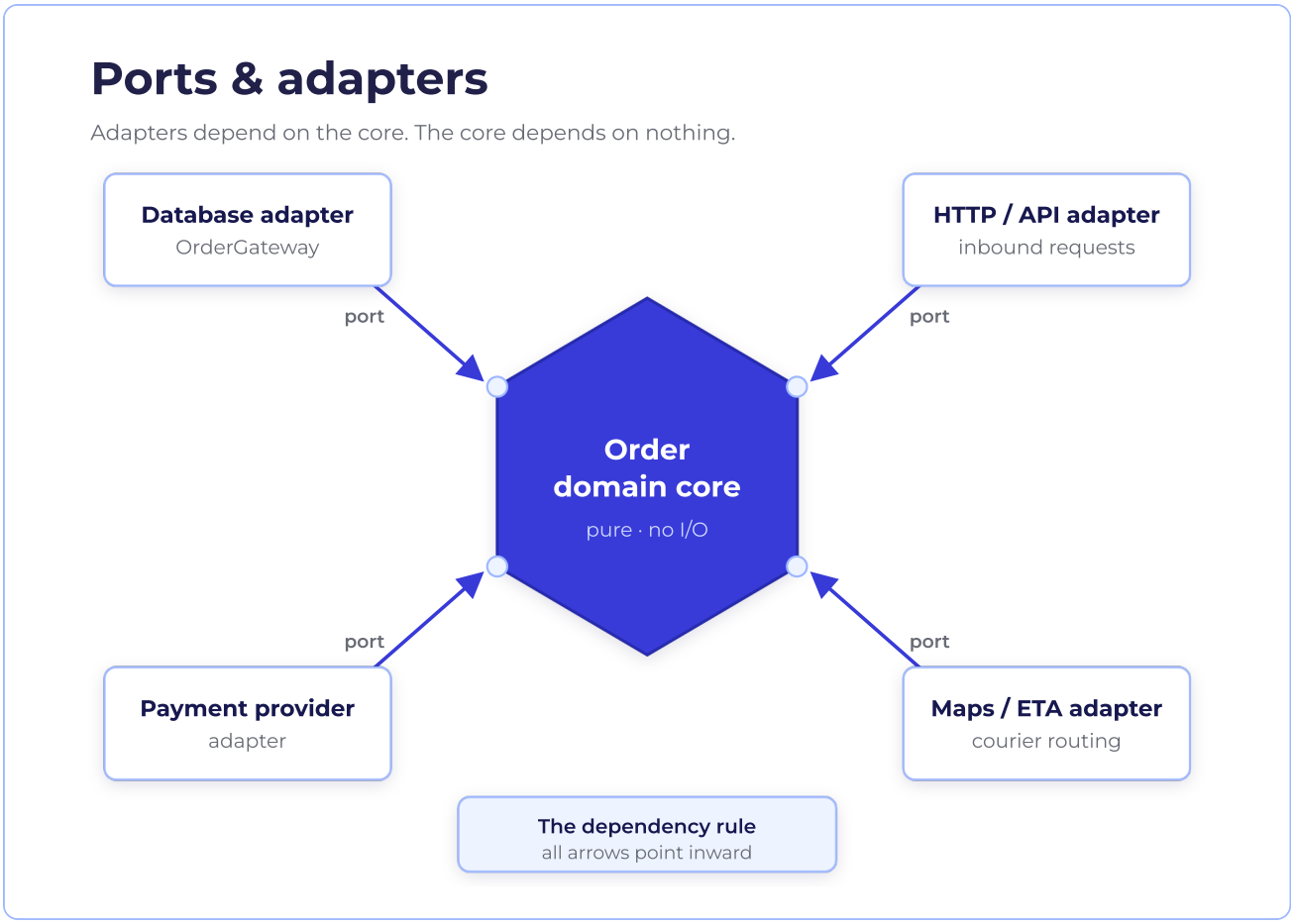
Robert C. Martin (Uncle Bob) — the house authority for structure

Where Fowler supplies the catalogue, Robert C. Martin supplies the principles that decide how the food-delivery service is wired together: which way the dependencies point, why

`CourierMatchingStrategy` is an interface and not a `switch`, and why the order domain never imports the payment SDK directly.

- **Martin, Robert C.** *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008. The naming and function-size discipline behind every snippet in the book. https://blog.cleancoder.com
- **Martin, Robert C.** *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall, 2017. The dependency-rule lineage behind the Ports & Adapters form the book teaches: the order domain at the centre, payment and courier providers at the edges.
- **Martin, Robert C.** *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The original full statement of the SOLID principles.
- **The SOLID principles** — Robert C. Martin.

- **S — Single Responsibility.** Why `OrderGateway` owns SQL and nothing else. - **O — Open/Closed.** Why a new courier-matching rule is a new `CourierMatchingStrategy`, not an edit to the matcher. - **L — Liskov Substitution.** Every payment `Adapter` honours the same contract. - **I — Interface Segregation.** The courier app's BFF does not depend on the restaurant's menu-edit methods. - **D — Dependency Inversion.** The DI anchor: the order service depends on an `IPaymentGateway` abstraction, not on a concrete provider. Pairs with Fowler's DI article above.

