Module 9: Tech 101 and System Design for PMs

Understand how software works well enough to ask the right questions, make trade-offs with engineers, and explain systems in interviews.

AUTHOR

Varun Kumar

Product Management Master Handbook
v0.1 · 23 October 2026
Built from the 55-lecture course study guide and extended to the 12-module Product Core curriculum.
Hypothetical examples are labelled; no real company results are invented.

Contents

9.1 How the internet works (the 60-second version) 3
9.2 Frontend, backend and client–server architecture 4
9.3 APIs, REST, GraphQL and JSON 5
9.4 Authentication, authorisation and OAuth 8
9.5 Databases, SQL vs NoSQL, and data modelling 9
9.6 Cloud computing: AWS, Azure and GCP 11
9.7 Monolith vs microservices 12
9.8 Caching, load balancing, queues, event-driven systems, webhooks and CDNs 12
9.9 Scalability, reliability, availability, latency and throughput 13
9.10 Security fundamentals 15
9.11 Privacy and compliance basics 16
9.12 System architecture diagrams 16
9.13 Technical debt and engineering estimation 16
9.14 Agile delivery, CI/CD and feature flags 17
9.15 Observability 17
9.16 Technical trade-offs a PM should recognise 18
9.17 System design examples 18
9.18 Exercises with model answers 22
9.19 Interview perspective 23
9.20 Key takeaways 24
9.21 Check your understanding 25
9.22 Recommended resources 26

MODULE 9

Tech 101 and System Design for PMs

Understand how software works well enough to ask the right questions, make trade-offs with engineers, and explain systems in interviews.

≈ 12 h study

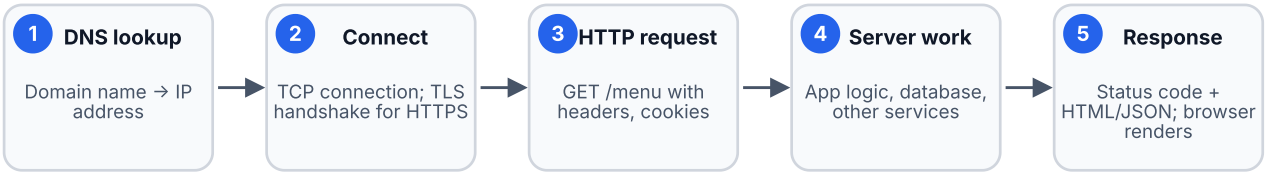
About the architectures

Diagrams are **simplified teaching architectures**, not descriptions of how any named company actually builds its systems. Numbers are illustrative.

Why a PM needs this

You don't write production code, but you decide *what* gets built and you trade off cost, speed, risk and quality with engineers. Knowing the moving parts lets you ask good questions (“what happens if the payment provider times out?”), spot risks early and explain your product credibly in interviews.

9.1 How the internet works (the 60-second version)



What happens when you open a web page. CDNs and caches can answer step 3 without reaching your servers. Each step adds latency and a possible failure point.

- **IP address:** numeric address of a machine. **DNS:** the internet's phone book (name → IP).
- **TCP/IP:** reliable delivery of data packets. **HTTP:** the request/response language of the web. **HTTPS = HTTP over TLS**, which encrypts data in transit and verifies the server's identity.

