A FIELD GUIDE

Information Science for the AI Era

A self-study primer on software architecture, data, and AI systems — for a product leader building in technology & banking

Written for a 10-hour flight · Read cover to cover, or jump in anywhere

Contents

HOW TO USE THIS BOOK · THE ONE-PAGE MAP

PART I — HOW SOFTWARE SYSTEMS ACTUALLY WORK

1. Thinking in systems: the journey of a single tap
2. Backend engineering, demystified
3. Software architecture: the shape of a system
4. How systems talk — and stay correct

PART II — DATA: THE BLOODSTREAM

5. Data foundations: where truth is stored
6. Data in motion: pipelines, streaming, and the single source of truth

PART III — AI & LARGE LANGUAGE MODELS

7. Machine learning, just enough
8. Inside a large language model
9. Building with LLMs
10. Agents and teams of agents

PART IV — BUILDING, SHIPPING, AND LEADING

11. The delivery lifecycle: from idea to production
12. Security, privacy, reliability & risk
13. System design in practice
14. The PM as technical leader

CAPSTONE & REFERENCE

Capstone: read your own product’s architecture

Glossary · What to study next

ORIENTATION

How to use this book

You don't need to write code to be dangerous in technical rooms. You need accurate *mental models*: a true-enough picture of how the pieces fit, so you can ask sharp questions, smell bad tradeoffs, estimate effort, and design products that engineers respect. That is what this book builds.

It is pitched at a curious product leader, not a fresher. It assumes you're comfortable with the *idea* of apps, APIs, and databases, and it takes you a layer deeper — into *why* systems are built the way they are, what problems each idea solves, and where the bodies are buried.

Throughout, you'll see recurring boxes:

EXPLAIN LIKE I'M 12

A plain-language analogy for a notoriously hard idea. If a concept ever feels slippery, read this box first, then the formal version.

WHY IT MATTERS

The real-world problem the idea solves — the "so what." If you remember nothing else from a section, remember this.

BANKING LENS

How the idea bites harder in a regulated, money-moving context, where mistakes are measured in lawsuits and lost trust.

PM MOVE

How to *use* the idea in your job: the question to ask, the tradeoff to surface, the decision to drive.

BUILD IT

A concrete tip for building real things, especially with AI.

TRY THIS

A short reflection or exercise — perfect for staring out the window at 38,000 feet.

Read it linearly for the full arc, or treat the contents page as a menu. Each chapter stands on its own.

THE BIG PICTURE

The one-page map

Almost every digital product — your banking app, a trading simulator, an AI assistant — is the same handful of layers arranged differently. Hold this shape in your head and everything later clicks into place:

```
YOU (a tap on glass)
     |
[ CLIENT ]      the app on your phone/browser: shows things, captures intent
     |  (a request travels over the network)
[ API / GATEWAY ]   the front door: checks who you are, routes the request
     |
[ SERVICES / BACKEND ]  the brains: business rules, "can Soe buy this?"
     |        \
[ DATA ]       [ OTHER SYSTEMS ]   databases, queues, caches, partners,
the memory      market-data feeds, payment rails, AI models
     |
[ INFRASTRUCTURE ]  the ground it all runs on: servers, cloud, networks
```

A request flows *down* this stack and a response flows back *up*. Data is the layer that *remembers*. AI is increasingly a service hanging off the side that the backend calls like any other. Everything in this book is either one of these boxes, the arrows between them, or how to build and run the whole thing safely.

WHY IT MATTERS

When something is “slow,” “broken,” or “expensive,” the cause lives in one of these layers or one of these arrows. Knowing the map lets you ask *where* instead of throwing up your hands. Half of technical leadership is localizing the problem to the right box.

PART I

How software systems actually work

The machinery beneath the glass

CHAPTER 1

Thinking in systems: the journey of a single tap

The problem it solves: a working app feels like magic — one surface, instant answers. In reality a dozen specialized parts cooperate across thousands of kilometers in under a second. To reason about products, you must see that hidden choreography.

Follow one tap all the way down

Suppose Soe taps "Buy" in your investing app. Here is the journey, which is essentially the same for any action in any app:

1. **The client** (the app) packages her intent — *buy 5 shares of MU* — into a structured message called a *request*.
2. That request travels over **the network**: phone → cell tower → the internet → a data center. This hop costs *latency* (time) and can fail.
3. An **API gateway** receives it at the "front door," checks *authentication* (is this really Soe?) and *authorization* (is she allowed?), and routes it onward.
4. A **backend service** runs the *business logic*: does she have enough cash? Is the market open? It then talks to the **database** to record the order and update balances.
5. A **response** — success or a specific error — travels all the way back up, and the client re-draws the screen.

EXPLAIN LIKE I'M 12

Ordering in a restaurant. You (the client) tell the waiter (the API) what you want. The waiter doesn't cook; they carry your order to the kitchen (the backend), which checks the pantry (the database) and cooks. The food comes back the same way it was ordered. You never see the kitchen — you just trust the meal arrives correct and hot.

Client vs. server: a fundamental split

The **client** is what runs on the user's device. It is great at presentation and capturing intent, but it cannot be trusted — a determined user can tamper with anything on their own phone. The **server** (backend) runs on machines *you* control. That's why every rule that matters — "you can't spend money you don't have" — must be enforced on the server, never only in the app.

BANKING LENS

The golden rule: **never trust the client.** If a balance check or transfer limit lives only in the app, an attacker simply bypasses the app and calls your API directly. In finance, server-side validation isn't a nicety; it's the line between a product and a fraud incident.

Latency: why distance and round-trips cost you

Light is fast but not instant, and every network hop adds delay. A round-trip to a server on another continent can cost 100–300 milliseconds *before* any work happens. Make ten such round-trips in sequence and your "instant" screen now takes two seconds. This is why fast apps do fewer, bigger requests, do work in parallel, and keep frequently-used data close to the user (a *cache*, Chapter 5).

WHY IT MATTERS

Speed is a feature users *feel* before they can name it. Amazon and Google both measured that every extra 100ms of latency measurably reduced sales and engagement. "Make it faster" usually means "make fewer round-trips" — a design decision, not a magic dial.

Synchronous vs. asynchronous: waiting vs. handing off

Some work you wait for (*synchronous*): you tap, you stare at a spinner, the answer returns. Other work you hand off and move on (*asynchronous*): you request a year-end tax statement and get "we'll email it" — the heavy job runs later. Choosing which is which is one of the most consequential design calls in any system.

EXPLAIN LIKE I'M 12

Synchronous is waiting at the counter for your coffee. Asynchronous is taking a buzzer and sitting down — the shop buzzes you when it's ready, so the queue doesn't freeze for everyone behind you.

PM MOVE

When a feature feels slow, ask: “Does the user actually need to *wait* for this, or can we do it in the background and notify them?” Turning a synchronous wait into an async hand-off is often the cheapest, biggest UX win available.

TRY THIS

Pick one action in a product you use daily. Sketch its journey through the seven layers of the one-page map. Where would *you* add a cache, and where would you make work asynchronous?

CHAPTER 2

Backend engineering, demystified

> **The problem it solves:** the backend is where correctness, money, and trust live. It is also the part PMs understand least, because it's invisible. This chapter makes the invisible concrete — and explains *why* good backend engineers are worth their weight in gold.