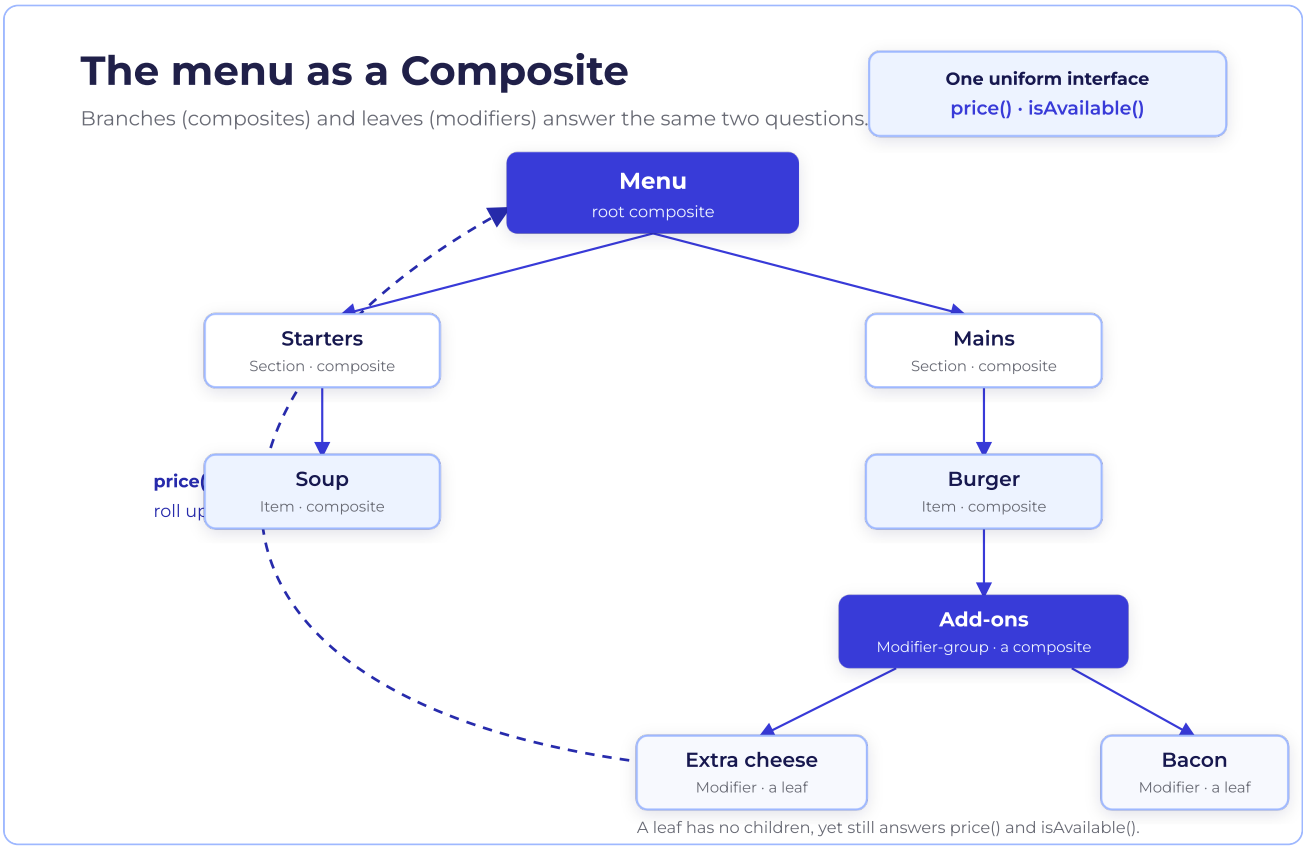


Altitude 1 — Object

Most of the patterns at this altitude come from a single source: **Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994. ISBN 0-201-63361-2. The "Gang of Four" (GoF). The original object-level catalogue of 23 patterns. Two more reach just past it, Fowler's Money and Woolf's Null Object, each cited below.

- **Strategy, Adapter, Decorator, Command, State, Proxy, Composite** — the curated seven, all from the original GoF catalogue. In the food-delivery app: the order lifecycle is **State**; courier-matching is **Strategy**; the payment provider is an **Adapter**; surge/promo/loyalty pricing stacks as **Decorator**; place/cancel/add-to-cart are **Command**; lazy-loaded menu images are a **Proxy**; and the **menu** (sections → items → modifier-groups → modifiers) is the **Composite**.