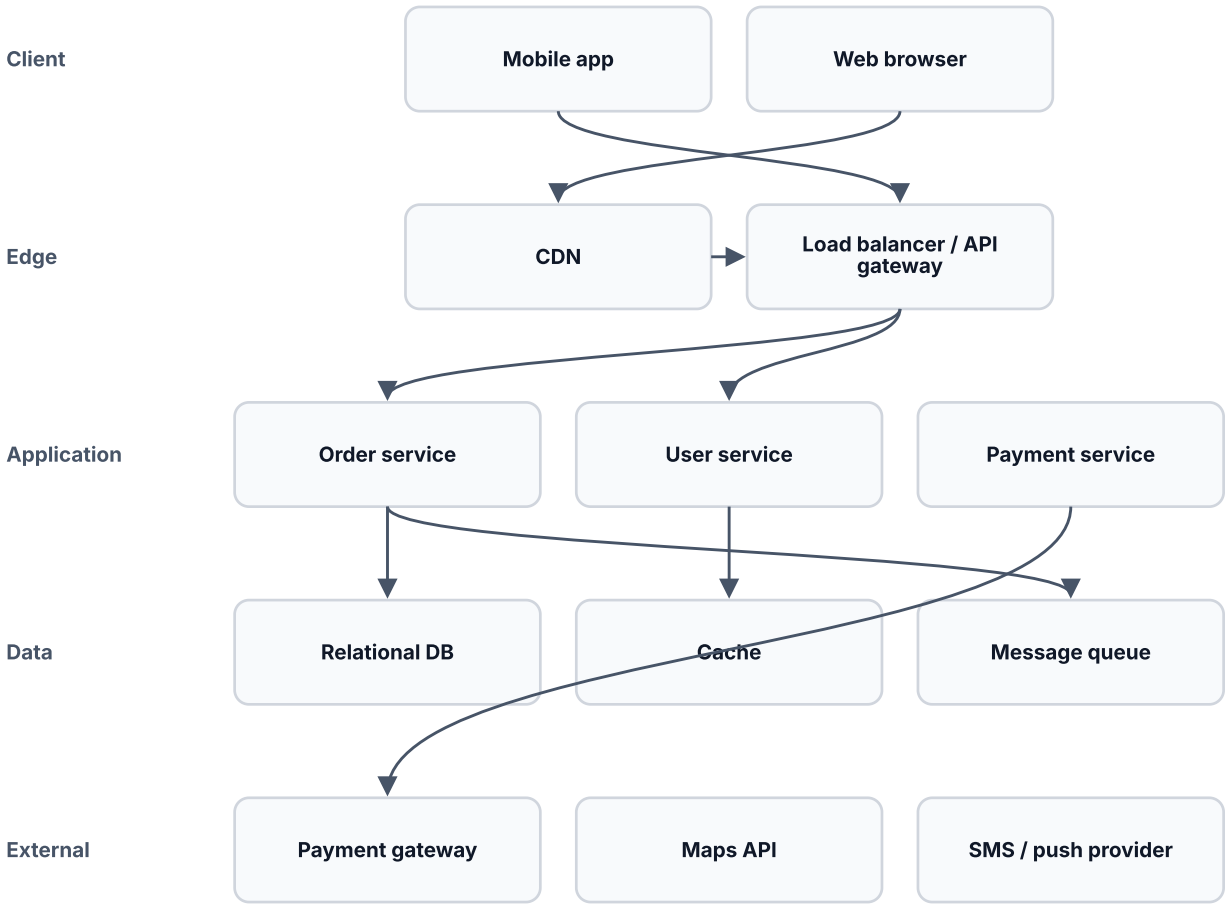
From your course notes (Lecture 33)

HTTPS protects data in transit using TLS. Consider authorisation, sensitive data handling and error recovery as part of any integration.

9.2 Frontend, backend and client–server architecture

From your course notes (Lecture 29; Companion ch. 7)

The **frontend** presents the interface (HTML structures content, CSS styles it, JavaScript adds behaviour; React and Angular are examples of frontend tools). The **backend** applies rules, connects services and stores or fetches data — *it includes application logic, not just databases.* **Databases** store information; **infrastructure** runs the services. Hosting makes the application reachable; a domain is its readable address. **Dev and staging are test environments; production serves real users.** Ask engineers where data is stored, which systems are required, what happens on timeout or retry, how user access is checked, and request explicit release and rollback plans. *A visible button alone is not proof that an order or payment actually happened.*



Client–server architecture of a typical app. Clients send requests; the backend applies business rules; data stores persist state; external providers do specialised jobs. Every arrow is a network call that can be slow or fail.

