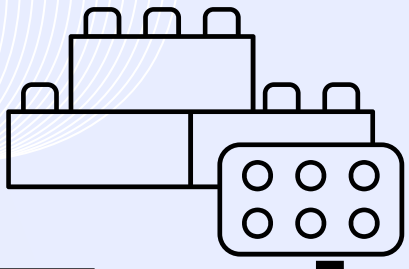


Tech Jargon

Part 8

for

Product Managers

Building Blocks

Monoliths | Microservices | Events | Caching

From an Engineer-Turned-PM

The System Architecture Mental Model

A way of understanding the **big shapes of your system** — monolith vs. microservices, events, queues, databases — and how those shapes determine what's easy to change and what fights you every step of the way.

Get this wrong, and you promise features that fight the architecture. Get it right, and you plan with the system's grain, not against it.

When you understand this mental model, you can:

- See why a feature in one service is a sprint but across services is a quarter
- Ask about sync vs. async, caching, and consistency before committing to a timeline
- Recognize when a "simple" request means coordinating across multiple services and teams
- Reason about database tradeoffs instead of treating storage as someone else's problem

This guide covers the 10 essential terms that bring this mental model to life.

Monolith

One codebase that does everything

A monolith is a single application where all features — auth, payments, notifications, search — deploy together as one unit. Fast early on: one repo, one deploy, one team. As the product grows, it gets painful — but a well-structured "modular monolith" is often the right choice.

One big kitchen where every chef works in the same room. Fast with a small team — chaotic when 20 cooks reach for the same stove.

We can't deploy the checkout fix until the search team's code passes tests — it's all one deploy.

For PMs, this means monoliths aren't outdated — many teams choose them deliberately. The question isn't "when do we move to microservices?" but "is this monolith structured enough for our scale?"

Microservices

Many small services that each own one thing

Microservices split your product into independent services — each owns one domain (payments, users, notifications) with its own codebase, database, and deploy cycle. Teams ship independently. But independence costs: more network calls, more failure points, and cross-service features need coordination.

A food court vs. one big kitchen. Each stall runs independently — but if a customer wants a meal from three stalls, someone has to coordinate the order.

The feature touches the user service, billing, and notifications — that's three teams and three deploy cycles.

For PMs, this means single-service changes are fast but cross-service features are slow. "It touches multiple services" isn't an excuse — it's the architecture telling you the real scope.

Dependency

When one thing can't work without another

A dependency is when one part of the system relies on another to function — Service A calls Service B, which queries Database C. Dependencies aren't bad, but hidden ones are. They're why a "small change" can break something three services away.

A chain of dominoes — push one, and the rest follow. The longer the chain, the harder to predict where the last one falls.

We can't upgrade the auth service — payments depends on an old version of its API.

For PMs, this means before scoping a feature, ask: "What depends on this, and what does this depend on?" The answer shapes your timeline more than the feature itself.

Event-driven architecture

Systems that react to things that happen

Services communicate by publishing events — "order placed," "user signed up," "payment failed" — and other services listen and react. Instead of calling each other directly, a service announces what happened, and anything that cares responds independently.

A bulletin board — you pin an announcement, and anyone interested reads it. No need to knock on every door.

When a user upgrades we publish an event — billing, notifications, and analytics all pick it up independently.

For PMs, this means adding a new reaction doesn't require changing the original service. But debugging is harder: cause and symptom can be in completely different services.

Message queue

A waiting line that absorbs the rush

A buffer between services — one puts a message in, another picks it up when ready. The sender drops the work and moves on instead of waiting.

A deli counter — you take a number, the staff works through orders at their own pace. Nobody cuts the line, the kitchen doesn't collapse during the rush.

We queue the export jobs instead of running them inline — otherwise a big export blocks the whole API for other users.

For PMs, this means queues are why some things aren't instant — by design. "Your report will be ready in a few minutes" usually means a queue is involved.

Coupling / Decoupling

How tangled — or independent — your systems are

Coupling is how much one part of the system depends on the internals of another. Tight coupling means a change in one place forces changes elsewhere. Decoupling means services interact through clean interfaces so they can change independently.

Puzzle pieces vs. Lego bricks. Puzzle pieces only fit one way — move one and you break the picture. Lego bricks snap apart and reconnect in any order.

The frontend is tightly coupled to the API response shape — if we change the payload, the app breaks.

For PMs, this means tight coupling is why "just move this field" turns into a multi-sprint project. When engineers push to decouple, they're investing in future speed.