What a backend actually does

Strip away the jargon and a backend does four things:

- **Accepts requests** through an API and decides who's allowed to do what.
- **Runs business logic** — the rules of your product ("a transfer over €10,000 needs extra verification").
- **Reads and writes data** — the durable memory of the system.
- **Coordinates other systems** — payment rails, market-data feeds, email, AI models.

> EXPLAIN LIKE I'M 12
>
> The backend is the kitchen *and* the rule-keeper of a board game rolled into one. It cooks what you ordered, but it also makes sure nobody cheats: you can't move a piece you don't own, you can't take money from the bank that isn't yours, and if two players grab the same card at once, it decides who really got it.

Why backend is notoriously hard

Drawing a button is hard to make *beautiful*; a backend is hard to make *correct under chaos*. Four forces make it brutal:

1. Concurrency — many things at once

Thousands of users hit the system simultaneously. Two requests can try to touch the same data at the same instant. Get this wrong and you get a *race condition*: an outcome that depends on accidental timing.

> EXPLAIN LIKE I'M 12
>
> Imagine one cookie left on a plate and two kids reaching at once. If nobody's in charge, both might grab it and you've handed out a cookie that didn't exist. Computers do this millions of times a second, so they need a referee (a *lock* or a *transaction*) to say "one at a time for this cookie."

> BANKING LENS
>
> The classic disaster: an account with €100 gets two simultaneous €100 withdrawals. Without concurrency control, both read "balance = 100," both approve, and the bank just paid out €200 it didn't have. *Transactions* and *locks* exist precisely to make this impossible. This is why "just check the balance" is never as simple as it sounds.

2. Failure — everything breaks, all the time

Networks drop. Disks die. A partner's API times out mid-request. A robust backend assumes failure is normal and is built to recover: it retries safely, it can pick up where it left off, and it never ends up in a half-finished state (you transferred the money *out* but a crash meant it never arrived).

> WHY IT MATTERS
>
> Users forgive "try again in a moment." They never forgive "your money vanished." Most backend complexity that looks like over-engineering is actually buying you graceful behavior when — not if — something fails.

3. State — remembering things correctly over time

State is everything the system must remember: balances, orders, who's logged in. Managing state across many machines, while users change it concurrently, is the deep difficulty of backend work. The dream is *stateless* services (each request is self-contained) that push all the hard remembering down into purpose-built databases.

4. Scale — staying fast as you grow

Code that works for 100 users can melt at 100,000. Scaling comes in two flavors: **vertical** (a bigger machine — simple but has a ceiling) and **horizontal** (many machines sharing the load — powerful but introduces coordination problems). Horizontal scaling is why the “how systems talk” problems in Chapter 4 exist at all.

EXPLAIN LIKE I’M 12

Vertical scaling is hiring one super-fast chef. Eventually even the best chef has two hands. Horizontal scaling is hiring twenty normal chefs — far more capacity, but now they must coordinate so two don’t cook the same order or grab the same pan.

Idempotency: the most useful word you’ll learn today

An operation is **idempotent** if doing it twice has the same effect as doing it once. “Set balance to €100” is idempotent; “add €100” is not. Because networks fail and clients retry, backends lean heavily on idempotency so that a retried request doesn’t double-charge a customer.

EXPLAIN LIKE I’M 12

A light switch labeled “ON” is idempotent — flip it twice and the light’s still just on. A switch labeled “toggle” is not — flip it twice and you’re back to off. Banks want every payment button to behave like “ON,” never “toggle.”

PM MOVE

For any money-moving feature, ask the team: “What happens if the user double-taps, or the network retries this? Are we idempotent?” You’ll sound like you’ve seen the dragon — because now you have.

WHY IT’S EXCITING

A great backend is invisible craftsmanship: millions of people trust it with their money and never think about it once. Understanding it lets you tell the difference between a feature that’s “basically done” and one that’s actually safe to ship — the gap where real engineering lives.

TRY THIS

Take a feature you’re shipping. Write down: what state does it change, what happens if two users race on it, and what happens if it fails halfway. If you can’t answer, you’ve found your next conversation with engineering.

CHAPTER 3

Software architecture: the shape of a system

The problem it solves: any non-trivial product is too big to hold in one head. Architecture is how you carve it into parts that can be built, changed, and scaled independently — and the carving lines you choose haunt you for years.

The two words that explain most architecture: coupling and cohesion

Cohesion = how well the things inside one part belong together. **Coupling** = how much one part depends on the guts of another. The eternal goal: *high cohesion, low coupling.* Parts that each do one job well, and that lean on each other only through clean, stable interfaces — not by reaching into each other's internals.

EXPLAIN LIKE I'M 12

Think LEGO vs. a sandcastle. LEGO bricks connect through one standard stud pattern (low coupling), so you can swap a red brick for a blue one without rebuilding anything. A sandcastle is all one blob (high coupling) — poke one tower and a wall you didn't touch collapses. Good architecture is LEGO.

WHY IT MATTERS

Coupling is the hidden tax on every future change. The reason a "tiny" request sometimes costs three weeks is that the relevant code is tangled into ten other things. When engineers push back on a small ask, they're often telling you the architecture is coupled there.

Architectural styles, and the tradeoff behind them

Style	What it is	Good for	The catch
Monolith	One big deployable application	Early products; speed; simplicity	Hard to scale teams & parts independently as it grows
Microservices	Many small services, each owning one capability	Large orgs; independent scaling & teams	Huge operational complexity; the network is now everywhere (Ch.4)
Serverless	You ship functions; the cloud runs them on demand	Spiky/unpredictable load; small teams	Less control; cost surprises; cold starts
Event-driven	Parts react to a stream of events rather than calling each other directly	Decoupling; real-time; audit trails	Harder to trace "what happened" end-to-end

PM MOVE

Beware "microservices" as a status symbol. They solve an *organizational* problem (many teams shipping independently) at a steep technical cost. A startup splitting into 30 services before it has 30 engineers is usually buying pain it can't yet afford. The right question: "What problem is this boundary solving *for us, now*?"

Layering: separating concerns

Inside any service, code is usually layered: **presentation** (what the user sees) → **application/business logic** (the rules) → **data access** (talking to the database). Keeping these separate means you can change the screen without touching the rules, or swap the database without rewriting the product. The same instinct, at every scale, is "separation of concerns."

Architecture Decision Records: the artifact you should fall in love with

An **ADR** is a short document capturing one significant decision: the context, the options considered, the choice, and the consequences. They're the institutional memory of *why* the system is the way it is.

BUILD IT

When you make any non-obvious technical choice (“we’ll snapshot portfolio value on every trade rather than recompute it”), write a 1-page ADR. Six months later, when someone asks “why on earth did we do this,” you’ll have the answer — and you’ll avoid relitigating settled debates.

WHY IT’S EXCITING

Architecture is where product strategy becomes physical. The boundaries you draw decide which futures are cheap and which are nearly impossible. A PM who can read and shape architecture is choosing the company’s option space, not just its next sprint.

TRY THIS

Find a feature that’s “always more work than expected.” Ask whether it’s a coupling problem — is one change forced to ripple through many parts? That’s an architecture smell, and naming it is the first step to funding the fix.

CHAPTER 4

How systems talk — and stay correct

The problem it solves: the moment you have more than one machine or service, they must communicate over an unreliable network — and somehow still agree on what's true. This is the deep, beautiful, maddening field of *distributed systems*.