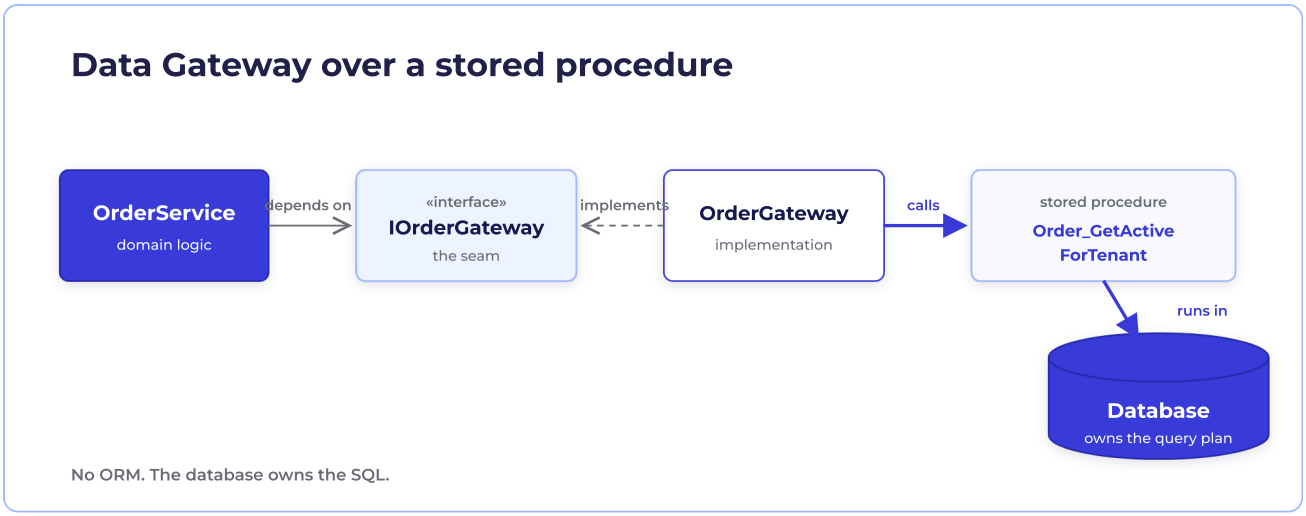
- **Factory Method, Facade, Template Method, Chain of Responsibility** — honorable mentions, also GoF.
- **Money** — Fowler, *Patterns of Enterprise Application Architecture* (PoEAA), 2002; the Money pattern, a Value Object. https://martinfowler.com/eaaCatalog/money.html Prices, surge, promo, and the payment charge all carry an amount and its currency as one immutable value, with rounding owned in a single place.
- **Null Object** — Bobby Woolf, *"Null Object,"* in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998; see also Fowler, *Refactoring* (Introduce Special Case). A degenerate Strategy whose algorithm is "do nothing": the silent `NullNotifier` handed to a customer who opted out of notifications.
- **Singleton** — GoF. Taught only as the over-engineering skip (global state; prefer DI lifetimes). The common critique is Miško Hevery, *"Singletons are Pathological Liars,"* http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/
- **The language-absorbed patterns** — Observer (now C# `event` / `IObservable<T>`, or a WebSocket pushing live order updates to the customer), Iterator (now `IEnumerable<T>` / `yield`), and Builder (now object/collection initialisers and `with` expressions) are GoF patterns the C# language absorbed. We don't teach what the compiler gives you.



Altitude 2 — Component

Primary sources: **Fowler (PoEAA + bliki)** and **Robert C. Martin (SOLID)**, both listed in full above.

- **Dependency Injection / Inversion of Control** — Fowler, *"...the Dependency Injection pattern"* (2004). https://martinfowler.com/articles/injection.html The "D" in SOLID (Robert C. Martin). The order service receives an `IPaymentGateway`, never constructs one. In C#: `Microsoft.Extensions.DependencyInjection`.
- **Data Gateway (Table Data Gateway / Row Data Gateway)** — Fowler, PoEAA. https://martinfowler.com/eaaCatalog/tableDataGateway.html `OrderGateway` and `MenuGateway` over stored procedures: the anchor for the book's SQL-first, anti-ORM stance. Contrast Data Mapper (https://martinfowler.com/eaaCatalog/dataMapper.html), which is what an ORM like EF implements.
- **Ports & Adapters (Hexagonal Architecture)** — Alistair Cockburn, 2005. https://alistair.cockburn.us/hexagonal-architecture/ Cousins: Robert C. Martin's Clean Architecture and Onion; we teach the minimal form, order domain at the centre.
- **Pipeline / Middleware** — Chain of Responsibility (GoF) at the object level; concretely, ASP.NET Core middleware that resolves `tenant_id` on every request. https://learn.microsoft.com/aspnet/core/fundamentals/middleware/
- **Mediator** — GoF Mediator at the object level; at the application level, the **MediatR** library by Jimmy Bogard. https://github.com/jbogard/MediatR A `PlaceOrder` request routed to its handler. The two are distinct; cite accordingly.
- **Anti-Corruption Layer** — Evans, *Domain-Driven Design* (2003). Summary: https://learn.microsoft.com/azure/architecture/patterns/anti-corruption-layer The higher-altitude cousin of Adapter; here it insulates the order model from a legacy restaurant POS.
- **Domain Events** — Evans (DDD); Fowler bliki. https://martinfowler.com/eaaDev/DomainEvent.html `OrderPlaced` fans out to kitchen, stock, and confirmation; in C#, often dispatched via MediatR `INotification`.
- **Specification** — Evans, *Domain-Driven Design* (2003); Evans & Fowler, *"Specifications"* (2002). https://www.martinfowler.com/apsupp/spec.pdf A composable business rule (`IsSatisfiedBy`) whose `And` / `Or` / `Not` combinators are a Composite; here "restaurant open and within range," reused across the search query, the order validation, and the UI.
- **Honorable mentions** — Result / ErrorOr functional error handling; plain Layered architecture.