9.3 APIs, REST, GraphQL and JSON

An **API** (Application Programming Interface) is a contract for how one piece of software asks another for data or an action.

From your course notes (Lecture 33; Companion ch. 8)

An API is a **contract for communication**: a request names an action or asks for data; a response returns information or an error. Agree the **endpoint, method, authentication, expected fields and response format**; different providers may use JSON or XML, requiring mapping and validation. **REST is an architectural style; SOAP is a protocol using XML messages.** A testing tool can send requests and inspect status, data and latency — *validate both expected and failure cases.* **A 200 response alone does not prove the returned data is correct.** An API need not be one-to-many; a website URL and its API endpoint need not match. A **webhook** is an event notification sent to an endpoint instead of the client repeatedly asking.

REST organises an API around *resources* (nouns) with standard HTTP methods; **GraphQL** exposes a typed schema where the client asks for exactly the fields it needs in one query.

```
REST
GET  /restaurants/42/menu            → menu items
POST /orders            {cart...}    → 201 Created, {order_id}
GET  /orders/9001                    → order status

GraphQL (one request, client chooses fields)
query {
  order(id: "9001") { status eta restaurant { name } items { name qty } }
}
```

	REST	GraphQL
Shape	Many endpoints, fixed responses	One endpoint, client-specified fields
Strengths	Simple, cacheable, widely understood	Avoids over/under-fetching; great for complex UIs
Weaknesses	Multiple round-trips; over-fetching	Caching and rate-limiting are harder; query cost control needed
PM question	“Do we need new endpoints for this screen?”	“Does the schema already expose these fields? Any expensive queries?”

JSON (JavaScript Object Notation) is a common text format of key–value pairs and lists:

```
{
  "order_id": "9001",
  "status": "out_for_delivery",
  "eta_minutes": 12,
  "items": [ { "name": "Veg biryani", "qty": 1, "price_inr": 260 } ],
  "fees": { "delivery_inr": 45, "platform_inr": 5, "is_estimate": false }
}
```

9.3.1 HTTP methods and status codes

Method	Typical meaning	Idempotent?
GET	Retrieve	Yes (no side effects)
POST	Create / trigger an action	No (repeating may create duplicates)
PUT	Replace a resource	Yes
PATCH	Partially update	Not necessarily
DELETE	Remove	Yes

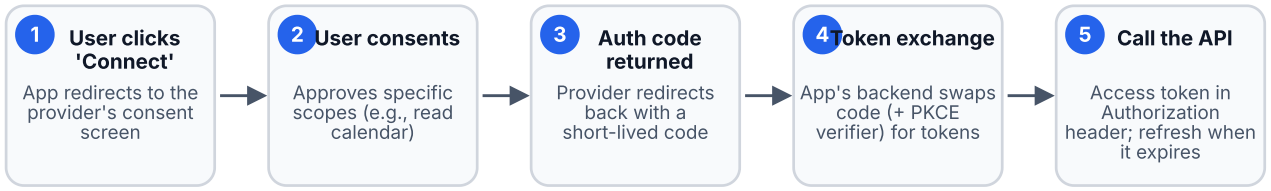
Actual behaviour depends on the API contract. **Idempotent** means doing it twice has the same effect as once — vital for payments (use **idempotency keys** so a retried “charge” doesn't charge twice).