Eventual consistency

Data that syncs — but not instantly

After a change, different parts of the system might briefly show different data — but will converge to the same state. It's the tradeoff for speed and availability. Strong consistency guarantees everyone sees the same data immediately, but it's slower.

Different ATMs showing different balances for a few seconds after a transfer. The money moved — the screens just haven't caught up yet.

The user updated their profile but the old name still shows on the dashboard — it's eventually consistent, give it a few seconds.

For PMs, this means stale data after an action might not be a bug — it might be the architecture. Know which parts of your product are eventually consistent and design the UX around it.

Caching

Storing copies so you don't fetch twice

Storing a copy of frequently requested data closer to where it's needed — so the system doesn't hit the source every time. Fast, but cached data goes stale. Cache invalidation (deciding when the copy is outdated) is one of the hardest problems in software.

Keeping a photocopy on your desk instead of walking to the filing room. Fast — until the original gets updated and your copy is wrong.

The product page loads in 200ms because it's cached — but when pricing updates, we need to bust the cache or users see old prices.

For PMs, this means when a change doesn't show immediately, the answer is often "it's cached." Ask how long the cache lives and what triggers a refresh.

SQL vs NoSQL

Structured tables vs. specialized storage

SQL databases (PostgreSQL, MySQL) store data in structured tables with defined relationships — great for complex queries. NoSQL is an umbrella: document stores (MongoDB), key-value (Redis), column stores (Cassandra), graph databases (Neo4j). Most products use SQL as the backbone and add specialized databases where needed.

A filing cabinet with labeled folders (SQL) vs. specialized tools — a Rolodex, a pinboard, a spreadsheet — each designed for a specific job (NoSQL).

User profiles are in Postgres, but activity logs go to DynamoDB — high-volume, write-heavy, no joins needed.

For PMs, this means you don't pick the database — but when engineers say "that query would be expensive," the database type is usually why.

Synchronous vs Asynchronous

Wait for the answer vs. move on

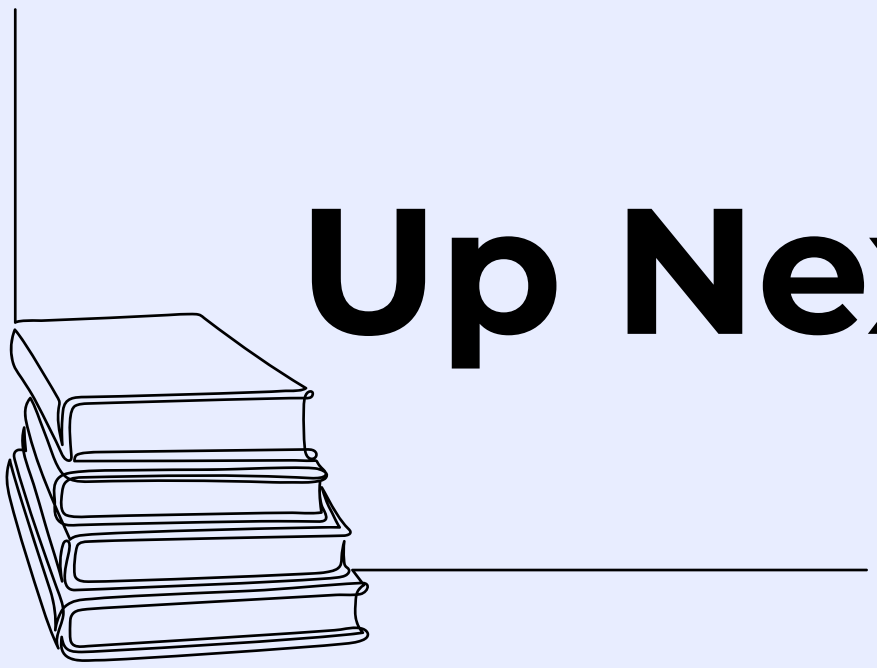
Synchronous: the caller waits for a response before continuing. Asynchronous: the caller sends a request and moves on — the result comes back via a callback, notification, or status check. Sync is simpler but creates bottlenecks. Async is resilient but harder to trace.

A phone call (sync) vs. a text message (async). On a call, you wait. With a text, you send it and move on — the reply comes when it comes.

Checkout is synchronous — we wait for the payment provider to confirm — but email notifications are async, fire and forget.

For PMs, this means anything that says "processing" or "we'll notify you" is likely async. Know which parts of your flow must be sync (fast) and which are async (need status communication).

Stay Tuned



Up Next

A plain-English guide to cloud, containers, serverless, and the infrastructure decisions that shape your product's cost and speed. Your product runs somewhere — on someone's servers, in some region, inside some container you've never seen.