Altitude 3 — Data & Persistence

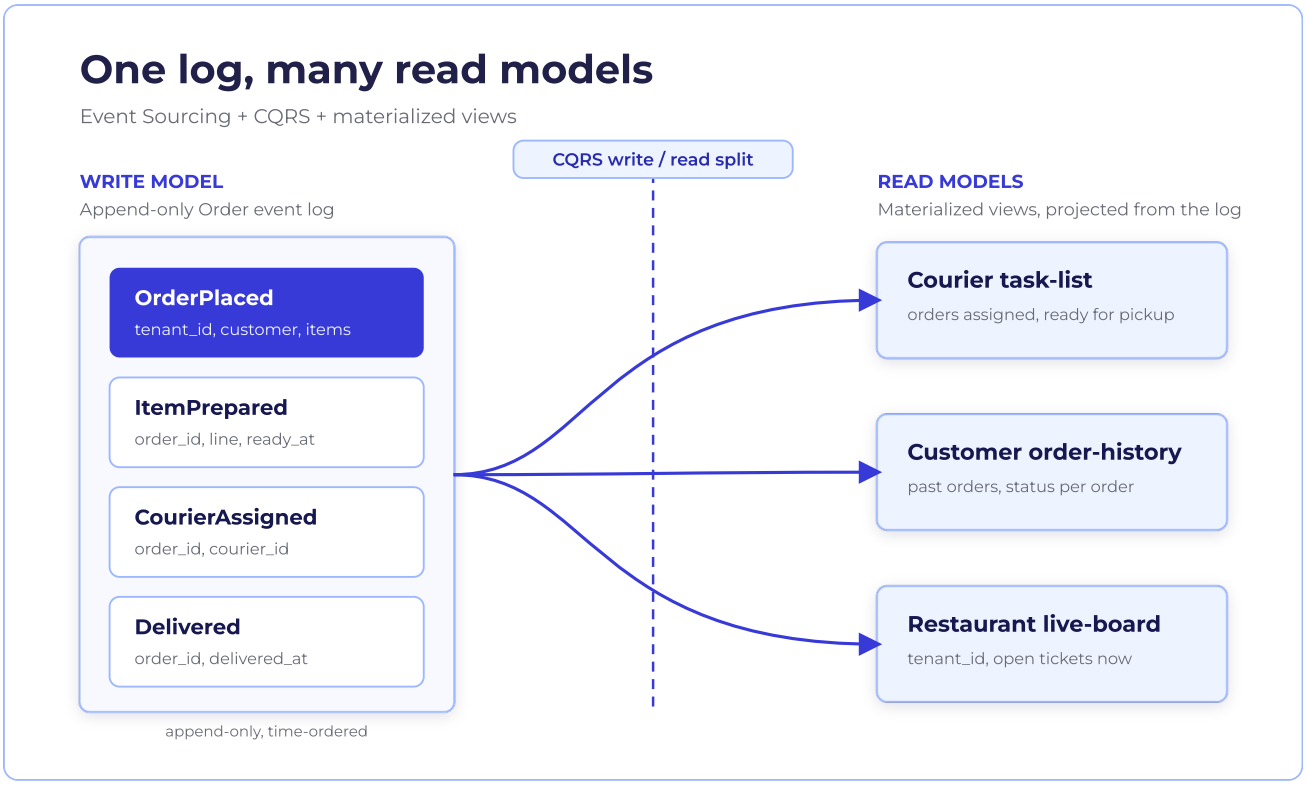
Primary sources: **Fowler (PoEAA + bliki)** for the read/write-split combo and concurrency; the cloud catalogue for the rest.

- **Multitenancy** — recommended default: shared schema + tenant discriminator (`tenant_id` , where the restaurant/brand is the tenant) + Row-Level Security.

- Microsoft, *Multitenant SaaS database tenancy patterns.* https://learn.microsoft.com/azure/azure-sql/database/saas-tenancy-app-design-patterns The database-per-tenant / schema-per-tenant / shared-schema spectrum. - PostgreSQL, *Row Security Policies (RLS).* https://www.postgresql.org/docs/current/ddl-rowsecurity.html The mechanism that turns shared-schema isolation by `tenant_id` into a database guarantee.

- **Event Sourcing** — Fowler bliki. https://martinfowler.com/eaaDev/EventSourcing.html Popularised by Greg Young. The order is an append-only event stream.
- **CQRS** — Greg Young and Udi Dahan; Fowler bliki. https://martinfowler.com/bliki/CQRS.html The umbrella separating the order write model from its read views. The book's heaviest skip-if; pair the power with the cost caveat.
- **Materialized View** — Azure Cloud Design Patterns. https://learn.microsoft.com/azure/architecture/patterns/materialized-view The read models in the golden combo: courier task-list, customer order-history, restaurant live-board.

- **Optimistic Concurrency (Optimistic Offline Lock)** — Fowler, PoEAA. https://martinfowler.com/eaaCatalog/optimisticOfflineLock.html Rowversion / `xmin` / ETag; stops two menu edits from clobbering each other.
- **Evolutionary Database Design / migrations** — Fowler & Sadalage, *"Evolutionary Database Design."* https://martinfowler.com/articles/evodb.html Versioned, forward-only migrations for the orders schema (DbUp / Flyway-style; not EF migrations).
- **Sharding / Partitioning** — Azure Cloud Design Patterns. https://learn.microsoft.com/azure/architecture/patterns/sharding Orders sharded by city.
- **Cache-Aside** — Azure Cloud Design Patterns. https://learn.microsoft.com/azure/architecture/patterns/cache-aside Load-on-miss for the menu; pair with an invalidation strategy.
- **Soft Delete / Temporal** — Fowler, temporal patterns (https://martinfowler.com/eaaDev/timeNarrative.html); SQL:2011 system-versioned temporal tables (https://learn.microsoft.com/sql/relational-databases/tables/temporal-tables). Menu-price history you can recover and audit.
- **Honorable mentions** — lightweight Unit of Work (Fowler, PoEAA, https://martinfowler.com/eaaCatalog/unitOfWork.html — used here only as a transaction boundary, not an ORM); Read Replicas; Connection Pooling.