Code	Meaning	PM implication
200 OK / 201 Created / 204 No Content	Success	Still validate the data
301 / 302	Redirect	SEO, deep links
304 Not Modified	Use your cached copy	Performance
400 Bad Request / 422 Unprocessable	Client sent invalid data	Better validation and error copy
401 Unauthorized	Not authenticated (missing/invalid credentials)	Sign-in flow
403 Forbidden	Authenticated but not allowed	Permissions model
404 Not Found	Resource doesn't exist	Empty states, broken links
409 Conflict	State conflict (e.g., duplicate)	Concurrency design
429 Too Many Requests	Rate limit hit	Back-off, quotas, pricing tiers
500 / 502 / 503 / 504	Server, gateway, unavailable, timeout	Retries, fallbacks, status page, incident

---

9.4 Authentication, authorisation and OAuth

- **Authentication (authN):** *who are you?* (password, OTP, passkey, biometrics, SSO).
- **Authorisation (authZ):** *what are you allowed to do?* (roles, permissions, scopes).
- **OAuth 2.0:** a standard for **delegated authorisation** — letting an app access a user's resources at another service without sharing the password, via scoped **access tokens**. **OpenID Connect (OIDC)** adds an identity layer on top of OAuth 2.0 for "Sign in with...".



OAuth 2.0 authorization-code flow (with PKCE), simplified. The app never sees the user's password. Scopes limit what it can do. Tokens are secrets — store them server-side and revoke when users disconnect.

From your course notes (Lectures 46, 49; Companion ch. 8)

API access can use tokens or OAuth, *not necessarily passwords.* **Keep API keys on the server or in approved secret storage — never in a public frontend or screenshot. CORS** is a browser security mechanism for controlling cross-origin requests, *not* a synonym for all communication between a UI and an API.

9.5 Databases, SQL vs NoSQL, and data modelling

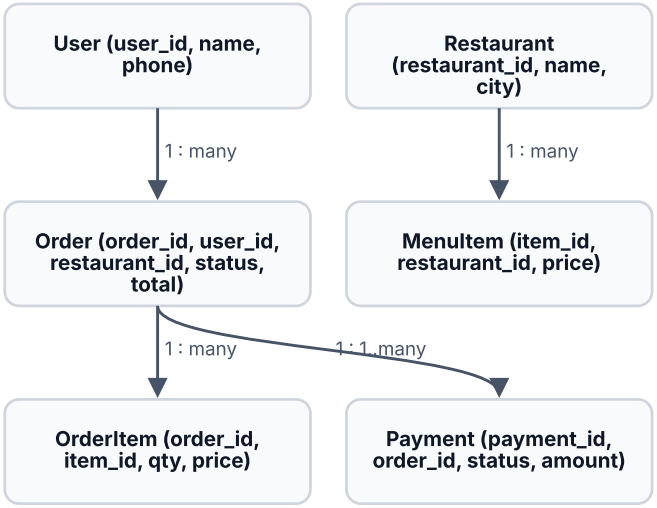
From your course notes (Lectures 29, 34, 35; Companion ch. 9)

Relational, NoSQL and vector databases serve **different needs**; vector indexes store embeddings for similarity search but **do not replace all other databases**. *Data does not automatically move from relational to NoSQL to vector storage as a product grows, and not all AI applications need vector databases.* CRUD (create, read, update, delete) describes common operations; specify **what users can do, who can do it and how information is protected.**

Type	Model	Good for	Examples
Relational (SQL)	Tables with rows/columns, joins, ACID transactions	Orders, payments, accounts — anything needing consistency	PostgreSQL, MySQL, SQL Server
Document	JSON-like documents	Flexible profiles, catalogues	MongoDB, Firestore
Key–value	Key → value	Caching, sessions	Redis, DynamoDB
Wide-column	Rows with flexible columns at huge scale	Time-series, event logs	Cassandra, Bigtable
Graph	Nodes and edges	Social graphs, fraud rings	Neo4j
Search	Inverted index	Full-text search	Elasticsearch, OpenSearch
Vector	Embeddings + similarity index	Semantic search, RAG	pgvector, Pinecone, Weaviate

ACID (Atomicity, Consistency, Isolation, Durability) guarantees matter for money: a payment and its ledger entry must succeed or fail together.