APIs: the contracts between parts

An **API** (Application Programming Interface) is a promise: "send me a request shaped like *this*, and I'll respond like *that*." It hides the messy internals behind a stable contract, so the caller doesn't care how the work gets done. Three common flavors:

- **REST** — the web's lingua franca; you act on "resources" (a user, an order) with standard verbs (get, create, update, delete). Simple, ubiquitous.
- **GraphQL** — the client asks for exactly the fields it wants in one query; great for rich mobile UIs that would otherwise make many REST calls.
- **gRPC** — fast, compact, strongly-typed; favored for service-to-service traffic inside a system.

EXPLAIN LIKE I'M 12

An API is a vending machine. You don't need to know how it stores or dispenses snacks. You just know: press B4, insert coins, get a specific chocolate bar. The buttons and slots are the "contract." As long as that stays the same, the machine's insides can be completely rebuilt and you'd never notice.

PM MOVE

API design *is* product design when partners or other teams build on you. A clean, stable API is a moat; a messy one is a tax everyone pays forever. Treat "what's the contract?" as a first-class product question, not an implementation detail.

Messages and events: talking without waiting

Instead of Service A calling Service B and waiting, A can drop a **message** onto a **queue**, and B picks it up when ready. Or A can *publish* an **event** ("OrderPlaced") that any interested service *subscribes* to. This decouples parts: B can be slow, restart, or scale up, and A doesn't care. It's the backbone of resilient, real-time systems.

EXPLAIN LIKE I'M 12

Direct calls are a phone call — both people must be free at the same second. A queue is a mailbox — you drop the letter and leave; the other person reads it whenever they're ready, and nothing breaks if they're napping.

BANKING LENS

Event streams double as an **audit trail**. If every change is an event ("moneyDebited," "limitChanged"), you can replay exactly what happened and when — gold for regulators, fraud investigations, and rebuilding state after a bug. This pattern is called *event sourcing*.

The hardest truth: you can't have everything (CAP)

When data lives on multiple machines and the network between them hiccups, you face a forced choice, captured by the **CAP theorem**: during a network **P**artition, you can stay **C**onsistent (everyone sees the same data, but some requests must wait or fail) *or* stay **A**vailable (everyone gets an answer, but answers may briefly disagree) — not both.

EXPLAIN LIKE I'M 12

Two shop tills that normally sync. The line between them goes down. Now: do you (a) stop selling the last concert ticket until the tills can talk again, so you never sell it twice (consistency), or (b) keep selling and risk selling it twice, sorting it out later (availability)? You genuinely cannot have both while the line is down. You must *choose* per situation.

WHY IT MATTERS

This is not academic. It's why your bank balance is rock-solid consistent (they choose C — better to delay than to be wrong about money) while a social feed's like-count is "eventually" right (they choose A — better to always load than to be perfectly accurate). Knowing which guarantee a feature needs is a core design decision.

Eventual consistency, retries, and idempotency — together

Many systems accept **eventual consistency**: data may be briefly out of sync but converges. To survive the network, callers **retry** failed requests — which is safe only if those requests are **idempotent** (Chapter 2). These three ideas are a package: distributed systems are mostly the art of being correct *despite* unreliability, not assuming it away.

WHY IT'S EXCITING

This is humanity coordinating thousands of imperfect machines into something that feels like one flawless service. It's genuinely one of the great engineering achievements of our era — and once you see the tradeoffs, you'll spot them in every product you touch.

TRY THIS

For your product, list three pieces of data. For each, decide: does it need strong consistency (money, inventory) or is eventual consistency fine (view counts, "last seen")? Notice how the answer changes the cost and the architecture.

PART II

Data: the bloodstream

Where truth is stored, how it moves, and why it drifts

CHAPTER 5

Data foundations: where truth is stored

The problem it solves: code is temporary; data is forever. The shape you give your data decides what questions you can answer cheaply, what's slow, and what's impossible. Choosing a database is choosing a worldview.

Data models: how you organize what you remember

A **data model** is the structure you impose on information — the tables, fields, and relationships. Get it right and features fall out easily; get it wrong and you fight it forever. The core tension is between *structure* (predictable, queryable, but rigid) and *flexibility* (easy to change, but messy).

The database family tree

Type	Shape	Best at	Example use
Relational (SQL)	Tables with rows & columns, strict relationships	Correctness, complex queries, transactions	Accounts, orders, ledgers
Document (NoSQL)	Flexible JSON-like documents	Evolving, varied data; speed of change	User profiles, product catalogs
Key-value	A giant dictionary: key → value	Blazing-fast lookups; caching	Sessions, feature flags
Time-series	Measurements stamped with time	Append-heavy, time-ordered data	Stock prices, portfolio value, metrics
Vector	Lists of numbers ("embeddings") you search by similarity	"Find things *like* this"	AI search, recommendations (Ch.9)
Graph	Nodes & relationships	Many-to-many connections	Fraud rings, social networks

EXPLAIN LIKE I'M 12

Databases are different kinds of organizers. A relational DB is a spreadsheet with strict rules. A document DB is a drawer of labeled folders that can each hold anything. A key-value store is a coat check — give a ticket, get your coat, instantly. A time-series DB is a diary, one dated entry after another. A vector DB is a "find me songs that *sound* like this one" machine.

OLTP vs. OLAP: doing vs. understanding

Two opposite jobs need two opposite designs. **OLTP** (transactional) handles the live business — many tiny, fast reads and writes ("place this order"). **OLAP** (analytical) handles understanding — few huge queries across years of data ("revenue by region by quarter"). You don't run heavy analytics on your live production database, or you'll slow down the very customers you're trying to analyze.

WHY IT MATTERS

This split is why companies copy data from the live system into a separate **data warehouse** for analytics. When an analyst says "the numbers are a day behind," this is usually why — and it's often a deliberate, healthy choice, not a bug.

Indexes: the difference between instant and unusable

An **index** is a pre-built lookup structure that lets the database find rows without scanning everything — like the index at the back of a book.

EXPLAIN LIKE I'M 12

Finding "dolphins" in a 900-page book: without an index you read every page (slow). With an index you flip to the back, see "dolphins — p.412," and jump straight there. Indexes are why an app with a billion records can still answer you in a blink. The cost: every index takes space and slightly slows down writing new data.

Transactions & ACID: the bedrock of trust

A **transaction** bundles several changes so they all succeed or all fail together — never half. Relational databases guarantee **ACID** properties: **A**tomic (all-or-nothing), **C**onsistent (rules never violated), **I**solated (concurrent transactions don't corrupt each other), **D**urable (once saved, it survives a crash).

BANKING LENS

A transfer is two changes: debit one account, credit another. ACID's atomicity guarantees you can *never* end up with the money debited but not credited, even if the server explodes between the two. This single guarantee is why banks have leaned on relational databases for fifty years. When someone proposes a trendy database for ledgers, "does it give us ACID transactions?" is the question that ends the debate.

Caching: keeping the hot stuff close

A **cache** is a small, fast copy of frequently-used data kept near where it's needed, so you don't pay the full cost of fetching it every time. It's the single most common performance trick in computing — and the source of the famous joke that the two hardest problems in software are "naming things, cache invalidation, and off-by-one errors." (The hard part is knowing when the cached copy is stale.)

WHY IT'S EXCITING

Data outlives every app built on top of it. The schema you design today may still be running the business in twenty years. Getting it right is one of the most leveraged things a team does — and being the PM who thinks clearly about data models makes you unusually valuable.

TRY THIS

For your product, name the one table whose corruption would be catastrophic. That's your "source of truth" — the thing that must be relational, transactional, backed up, and audited. Everything else can be more relaxed.