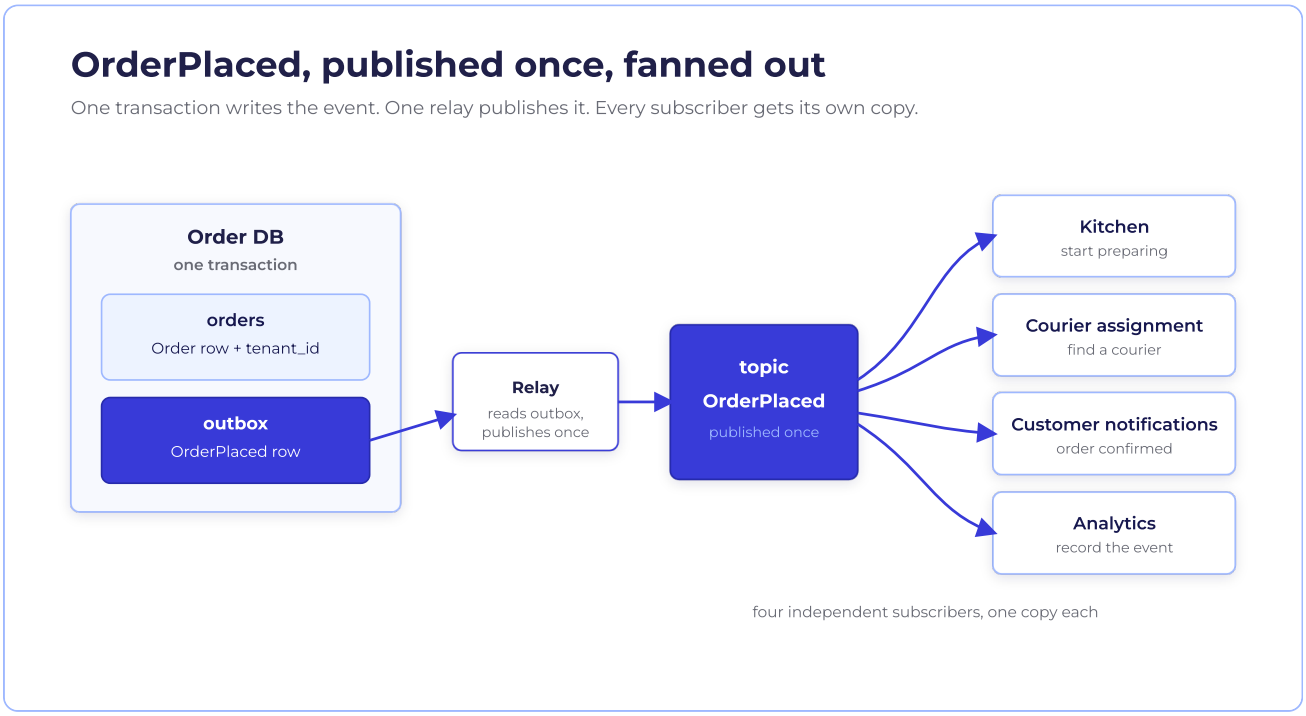


Altitude 4 — Messaging & Scale

Primary sources: **Azure Cloud Design Patterns** (https://learn.microsoft.com/azure/architecture/patterns/) and **Hohpe, Gregor; Woolf, Bobby.** *Enterprise Integration Patterns* (EIP). Addison-Wesley, 2003. https://www.enterpriseintegrationpatterns.com/

- **Queue-Based Load Levelling** — Azure Cloud Design Patterns. https://learn.microsoft.com/azure/architecture/patterns/queue-based-load-leveling Absorb the lunch/dinner rush.
- **Competing Consumers** — Azure Cloud Design Patterns (https://learn.microsoft.com/azure/architecture/patterns/competing-consumers); EIP. A pool of workers draining the courier-assignment queue.
- **Publish/Subscribe** — Azure Cloud Design Patterns (https://learn.microsoft.com/azure/architecture/patterns/publisher-subscriber); EIP Publish-Subscribe Channel. `OrderPlaced` fans out to kitchen, courier, customer, and analytics.
- **Transactional Outbox** — Chris Richardson, microservices.io. https://microservices.io/patterns/data/transactional-outbox.html The reliable bridge between the order's database transaction and the broker.
- **Dead-Letter Queue / Backpressure** — EIP Dead Letter Channel; Reactive Streams backpressure (https://www.reactive-streams.org/). Where an unprocessable order event lands.
- **Saga / Process Manager** — Hector Garcia-Molina & Kenneth Salem, *"Sagas,"* ACM SIGMOD, 1987 (https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf); Chris Richardson, microservices.io (https://microservices.io/patterns/data/saga.html). Order fulfilment as charge → assign courier → confirm with restaurant, with compensations (refund) on failure. Heavy skip-if.
- **Claim-Check** — EIP (https://www.enterpriseintegrationpatterns.com/patterns/messaging/StoreInLibrary.html); Azure Cloud Design Patterns (https://learn.microsoft.com/azure/architecture/patterns/claim-check). Store the large receipt, pass a token.
- **Honorable mentions** — Idempotent Consumer (see Altitude 5); Event-Carried State Transfer (Fowler, https://martinfowler.com/articles/201701-event-driven.html); Priority Queue (Azure Cloud Design Patterns).

- **Brokers (neutral)** — GCP Pub/Sub · AWS SQS + SNS · Azure Service Bus.