Data modelling defines entities, attributes and relationships — often drawn as an ER diagram:



Simplified data model for food ordering (entity relationships). Storing the item price on OrderItem preserves what the customer actually paid even if the menu price changes later — a classic product-relevant modelling decision.

9.6 Cloud computing: AWS, Azure and GCP

Cloud computing rents computing resources on demand. Service models: **IaaS** (virtual machines, storage), **PaaS** (managed platforms/databases), **SaaS** (finished software), plus **serverless** (run code without managing servers; pay per use).

Capability	AWS	Microsoft Azure	Google Cloud
Virtual machines	EC2	Virtual Machines	Compute Engine
Serverless functions	Lambda	Azure Functions	Cloud Run functions
Object storage	S3	Blob Storage	Cloud Storage
Managed relational DB	RDS / Aurora	Azure SQL Database	Cloud SQL / AlloyDB
Containers / Kubernetes	EKS / ECS	AKS	GKE
Message queue / pub-sub	SQS / SNS	Service Bus	Pub/Sub
CDN	CloudFront	Azure Front Door / CDN	Cloud CDN
Data warehouse	Redshift	Synapse / Fabric	BigQuery

PM questions: What does it cost per user/transaction? Which region holds the data (residency rules)? What are the provider's SLAs? Are we locked into proprietary services? *Product names and offerings change — check current documentation.*

9.7 Monolith vs microservices

	Monolith	Microservices
Structure	One deployable application	Many small services, each owning a capability and its data
Pros	Simple to build, test, deploy early; fewer network calls	Independent scaling and deployment; team autonomy; fault isolation
Cons	Hard to scale teams; one bug can take everything down; slow deploys at size	Distributed-system complexity: network failures, data consistency, observability, ops cost
Good default for	Early products, small teams	Large orgs with clear domain boundaries

Level 3 · Senior PM

Microservices are an *organisational* scaling tool as much as a technical one (Conway's law: systems mirror communication structures). A PM should ask what a migration buys the *customer* (speed of change, reliability) and fund it explicitly — not let it silently consume a year of roadmap.

9.8 Caching, load balancing, queues, event-driven systems, webhooks and CDNs

- **Caching:** keep frequently used data in fast storage (memory) to cut latency and load. Trade-off: **staleness** — when must the cache refresh?
- **Load balancer:** spreads requests across servers; removes unhealthy ones; enables horizontal scaling.
- **Message queue:** producers put work on a queue; consumers process it later — smooths spikes and decouples services (e.g., send confirmation emails asynchronously).
- **Event-driven architecture:** services publish events ("OrderPlaced") that others subscribe to; adds flexibility, complicates tracing.

- **Webhook:** an HTTP callback one system sends to another when an event happens ("payment succeeded"). Receivers must **verify signatures**, handle **duplicates** and **out-of-order** events, and **reconcile** with their own records.
- **CDN (Content Delivery Network):** servers near users cache static content (images, scripts, video) to reduce latency.

Worked example — cache effect on average latency

Cache lookup 5 ms; database 80 ms; cache hit ratio 90%. Average = 0.9 × 5 + 0.1 × (5 + 80) = 4.5 + 8.5 = **13 ms** versus 80 ms without a cache. At a 50% hit ratio: 0.5 × 5 + 0.5 × 85 = **45 ms.** Hit ratio is the lever; staleness is the risk (menus OK to cache for minutes; payment status not).

From your course notes (Companion ch. 8)

A payment provider may send a webhook after payment status changes; **the backend verifies the signature and reconciles the event with its own order record.** Retries must be safe: **use idempotency or deduplication for repeated payment events.** Ask what happens for 4xx client errors, 5xx server errors, rate limits and timeouts.