CHAPTER 6

Data in motion: pipelines, streaming, and the single source of truth

> **The problem it solves:** data is born in one place (a trade, a tap, a sensor) but needed in many (dashboards, AI models, reports, the customer's screen). Moving it reliably, on time, and without it drifting out of agreement is a discipline of its own.

Batch vs. streaming: two clocks

Batch processing gathers data and crunches it on a schedule ("every night, recompute everyone's balances"). **Streaming** processes each event the instant it arrives ("update the portfolio the moment a price ticks"). Batch is simpler and cheaper; streaming is fresher and harder.

> EXPLAIN LIKE I'M 12
>
> Batch is doing all your laundry on Sunday. Streaming is washing each sock the moment it gets dirty. Sunday laundry is efficient but you might be out of clean socks on Wednesday. Per-sock washing means always-fresh socks but a lot more running of the machine.

> BANKING LENS
>
> Recall the investing app's "portfolio takes 10–20 minutes to update after a trade" bug. That is the smell of a *batch* pipeline where users expect *streaming* freshness. The fix is to make a trade *emit an event* that updates value immediately, demoting the slow batch job to a nightly correctness check. Recognizing "this should be streaming, not batch" is a genuinely senior insight.

The modern data stack, in one breath

Most analytics setups are a pipeline: **ingest** (pull data from sources) → **store** (a warehouse or "lake") → **transform** (clean and reshape it) → **serve** (dashboards, models, reports). A **data warehouse** stores structured, query-ready data; a **data lake** stores raw everything (cheaper, messier); a **lakehouse** tries to be both.

The single source of truth — and why systems drift

When the same fact is stored or computed in several places, those places *will* eventually disagree — this is **data drift**. The cure is a **single source of truth (SSOT)**: one authoritative place each fact is computed, with everything else *deriving* from it rather than computing its own version.

> EXPLAIN LIKE I'M 12
>
> If three clocks in your house are each set by hand, they'll slowly disagree and you'll never know the real time. The fix isn't to keep resetting them — it's to have *one* master clock that all the others copy from. In software, that master clock is the single source of truth.

> BANKING LENS
>
> If the header shows one balance, the chart implies another, and the holdings list a third, the user stops trusting *all* of them. The fix: compute the value *once* and have the header, chart, and list all read that same number. **Reconciliation** — routinely checking that derived numbers still match the source — is a daily ritual in finance, and a drift alert is your early-warning system for a data bug.

Data quality, lineage, and governance

Three ideas that separate amateurs from professionals: **Quality** — is the data accurate, complete, on time? **Lineage** — can you trace any number back to where it came from and every step it took? **Governance** — who's allowed to see and change what, and can you prove it?

> WHY IT MATTERS
>
> "Garbage in, garbage out" is the iron law. The most sophisticated AI or dashboard built on bad data confidently produces bad answers — which is worse than no answer, because people trust it. In AI especially, data quality *is* product quality.

WHY IT'S EXCITING

Data is the raw material of both insight and intelligence. Every AI capability in this book is, at bottom, patterns learned from data. The teams that win the next decade are the ones whose data is clean, fresh, traceable, and trusted. Owning that mindset as a PM is a superpower.

TRY THIS

Pick a number your product shows users. Ask: where is it computed, how many *other* places recompute it, and how would you know if they disagreed? If the answer is "several places, and we wouldn't," you've found a drift bug waiting to happen.

PART III

AI & large language models

From “machines that learn” to teams of agents you can direct

CHAPTER 7

Machine learning, just enough

The problem it solves: some tasks are impossible to write step-by-step rules for — recognizing a cat, judging if a transaction is fraud, predicting a churn. Machine learning lets a system *learn the rules from examples* instead of being told them.

The one idea behind all of it

Traditional software: *you* write the rules, the computer follows them. Machine learning flips this: you show the computer many examples of inputs and correct outputs, and it *finds the rules itself* — rules often too subtle or numerous for any human to write.

EXPLAIN LIKE I'M 12

You never taught a toddler the "rules" for recognizing dogs — no checklist of ear-shapes and tail-lengths. You just pointed at hundreds of dogs and said "dog," and their brain figured out the pattern. Machine learning is that, for computers: lots of labeled examples in, a learned sense of the pattern out.

The three flavors

- **Supervised learning** — learn from labeled examples (emails marked "spam"/"not spam"). The workhorse of business ML.
- **Unsupervised learning** — find structure in unlabeled data (group customers into natural segments nobody pre-defined).
- **Reinforcement learning** — learn by trial, reward, and error (a system that gets "points" for good moves, like game-playing AIs — and, as we'll see, part of how chatbots are tuned).

Training vs. inference: two very different moments

Training is the expensive, one-time (ish) process of learning the pattern from data — think months of computation on giant machines. **Inference** is using the trained model to answer a new question — fast and cheap by comparison. The trained pattern is stored as millions or billions of numbers called **parameters** (or "weights").

EXPLAIN LIKE I'M 12

Training is studying all semester for an exam — slow, effortful, done once. Inference is answering a single question on the exam — quick, using what you already learned. The model's "parameters" are everything it crammed into its head during study.

PM MOVE

This split explains AI economics. Training a frontier model costs a fortune and only a few companies do it. *Using* one (inference) is cheap per call but adds up at scale. Your cost lever as a builder is almost always inference: how often you call, how big the model, how many tokens (Ch.9).

Embeddings: turning meaning into numbers

A pivotal trick: represent a word, sentence, image, or product as a list of numbers (a **vector**) such that *similar things end up near each other* in that number-space. These are **embeddings**. They're how machines do "find things like this" and the foundation of modern AI search (Ch.9).

EXPLAIN LIKE I'M 12

Imagine a giant map where every word has a location, and words that mean similar things sit close together — "king" near "queen," "dog" near "puppy," "bank-where-you-keep-money" far from "river-bank." An embedding is just the map coordinates for a thing. Once meaning is coordinates, "similar" becomes "nearby," which a computer can measure instantly.

Why models get things wrong: generalization, overfitting, bias

A good model **generalizes** — it works on new data it never saw. A model that **overfits** has merely memorized its training examples and flunks the real world. And a model trained on biased data learns those biases — a profound issue in lending, hiring, and anything touching people's lives.

BANKING LENS

If a credit model is trained on historical decisions that were themselves biased, it will faithfully reproduce and even amplify that bias — while looking objective. This is why regulated finance demands **explainability** and **model risk management**: you must be able to justify *why* a customer was declined, not just point at a black box.

WHY IT'S EXCITING

For the first time, software can handle the fuzzy, human, pattern-rich problems that rigid rules never could. That's the unlock behind the last decade — and large language models are the most general-purpose version of it yet.

TRY THIS

Name one decision in your product currently made by hand-written rules that's always "almost right but not quite." That fuzziness is often a sign the problem wants a learned model rather than more rules.

CHAPTER 8

Inside a large language model

The problem it solves: language is the universal interface to human knowledge and work. An LLM is a single model that can read and write language well enough to summarize, translate, code, reason, and converse — making it the most flexible building block in software history.

The shockingly simple core idea

At heart, an LLM does one thing: given some text, predict the next chunk of text. Repeat, feeding each prediction back in, and it writes paragraphs, code, or answers. Everything impressive — reasoning, translation, coding — emerges from doing this *extremely* well, having learned from a vast slice of human writing.