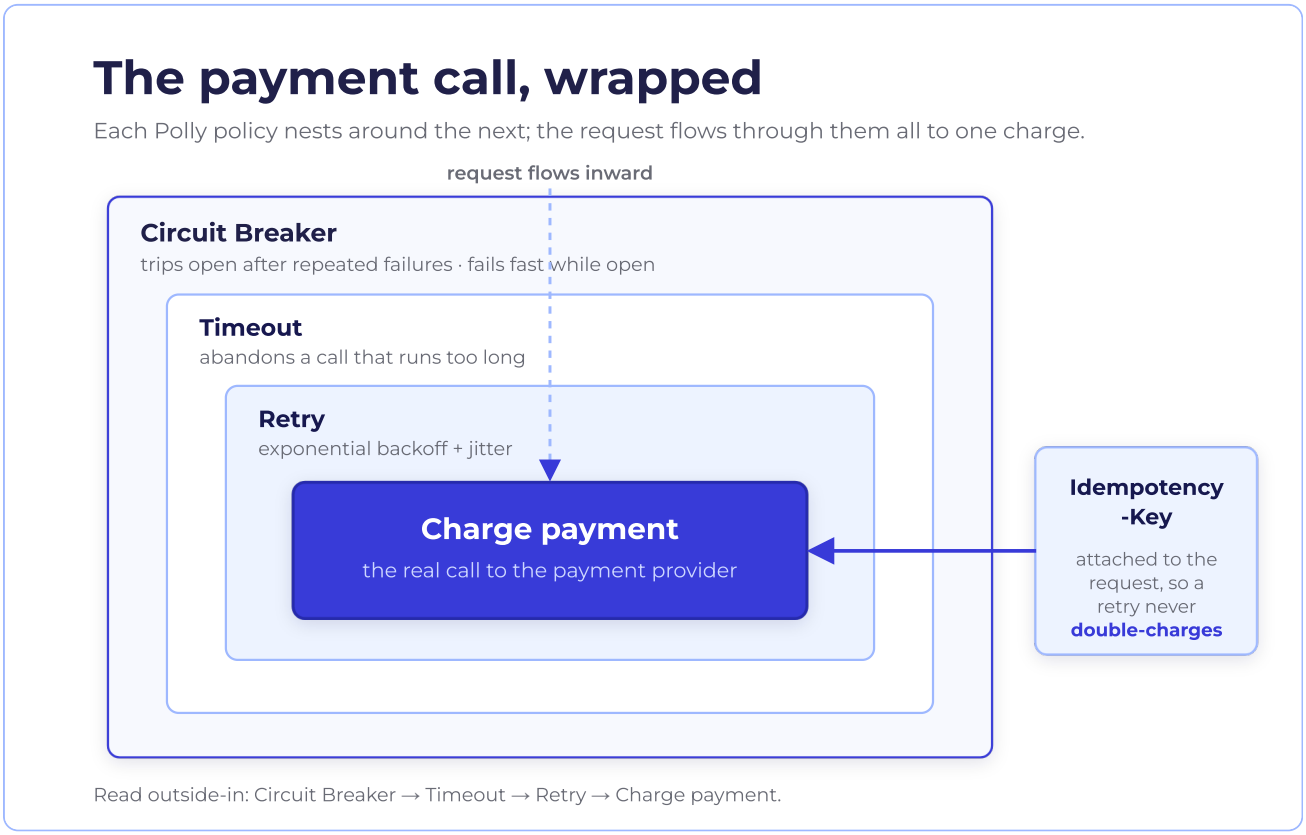
Altitude 5 — Resilience

Primary source: **Nygard, Michael T.** *Release It! Design and Deploy Production-Ready Software.* 2nd ed., Pragmatic Bookshelf, 2018. Library: **Polly** (https://www.pollydocs.org/).

- **Circuit Breaker** — Nygard, *Release It!*; Fowler bliki. https://martinfowler.com/bliki/CircuitBreaker.html Stop calling a failing payment provider.
- **Bulkhead** — Nygard, *Release It!* Isolate surge-pricing calls from order placement so one can't sink the other.
- **Timeout** — Nygard, *Release It!* Bound the wait on the payment gateway.
- **Steady State** — Nygard, *Release It!* Bound every growing resource: purge old courier GPS pings, cap the courier-location cache, rotate logs. Honorable mentions here: Graceful Degradation, Load Shedding, Failover / Redundancy.
- **Retry with exponential backoff + jitter** — Marc Brooker, AWS Architecture Blog, *"Exponential Backoff And Jitter."* https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/ Retry a flaky payment authorisation without stampeding.

- **Rate Limiting / Throttling** — Azure Cloud Design Patterns (https://learn.microsoft.com/azure/architecture/patterns/throttling); .NET rate limiting (`System.Threading.RateLimiting`). Cap the public order API.
- **Idempotency** — idempotency keys, Stripe API documentation (https://stripe.com/docs/api/idempotent_requests; illustrative). The property that makes a retried order placement safe: never double-charge, never double-place.

> *The compounding-failure illustration (95% per-step success over N steps = 0.95^N, roughly 60% at ten steps and 36% at twenty) is arithmetic, not a measured benchmark. Treat it as a worked example, never as a cited statistic.*