9.9 Scalability, reliability, availability, latency and throughput

- **Scalability:** ability to handle more load by adding resources — **vertical** (bigger machine) or **horizontal** (more machines).
- **Reliability:** the system does the right thing consistently (correct results, no data loss).
- **Availability:** share of time the system is usable.
- **Latency:** time to respond to one request; report **percentiles** (p50, p95, p99), not just averages.
- **Throughput:** requests (or transactions) processed per second.

```
Allowed downtime = (1 - availability) × period
Availability of components in series = A₁ × A₂ × … × Aₙ
Little's law: concurrent requests in flight (L) = arrival rate (λ) × time in system (W)
```

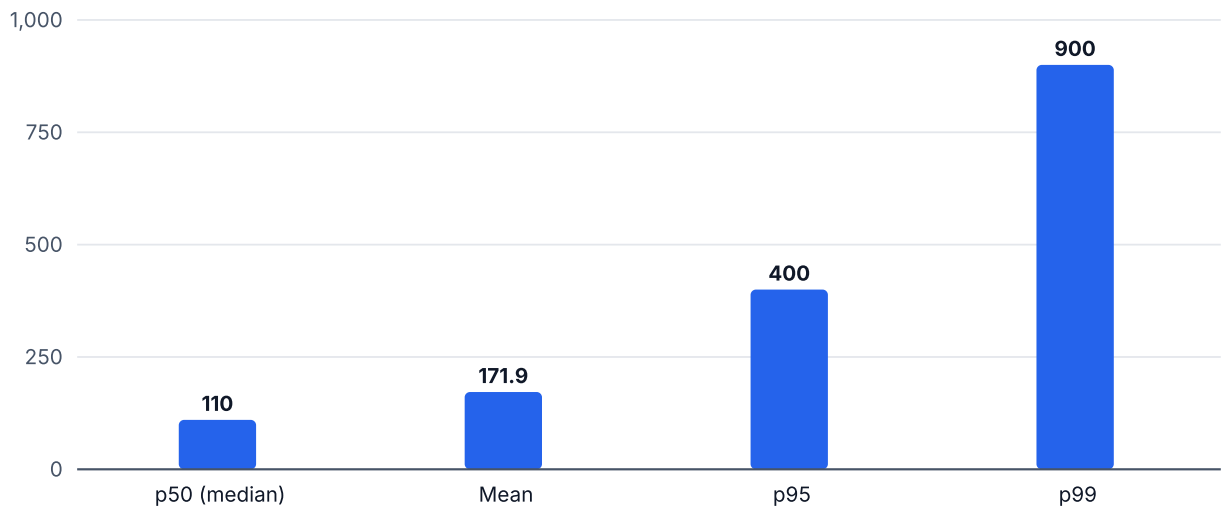
Availability	Downtime per year	Per 30-day month
99%	87.6 hours	7.2 hours
99.5%	43.8 hours	3.6 hours
99.9% (“three nines”)	8.76 hours	43.2 minutes
99.95%	4.38 hours	21.6 minutes
99.99% (“four nines”)	52.6 minutes	4.32 minutes

Worked example — dependencies multiply

Checkout needs the API (99.95%), the database (99.99%) and the payment gateway (99.9%). In series: 0.9995 × 0.9999 × 0.999 ≈ **0.99840 → 99.84%**, about **14 hours of downtime a year** — worse than any single component. Mitigations: redundancy (second gateway), graceful degradation (queue the order; retry), and clear user messaging.

Worked example — why percentiles matter

20 response times (ms), sorted: 80, 85, 90, 92, 95, 98, 100, 102, 105, 110, 115, 120, 125, 130, 140, 150, 180, 220, 400, 900. **Mean** = 3,437 ÷ 20 = **171.9 ms. Median (p50)** ≈ **110–115 ms. p95** (19th value, nearest-rank) = **400 ms. p99** = **900 ms.** The mean hides that 1 in 20 users waits ≥ 400 ms. Set targets on p95/p99 for user-facing flows.



Mean vs percentiles for the same 20 requests (hypothetical). The tail (p95, p99) is what frustrated users experience. Averages are pulled up by outliers yet still understate the worst experiences.