EXPLAIN LIKE I'M 12

It's the "predict the next word" game on your phone keyboard — but supercharged by reading basically the whole internet. Your keyboard suggests "you" after "thank." An LLM has read so much that it can continue "Write a poem about the sea in the style of…" convincingly. It's autocomplete that went to university for a thousand years.

Tokens: the units of LLM thought

LLMs don't see words exactly; they see **tokens** — chunks of text (roughly ¾ of a word on average). "Unbelievable" might be three tokens. You pay for AI by the token (in *and* out), and every model has a maximum number of tokens it can consider at once — its **context window**.

BUILD IT

Two practical consequences: (1) Cost and speed scale with tokens, so concise prompts and outputs save real money. (2) The context window is the model's "working memory" for one conversation — stuff too much in and it gets slower, pricier, and can lose the thread. Managing context is a real craft (Ch.9).

The transformer and "attention," without the math

The breakthrough architecture (2017) is the **transformer**, and its key trick is **attention**: when processing each word, the model weighs how much every other word matters to it. This lets it track context across long passages — who "she" refers to, which earlier clause a phrase depends on.

EXPLAIN LIKE I'M 12

Reading "The trophy didn't fit in the suitcase because *it* was too big," your brain instantly knows "it" = the trophy, not the suitcase. You did that by paying *attention* to the right earlier words. The transformer does the same: for every word, it asks "which other words should I focus on to understand this one?" Doing that across a whole document is what makes it feel like it "understands."

How a chatbot is made: pretrain, then post-train

- **Pretraining** — read an enormous amount of text, learning grammar, facts, and patterns by predicting next tokens. Result: a powerful but raw "knows-everything-says-anything" model.
- **Post-training / fine-tuning** — teach it to be helpful, follow instructions, and behave, partly via **RLHF** (Reinforcement Learning from Human Feedback): humans rate responses, and the model learns to produce the kind people prefer.

EXPLAIN LIKE I'M 12

Pretraining is a kid reading every book in the library — brilliant but blurts out anything. Post-training is finishing school: learning manners, how to answer the question actually asked, and what not to say. Same brain, much better behaved.

Sampling, temperature, and why answers vary

The model outputs *probabilities* for the next token and then picks one. **Temperature** controls how adventurous that pick is: low = focused and repeatable (good for facts, code); high = varied and creative (good for brainstorming). This is also why you can ask the same question

twice and get different answers.

Why they hallucinate — and what that really means

An LLM has no built-in notion of truth; it produces *plausible* text. When it doesn't know, it may still generate a confident, fluent, wrong answer — a **hallucination**. It's not lying; it's doing exactly what it was built to do (produce likely text), and likely-sounding isn't the same as true.

BANKING LENS

This single fact dictates safe AI design in finance: never let a raw LLM be the source of truth for a number, a balance, or a policy. Use it to *understand* and *communicate*, but ground every fact in a real system (a database, a calculation) — the pattern called *retrieval-augmented generation* (Ch.9). "Where did this number actually come from?" must always have a non-AI answer.

WHY IT'S EXCITING

One model, no retraining, can draft an email, explain tax law, write code, and tutor a child — a general-purpose reasoning-ish engine you summon with plain language. Knowing how it works (and where it's untrustworthy) is the difference between using it as a toy and wielding it as infrastructure.

TRY THIS

Ask an AI a factual question in your domain, then ask "how confident are you and what's your source?" Notice how fluent confidence and actual reliability are two different things — the core intuition for building safely.

CHAPTER 9

Building with LLMs

The problem it solves: a raw model in a chat box is a curiosity. Turning it into a reliable product feature — accurate, grounded, affordable, safe — is an engineering discipline. This chapter is your toolkit.

Prompting: programming in English

A **prompt** is your instruction to the model. Good prompting is surprisingly like good delegation: be clear about the goal, give context and constraints, show an example or two, and specify the format you want back. The **system prompt** sets persistent rules ("you are a careful banking assistant; never invent figures").

BUILD IT

Reliable patterns: (1) give the model a *role* and a goal; (2) provide examples of good output ("few-shot"); (3) ask it to *think step by step* for reasoning tasks; (4) demand a strict output format. Vague prompts get vague results — the model mirrors the care you put in.

Structured output & tool calling: from chatbot to component

You can require the model to answer in a strict machine-readable format (like JSON), so its output plugs straight into your code. Even more powerful: **tool calling** (a.k.a. function calling) lets the model *request actions* — "call get_stock_price(MU)" — and your code runs the real function and hands back the result. Now the LLM can *do* things, not just talk.

EXPLAIN LIKE I'M 12

Plain LLM = a brilliant friend locked in a room with no phone; they can only talk. Tool calling hands them a phone and a to-do list: now they can look up real facts, press real buttons, and check real answers instead of guessing. That's the leap from "chatbot" to "assistant that gets things done."

RAG: grounding answers in your real data

Retrieval-Augmented Generation is the most important pattern for trustworthy AI products. Instead of hoping the model "knows," you first *retrieve* the relevant real documents (using embeddings + a vector database, Ch.5&7) and feed them into the prompt, then ask the model to answer *only from that*. The model becomes a brilliant reader of *your* facts rather than a guesser.

EXPLAIN LIKE I'M 12

Closed-book exam vs. open-book exam. A raw LLM takes everything from memory and sometimes misremembers. RAG is open-book: before answering, it's handed the exact right pages from the textbook and told "answer using these." Far fewer made-up answers, and you can *cite the page*.

BANKING LENS

RAG is how you build an AI that answers from *your* policies, *this* customer's real data, and *today's* prices — with citations — rather than from the model's frozen, generic memory. It's the bridge between "impressive demo" and "defensible in front of a regulator."

Three ways to specialize a model — pick the cheapest that works

Approach	What it does	When to use
Prompting	Just instruct the model well	Default. Try this first, always.
RAG	Feed it your relevant data at query time	When answers must reflect *your* facts / fresh data
Fine-tuning	Further-train the model on your examples	When you need a consistent style/format or a narrow skill at scale — expensive, do last

PM MOVE

Teams reach for fine-tuning far too early. The ladder is almost always: prompt → add examples → add RAG → *only then* consider fine-tuning. Each rung is cheaper, faster, and easier to change than the next. Push your team to climb in order.

Evals: the test suite for AI

Because AI output varies and can degrade silently, you need **evals** — a repeatable set of test cases scoring whether the AI does its job well. Without them, "the model feels worse this week" is an argument, not a fact; with them, it's a number you can act on.

BUILD IT

Evals are to AI what unit tests are to code. Start collecting real examples of good and bad outputs from day one. The teams that win with AI aren't the ones with the cleverest prompts — they're the ones who can *measure* quality and so improve it deliberately.

Guardrails, cost, and latency: the production realities

- **Guardrails** — checks around the model: validate inputs, filter unsafe outputs, defend against **prompt injection** (malicious text that tries to hijack the model's instructions — Ch.12).
- **Cost** — priced per token; use the smallest model that passes your evals, cache repeated calls, trim context.
- **Latency** — big models are slower; stream output so users see words appearing, and pick model size per task.

WHY IT'S EXCITING

These techniques turn an unreliable oracle into a dependable product ingredient. The gap between "cool demo" and "trusted feature" is exactly this engineering — and right now, very few people truly understand it. Being one of them is a career edge.

TRY THIS

Take an AI feature idea and map it: what's the prompt, what real data does it need (RAG?), what tools must it call, how will you measure if it's good (evals), and what could go wrong (guardrails)? That five-part sketch is a genuine AI PRD.