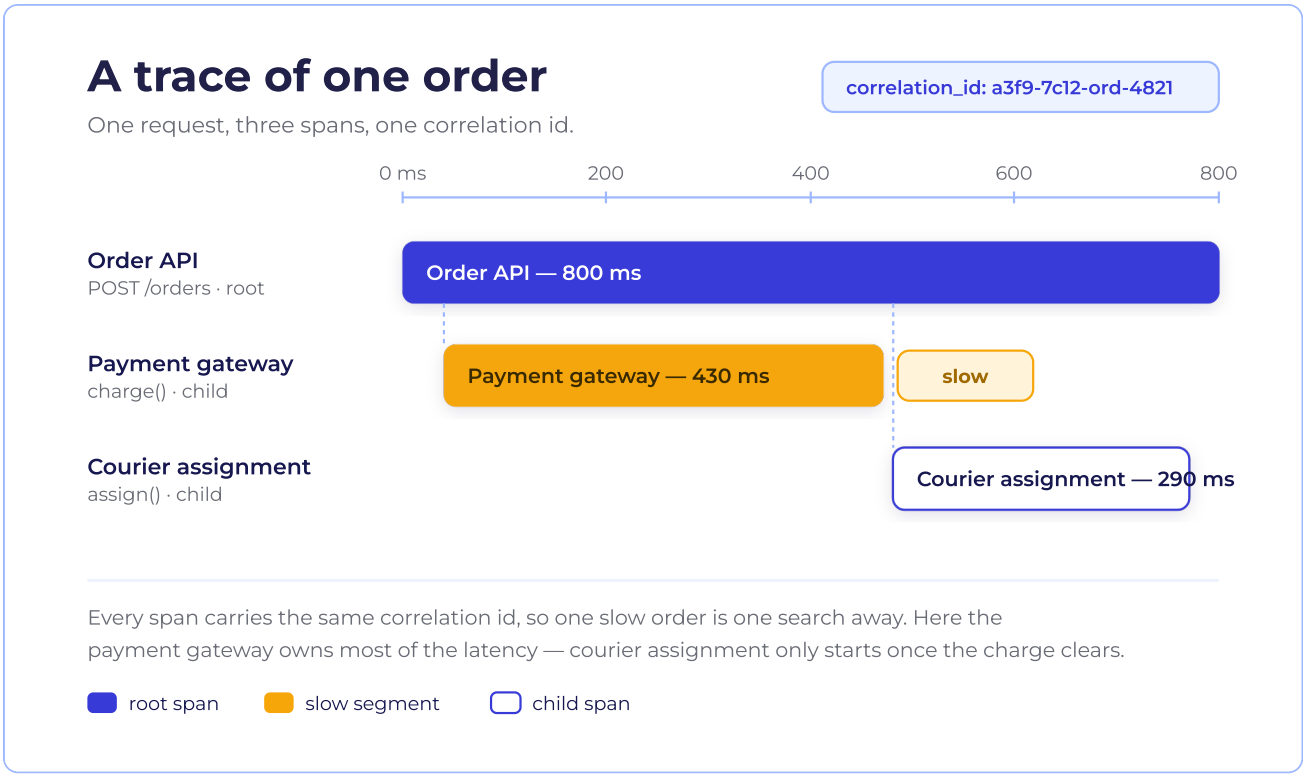


Altitude 6 — Observability & Diagnostics

- **The Four Golden Signals** (latency, traffic, errors, saturation) — Google, *Site Reliability Engineering* (O'Reilly, 2016), Ch. 6, *Monitoring Distributed Systems.* https://sre.google/sre-

book/monitoring-distributed-systems/ Watched on the order service at the dinner rush. Complements: the RED method (Tom Wilkie) and the USE method (Brendan Gregg, https://www.brendangregg.com/usemethod.html).

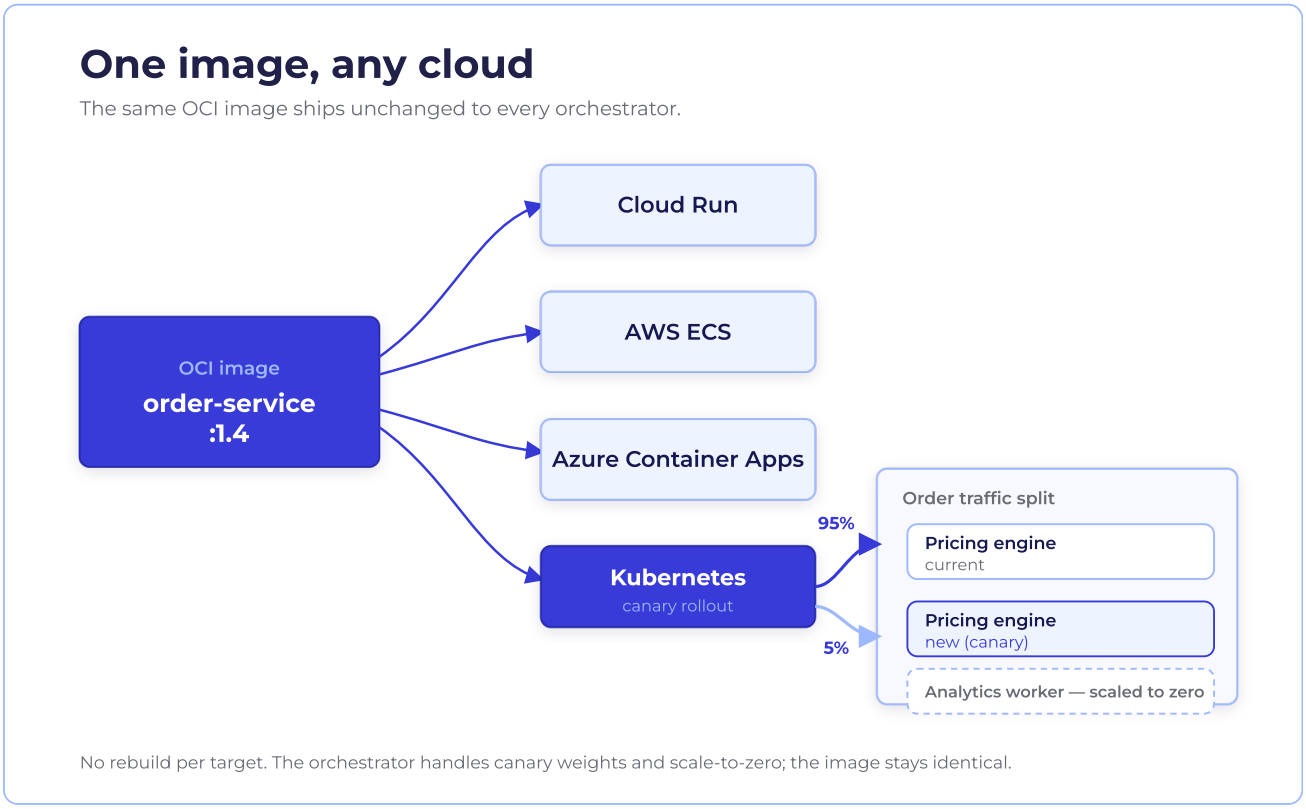
- **Health Endpoint Monitoring** — Azure Cloud Design Patterns (https://learn.microsoft.com/azure/architecture/patterns/health-endpoint-monitoring); ASP.NET Core health checks. Liveness and readiness for the order service's orchestrator.
- **Distributed Tracing, Metrics, Logs (the three pillars) + Correlation IDs — OpenTelemetry** (CNCF). https://opentelemetry.io/docs/ Trace one order across order → payment → courier. Vendor-neutral; export to Cloud Trace / Monitoring (X-Ray / CloudWatch · Azure Monitor).
- **Structured Logging** — Serilog. https://serilog.net/ Logs keyed on `order_id` and `tenant_id`.
- **Externalised Configuration** — *The Twelve-Factor App*, "III. Config." https://12factor.net/config Surge thresholds as config, not code. Overlaps Altitude 7; taught here as the diagnostics-friendly habit.
- **Alerting & SLOs** — Google SRE, *Service Level Objectives* (https://sre.google/sre-book/service-level-objectives/) and *Alerting on SLOs* (https://sre.google/workbook/alerting-on-slos/). "99% of orders confirmed within 10s": page on objectives, not noise.
- **Audit Logging** — OWASP Logging Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html An immutable who-did-what trail for refunds and cancellations: references, not PII.
- **Honorable mentions** — Log Aggregation; Synthetic Monitoring; the RED and USE methods.