CHAPTER 10

Agents and teams of agents

The problem it solves: a single prompt answers a question. But real work is multi-step: research, decide, act, check, repeat. An **agent** is an LLM put in a loop with tools and memory so it can pursue a goal across many steps — and several agents can divide labor like a team.

Anatomy of an agent

An agent is four things working in a loop:

- **A model** (the reasoning core).
- **Tools** (things it can *do*: search, call APIs, run code, query a database).
- **Memory** (what it's learned so far — short-term context, and long-term stores it can look things up in).
- **A loop**: observe → think → act with a tool → observe the result → repeat until the goal is met.

EXPLAIN LIKE I'M 12

A regular LLM is answering one trivia question. An agent is a capable intern with a goal, a phone, a notepad, and the internet: "Book me the cheapest flight Friday." They search, compare, maybe hit a dead end and try again, jot notes as they go, and come back when it's done — making many small decisions on the way, not just one.

The reason-and-act loop

The dominant pattern is simple: the agent *reasons* about what to do next, *acts* by calling a tool, *observes* the outcome, and loops. This is how it recovers from surprises — if a search returns junk, it can rethink and try another approach, rather than barreling ahead. The loop is what makes agents feel autonomous.

Teams of agents: orchestration patterns

Once one agent works, you can compose many. Common shapes:

- **Orchestrator–workers** — a "manager" agent breaks a goal into subtasks and hands them to specialist worker agents (researcher, writer, checker), then assembles the results.
- **Pipeline** — agents in a line, each transforming the previous one's output (draft → edit → fact-check).
- **Debate / critic** — one agent produces, another critiques, improving quality through challenge.

EXPLAIN LIKE I'M 12

One smart intern is great. A small *team* — a researcher, a writer, and a fact-checker who reviews the writer — is better for big jobs. The trick is the same as with people: clear roles, clear hand-offs, and someone making sure it all comes together. Too many cooks, though, and it's chaos.

PM MOVE

Designing a multi-agent system is mostly an *org-design* problem you already know: define roles, responsibilities, hand-offs, and a way to check the work. Your product instincts about clear ownership and good interfaces transfer almost directly — this is a place your PM brain is an advantage, not a gap.

The Model Context Protocol (MCP) and standard tools

For agents to use tools and data sources without custom glue each time, the industry is converging on standards — notably **MCP** (the Model Context Protocol), a common way to expose tools, data, and actions to any AI model. Think of it as a universal adapter so agents can plug into your systems (and partners') in a consistent, governable way.

When NOT to use agents, and keeping a human in the loop

Agents are powerful and *unpredictable*: more autonomy means more ways to go wrong, more cost, and harder debugging. For well-defined tasks, a simple prompt or a fixed sequence of steps is more reliable than a free-roaming agent. And for anything consequential — moving

money, sending a customer message — keep a **human in the loop** to approve high-stakes actions.

BANKING LENS

The dangerous combination is autonomy plus authority. An agent that can both *decide* and *act* on real money or customer data needs hard limits: scoped permissions, spending caps, mandatory human approval for big actions, and a complete log of everything it did (back to event sourcing, Ch.4). Autonomy is earned incrementally as trust and evals accumulate.

WHY IT'S EXCITING

This is the frontier: software that doesn't just respond but *pursues goals* — researching, building, coordinating. Done well, a PM can soon direct a small "team" of agents the way they direct people, multiplying what one person can build. Done carelessly, it's an expensive, unaccountable mess. The judgment of *where* and *how much* autonomy to grant is the new core skill — and it's squarely a product skill.

TRY THIS

Take a multi-step task you do weekly. Break it into roles a team of agents could own, mark each hand-off, and circle every step where a human *must* approve before action. You've just designed an agentic workflow — and found its safety rails.

PART IV

Building, shipping, and leading

Turning systems into products, and products into outcomes

CHAPTER 11

The delivery lifecycle: from idea to production

The problem it solves: writing code is a fraction of the job. Getting it safely, repeatedly, and quickly into users' hands — and keeping it healthy — is what separates teams that ship from teams that thrash.

Version control (Git): the time machine

Git records every change to the code, who made it, and why — and lets many people work in parallel on *branches* that later *merge*. It's the substrate of all modern software collaboration.

EXPLAIN LIKE I'M 12

Git is infinite save-points in a video game, for a whole team. Everyone can explore their own path (a branch), and if someone messes up, you reload an earlier save. When two people's paths are both good, you combine them (merge). Nobody's work gets stomped on.

CI/CD: the assembly line

Continuous Integration automatically tests every change the moment it's proposed. **Continuous Delivery/Deployment** automatically packages and ships changes that pass. Together they turn releasing from a scary quarterly event into a calm, daily (or hourly) non-event.

WHY IT MATTERS

Shipping speed compounds. A team that can safely deploy ten times a day learns from real users ten times faster than one that ships monthly. Most of that speed comes not from typing faster but from *automation and tests* that make change safe.

The testing pyramid

Tests catch mistakes before users do. The healthy mix is a **pyramid**: many fast **unit tests** (one small piece), fewer **integration tests** (pieces together), and a handful of slow **end-to-end tests** (the whole flow). Plus, for visual products, *visual-regression* tests; for AI, *evals* (Ch.9).

Environments & feature flags: de-risking change

Code flows through **environments** — *development* (your machine), *staging* (a production-like rehearsal), *production* (real users). **Feature flags** let you ship code "off," then turn it on for 1% of users, then 50%, then everyone — and instantly off again if it misbehaves. This decouples *deploying* code from *releasing* a feature.

PM MOVE

Feature flags are a product superpower: gradual rollouts, A/B tests, instant kill-switches, and "dark launches" (ship and test in production before announcing). Ask for them by name — they convert risky big-bang launches into reversible, measurable ones.

Observability & SRE: knowing your system is healthy

Once live, you must *see* the system: **logs** (what happened), **metrics** (how much/how fast), and **traces** (one request's path through many services). **Site Reliability Engineering** formalizes this with **SLOs** (Service Level Objectives) — explicit targets like "99.9% of requests succeed in under 300ms" — and an **error budget**: the small amount of failure you're willing to spend, which balances reliability against shipping speed.

EXPLAIN LIKE I'M 12

Observability is the dashboard of a car. Logs are the trip diary, metrics are the speedometer and fuel gauge, traces are GPS showing the exact route taken. Without the dashboard you only find out something's wrong when the engine's already smoking. With it, you see the warning light early.

WHY IT'S EXCITING

This is the machinery that lets a tiny team run something millions depend on, around the clock, and sleep at night. Treating reliability as a measurable budget — not a vague aspiration — is one of the most mature ideas in the industry.

TRY THIS

Write one SLO for your product’s most important action (“X% of buys confirm in under Y seconds”). Notice how a fuzzy “it should be fast” becomes a number engineering can design toward and you can hold the line on.

CHAPTER 12

Security, privacy, reliability & risk

The problem it solves: in banking, a single breach or wrong number can mean ruined customers, regulatory fines, and lost trust that never returns. Security and risk aren't a feature — they're the licence to operate.

The two gates: authentication and authorization

Authentication = proving *who* you are (password, passkey, biometric). **Authorization** = what you're *allowed* to do once identified. Confusing them is a classic, costly bug.

EXPLAIN LIKE I'M 12

Authentication is showing your ID at the airport to prove you're really you. Authorization is your boarding pass deciding which plane and which seat you can actually board. Being correctly identified doesn't mean you're allowed everywhere — two separate checks.

Encryption: secrets in transit and at rest

Encryption scrambles data so only holders of the key can read it — protecting it *in transit* (moving across the network) and *at rest* (sitting in a database). It's the reason a thief who steals the disks or taps the wire still gets gibberish.

EXPLAIN LIKE I'M 12

Encryption is a locked diary. Even if someone steals it, the words are scrambled nonsense without the key. "In transit" is the diary locked while being mailed; "at rest" is it locked in a drawer at home. You want it locked in both places.

Threat modeling and the usual suspects

Threat modeling is calmly asking, before you build: who might attack this, what do they want, and how would they try? Most attacks are unglamorous and well-known (the **OWASP Top 10** catalogs them): injection attacks, broken access control, leaked secrets, and so on. Defending against the known basics prevents the large majority of real incidents.

BANKING LENS

Beyond engineering, finance is governed by hard rules: **GDPR** (EU data privacy — consent, data minimization, the right to be forgotten), **PSD2** (secure access and strong customer authentication for payments), **KYC/AML** (know your customer / anti-money-laundering), and increasingly the **EU AI Act** (obligations scaled to an AI system's risk). These aren't optional. "Is this compliant?" belongs in every product review, early — retrofitting compliance is brutally expensive.

AI-specific risks (new and sharp)

- **Prompt injection** — malicious text hidden in data the model reads (a web page, a document) that hijacks its instructions. The agentic equivalent of a con artist slipping your assistant fake orders.
- **Data leakage** — sensitive data sent to a model or pasted into a tool may be logged, retained, or exposed. Know what leaves your walls.
- **Model risk** — the model is wrong, biased, or drifts over time; in regulated use you must monitor, validate, and explain it.
- **Over-trust** — fluent confidence (Ch.8) leads people to skip verification. Design for healthy skepticism.

PM MOVE

Make "what could go wrong, and who could abuse this?" a standing agenda item, not a launch-week scramble. The PMs who raise risk early are seen as senior; the ones who discover it in an incident review are not. Privacy and security designed in are cheap; bolted on, they're ruinous.

WHY IT'S EXCITING

Trust is the entire product in finance. Everything else — features, design, AI — is built on the belief that you'll keep customers' money and data safe. Mastering this domain means you're protecting the foundation everything else stands on.

TRY THIS

For your next feature, spend ten minutes as the attacker: how would *you* abuse it, leak data through it, or trick its AI? Write down three attacks and one defense for each. That ten-minute habit prevents most incidents.

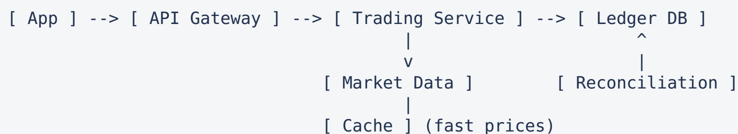
CHAPTER 13

System design in practice

> **The problem it solves:** ideas become real only when you can sketch how the pieces fit, estimate whether it'll hold up, and reason about cost. This is the everyday craft of turning a vision into a buildable system — and a language you can share with engineers.

Reading and drawing architecture diagrams

Most system designs are boxes (components) and arrows (data/requests flowing between them). You don't need to draw them perfectly; you need to *read* them, spot the bottleneck, and ask "what happens to this arrow if that box is down or slow?" A diagram you can interrogate is worth a hundred vague conversations.



```
[ App ] --> [ API Gateway ] --> [ Trading Service ] --> [ Ledger DB ]
                                         |                     ^
                                         v                     |
                                  [ Market Data ]        [ Reconciliation ]
                                         |
                                  [ Cache ] (fast prices)
```

Back-of-the-envelope estimation

Senior technical thinking often comes down to rough math done early. How many users × actions each × data per action = roughly how much traffic and storage? You're not seeking precision; you're checking whether an idea is "obviously fine," "obviously impossible," or "needs real thought."

EXPLAIN LIKE I'M 12

Before planning a party you guess: ~30 guests, ~2 slices of pizza each = ~60 slices = ~8 pizzas. You don't count exact bites; you just check you're not ordering 2 pizzas (disaster) or 200 (waste). Engineers do this with users, requests, and storage to size a system before building it.

PM MOVE

Do this in the room: "If this hits a million users doing five actions a day, is that thousands or billions of operations — and does our design care?" A quick order-of-magnitude check catches doomed ideas before they consume a quarter, and earns instant credibility.

Scalability, bottlenecks, and cost (FinOps)

A system is only as fast as its slowest part — its **bottleneck**. Scaling is the art of finding and relieving the current bottleneck (more machines, caching, async work), knowing a new one will appear. And in the cloud, every one of these choices has a price tag — **FinOps** is treating cloud and AI cost as a first-class design constraint, not a quarterly surprise.

WHY IT MATTERS

"It works on my laptop" and "it works for ten million users without bankrupting us" are different engineering problems. Performance and cost are designed in from the start — and with AI, where each call costs real money, cost-awareness is suddenly everyone's job, including the PM's.

The system-design method (use it for any feature)

1. **Clarify** the goal and constraints (scale, latency, consistency, budget).
2. **Sketch** the components and the data flow between them.
3. **Choose** the data stores and the consistency each needs (Ch.4–5).
4. **Stress it**: where's the bottleneck, what fails, what's the cost at scale?
5. **Decide & record** the tradeoffs (an ADR, Ch.3).

WHY IT'S EXCITING

This is the closest thing to a universal problem-solving language in tech. The same five steps design a payments system, an AI agent platform, or a charting engine. Once it's in your bones, you can sit in any technical room and contribute — not as a spectator, but as a designer.

TRY THIS

Take a feature on your roadmap and run all five steps on a napkin (or a tray table). Where did you have to guess? Those guesses are exactly the questions to bring to your architect.

CHAPTER 14

The PM as technical leader

The problem it solves: all of this knowledge is only valuable if it changes how you lead. This chapter turns understanding into leverage — better decisions, sharper conversations, and teams (human and agent) that ship great things.

Fluency, not expertise

You don't need to out-engineer engineers; you need enough fluency to *follow the tradeoff*, ask the question that exposes risk, and translate between business and build. Fluency earns trust, and trust is what lets you influence technical decisions without authority over them.

Estimation and the tradeoff triangle

Every feature trades among **scope, time, quality,** and **cost** — you can't max all four. The mature conversation isn't "can we do it?" but "what are we willing to trade?" And when engineers estimate in wide ranges, that's honesty about uncertainty, not evasion — help them narrow it by clarifying scope, not by pushing for a smaller number.

PM MOVE

When you hear "that's a big change," get curious, not pushy. Is it big because of *coupling* (architecture, Ch.3), *unknowns* (needs a spike), or *genuine scope*? Each has a different response. Naming which one it is, in their language, is the move that marks a technical PM.

Build vs. buy, and tech debt

Build vs. buy: build what's core and differentiating; buy/rent the rest (you wouldn't build your own database or payment rails). **Tech debt** is the accumulated cost of past shortcuts — sometimes a wise loan taken to ship fast, sometimes reckless borrowing. Like financial debt, a little is leverage; too much, and interest payments (slow, fragile development) eat you alive.

EXPLAIN LIKE I'M 12

Tech debt is leaving your room messy to get to the party faster. Fine occasionally. But if you *never* tidy, soon you can't find anything, can't open the door, and a five-minute task takes an hour. Teams "pay down debt" by cleaning up so they can move fast again.