Altitude 7 — Hosting

- **The Twelve-Factor App** — Adam Wiggins / Heroku, 2011. https://12factor.net/ The backbone: stateless processes, externalised config, disposability, port binding. The order service runs stateless, state in DB and queue.
- **Container as the unit** — OCI Image Specification (https://opencontainers.org/); Docker. One image for the order service.
- **Sidecar / Ambassador** — Azure Cloud Design Patterns (https://learn.microsoft.com/azure/architecture/patterns/sidecar and .../ambassador); Kubernetes pod sidecars. Telemetry shipped by a sidecar.
- **Scale-to-Zero** — Knative / Cloud Run (https://cloud.google.com/run/docs); AWS App Runner · Azure Container Apps. The analytics/report worker scales to zero off-peak. Note the cold-start tradeoff.
- **Graceful Shutdown (SIGTERM)** — Kubernetes pod lifecycle and container lifecycle hooks. https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/ Finish in-flight orders before exit.

- **Orchestrator-Agnostic Deploy** — the same OCI image to Cloud Run / ECS / Container Apps / Kubernetes.
- **Infrastructure as Code** — Terraform / Pulumi / Bicep; Fowler bliki. https://martinfowler.com/bliki/InfrastructureAsCode.html The whole food-delivery stack as reproducible, reviewable, cloud-portable environments.
- **Blue-Green Deployment** — Fowler bliki. https://martinfowler.com/bliki/BlueGreenDeployment.html
- **Canary Release** — Danilo Sato, Fowler bliki. https://martinfowler.com/bliki/CanaryRelease.html Route 5% of orders through a new pricing engine first, with an instant rollback path.
- **Honorable mentions** — Feature Flags / Feature Toggles (Fowler, https://martinfowler.com/articles/feature-toggles.html); Gateway / Backend-for-Frontend (Sam Newman; Fowler, https://martinfowler.com/articles/gateway-pattern.html) — a BFF each for the customer and courier apps; Secrets Management (GCP Secret Manager · HashiCorp Vault); Service Discovery.



Other foundational and framing texts

- **Evans, Eric.** *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003. The source for Anti-Corruption Layer (over the legacy restaurant POS), Domain Events, and Specification.
- **Microsoft. Azure Cloud Design Patterns.** A vendor-published but broadly applicable catalogue of cloud patterns. https://learn.microsoft.com/azure/architecture/patterns/ Cited as "(Azure Cloud Design Patterns)" throughout; individual pages are listed by altitude above.
- **Beyer, Betsy; Jones, Chris; Petoff, Jennifer; Murphy, Niall Richard (eds.).** *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly, 2016. Free online: https://sre.google/sre-book/table-of-contents/ The source for the four golden signals and SLO-based alerting.
- **OpenTelemetry** (CNCF). https://opentelemetry.io/docs/ The vendor-neutral standard behind the observability altitude.

Framing & delivery metrics

Used to frame what good delivery looks like, not as quoted benchmarks.

- **DORA — the four key metrics** (deployment frequency, lead time for changes, change failure rate, time to restore service). https://dora.dev/guides/dora-metrics-four-keys/ A framework, not a multiplier; don't quote elite-versus-low figures without the report.
- **Standish Group, CHAOS 2015** — small projects succeed far more often than grand ones, which underpins the case for small, scoped work. Cite as "(Standish CHAOS 2015)"; use sparingly.

A note on the numbers: this book is built on provenance. If a claim has no source above, it does not belong in the book.