Platform thinking

The highest-leverage move is often to build a *capability* many features reuse, rather than one-off features. The shared charting engine in a finance app, the reusable AI-evaluation harness, the internal agent framework — these are platforms. They cost more up front and pay back across every future feature.

Leading teams that build with AI — and teams of agents

AI changes the team's shape. More work becomes *specifying clearly*, *evaluating output*, and *setting guardrails* — the very things product leaders are already good at. As agents take on multi-step work (Ch.10), your job increasingly resembles *directing*: defining goals and roles, deciding where humans must stay in the loop, and judging quality. The scarce skill becomes clear thinking about *what good looks like* and *how much autonomy to grant*.

BANKING LENS

In a regulated, AI-augmented org, the leaders who thrive can hold two things at once: the ambition to build fast with AI, and the discipline to demand traceability, approval gates, and explainability. That balance — bold *and* accountable — is exactly the judgment that gets trusted with the biggest bets.

WHY IT'S EXCITING

You are arriving at a rare moment: the tools to build are collapsing in cost, and the bottleneck is shifting from "can we build it?" to "do we understand the system, the data, the risk, and the user well enough to build the *right* thing safely?" That is a product leader's question. The technical fluency in this book is what lets you answer it — and out-build people who have only one half.

TRY THIS

Reflect on your last hard technical decision. With what you now know, which question would you have asked differently? Write it down. That delta is this book turning into leverage.

CAPSTONE & REFERENCE

Putting it together

One worked example, then the tools to keep going

CAPSTONE

Read your own product’s architecture

> Let’s assemble everything into one story, using a familiar shape: an investing app where the core feature is a fast, trustworthy chart. Watch how every chapter shows up.

The walk-through

A user taps “Buy.” The **client** (Ch.1) sends a request over the network; the **API gateway** authenticates and authorizes her (Ch.12); the **trading service** (Ch.2) runs the business rules — enough cash? market open? — using a **transaction** so the debit and the position update are all-or-nothing (Ch.5). The trade **emits an event** (Ch.4) that updates her portfolio value *immediately* instead of waiting for a nightly **batch** — fixing the “10-minute lag” by choosing **streaming** freshness (Ch.6).

The portfolio value is computed in *one* place — a **single source of truth** — that the header, chart, and holdings all read, so they can never **drift** apart (Ch.6). Prices come from a market-data feed, kept close in a **cache** for speed (Ch.5). The chart is a reusable component — a small **platform** (Ch.14) — tested with **golden and visual-regression tests** (Ch.11) and held to an **SLO** (“chart paints in one frame”) with an **error budget** (Ch.11).

Now add intelligence: an AI assistant explains *why* the portfolio moved. It must never invent numbers (Ch.8), so it uses **RAG** (Ch.9) — retrieving the user’s real positions and today’s real news — and **tool calls** to fetch exact figures, with **guardrails** against **prompt injection** from news content (Ch.12). If you let an **agent** (Ch.10) act — say, rebalance — you scope its permissions and require **human approval** for anything touching money (Ch.12), logging every step as events for the **audit trail** (Ch.4). The whole thing is sized with **back-of-the-envelope** math and watched with **observability** (Ch.11–13), and every big choice is captured in an **ADR** (Ch.3).

> WHY THIS IS THE POINT
>
> No single idea here is exotic. The skill — the thing this book is really teaching — is holding the *whole chain* in your head at once: seeing how a product decision ripples down into architecture, data, AI, risk, and operations, and back up into user trust. That is what it means to think like an information scientist who builds.

REFERENCE

Glossary

ACID — database guarantees (Atomic, Consistent, Isolated, Durable) that make transactions trustworthy.

ADR — Architecture Decision Record; a short note capturing why a technical choice was made.

Agent — an LLM in a loop with tools and memory, pursuing a goal across multiple steps.

API — a contract for how software talks to other software.

Asynchronous — work handed off to finish later, rather than waited on.

Authentication / Authorization — proving who you are / what you're allowed to do.

Batch vs. streaming — processing data on a schedule vs. the instant it arrives.

Cache — a fast nearby copy of frequently-used data.

CAP theorem — during a network partition you must trade consistency against availability.

CI/CD — automated testing and releasing of code changes.

Context window — how much text (in tokens) an LLM can consider at once.

Coupling / Cohesion — how dependent parts are on each other / how well a part's contents belong together. Aim: low coupling, high cohesion.

Embedding — a list of numbers representing meaning, so "similar" becomes "nearby."

Eventual consistency — data that's briefly out of sync but converges.

Event sourcing — storing every change as an event, giving a replayable audit trail.

Feature flag — a switch to turn features on/off for some users without redeploying.

Fine-tuning — further-training a model on your own examples.

Hallucination — a confident, fluent, wrong answer from an LLM.

Idempotent — an operation safe to repeat; doing it twice equals doing it once.

Index — a lookup structure that lets a database find data without scanning everything.

Inference — using a trained model to answer (vs. training it).

Latency — the time delay before work happens; often dominated by network round-trips.

LLM — large language model; predicts the next chunk of text, very well.

MCP — Model Context Protocol; a standard way to give AI models tools and data.

Microservices / Monolith — many small independent services vs. one large application.

Observability — seeing inside a running system via logs, metrics, and traces.

OLTP / OLAP — live transactional workloads vs. heavy analytical ones.

Prompt injection — malicious text that hijacks an LLM's instructions.

RAG — Retrieval-Augmented Generation; grounding an LLM in real, retrieved data.

RLHF — Reinforcement Learning from Human Feedback; tuning a model toward preferred answers.

Race condition — a bug where the outcome depends on accidental timing of concurrent operations.

Single source of truth — one authoritative place a fact is computed; everything else derives from it.

SLO / Error budget — an explicit reliability target / the allowed amount of failure.

Token — the chunk of text an LLM processes and is billed by (~¾ of a word).

Transaction — a bundle of changes that all succeed or all fail together.

Vector database — a store optimized for finding items by similarity of their embeddings.

REFERENCE

What to study next

If you want the systems & architecture path

- *Designing Data-Intensive Applications* (Martin Kleppmann) — the definitive, readable bible on data and distributed systems. The one book to own.
- *A Philosophy of Software Design* (John Ousterhout) — short, sharp, on complexity and good design.
- The “System Design Primer” (free, online) — a structured tour of the building blocks.
- *Fundamentals of Software Architecture* (Richards & Ford) — styles and tradeoffs, for the architecture-curious.

If you want the AI/LLM path

- Anthropic’s and other model-makers’ engineering guides on prompting, tool use, evals, and building agents — practical and current.
- *AI Engineering* (Chip Huyen) — building real products on top of foundation models.
- 3Blue1Brown’s neural-network and transformer videos — the best visual intuition for how LLMs work.
- Andrej Karpathy’s talks/videos on LLMs — clear mental models from a builder.

If you want the build-it-yourself path

- Keep building prototypes with an AI coding assistant — the fastest way to make all of this concrete. Each idea in this book becomes real the moment you hit it in your own project.
- Pick one chapter’s “Try this” and apply it to a current product this week.

A CLOSING THOUGHT

You came in wanting to understand how systems, data, and AI fit together so you can build better and lead teams — including teams of agents. The secret you now hold is that it’s all *one* picture: requests flowing through layers, truth stored and kept consistent, intelligence grounded in real data, and humans deciding what’s worth building and where the guardrails go. You don’t need to know every detail. You need the map, the vocabulary, and the instinct for the right question — and now you have all three. Enjoy the flight.