Little's law example: 200 requests/second × 0.25 s average time in system = **50 requests in flight** — useful for sizing connection pools and worker counts.

9.9.1 Rate limits

APIs limit how many requests a client can make (e.g., 100/minute) to protect capacity and enforce fairness or pricing tiers. Exceeding limits returns **429** often with a `Retry-After` header. **PM implications:** quotas per plan, back-off and retry design, batching, and partner contracts (third-party API limits can cap your feature).

9.10 Security fundamentals

- **Principles:** least privilege, defence in depth, secure defaults, encrypt in transit (TLS) and at rest, validate all input, log security events, rotate secrets.
- **OWASP Top 10** (web risks such as broken access control, injection, cryptographic failures, security misconfiguration, vulnerable components) is the standard checklist.
- **Common PM-visible issues:** broken authorisation (user A can see user B's order by changing an ID), exposed API keys in frontends or repos, weak password resets, missing rate limits on OTP/login.
- **Card data:** handling card numbers brings **PCI DSS** obligations; most products use a payment provider's hosted fields/tokenisation to stay out of scope.

> **From your course notes (Companion ch. 24)**
>
> **Never commit passwords or API keys.** If a secret is exposed, tell the responsible team and rotate it; *deleting it from the latest file does not necessarily remove it from history.*

9.11 Privacy and compliance basics

- **Data minimisation, purpose limitation, consent, retention limits, user rights** (access, correction, deletion) are common principles.
- Regimes you may meet: **GDPR** (EU), **India's Digital Personal Data Protection Act, 2023**, **CCPA/CPRA** (California), sector rules (banking/KYC, health). Payment card data: **PCI DSS.**
- PM practice: privacy review for new data uses; no PII in analytics events; data-residency questions in vendor selection; get qualified legal advice for interpretation.

9.12 System architecture diagrams

When drawing a system in an interview or doc: **clients → edge (CDN, gateway) → services → data stores → external providers**, then annotate **data flows**, **sync vs async** calls, **failure points** and **where PII lives.** Label assumptions. Keep it to 6–12 boxes.

9.13 Technical debt and engineering estimation

> **From your course notes (Extra topic 12)**
>
> Technical debt is the future cost of a technical shortcut or outdated implementation; *not every bug or maintenance item is technical debt.* A PM should understand its effect **without pretending to estimate technical work alone.** Record dependencies on the roadmap, track blockers, reserve capacity for reliability where evidence supports it, define release criteria and a rollback path.

Estimation: ask for **ranges with assumptions** ("5–8 days if the pricing API is stable"), break work down, identify unknowns (spikes), and track estimate accuracy over time — not to blame, but to plan. Common traps: anchoring on the PM's wish date; ignoring testing, review and deployment; forgetting non-functional work.

9.14 Agile delivery, CI/CD and feature flags

- **CI (Continuous Integration):** code changes are merged frequently and automatically built and tested.
- **CD (Continuous Delivery/Deployment):** every change that passes tests is releasable (delivery) or automatically released (deployment).
- **Feature flags:** turn features on/off (or for a percentage/segment) without redeploying — enabling staged rollouts, A/B tests and kill switches. Clean up stale flags (they are a form of tech debt).

A CI/CD pipeline. A merged pull request is not necessarily deployed or validated in production. Ask where your change is in this pipeline.

From your course notes (Lecture 29; Companion ch. 24)

An IDE helps developers edit code; repositories support version control; **CI/CD pipelines build, test and deploy across environments.** A repository holds project files and history; a branch isolates work; a commit records a snapshot; a **pull request** proposes changes for review. Learn protected branches, CODEOWNERS, automated tests and CI/CD at a conceptual level.

9.15 Observability

Observability = being able to understand what a system is doing from its outputs: **logs** (events), **metrics** (numbers over time), **traces** (a request's path across services). Reliability targets:

- **SLI** (service level indicator): a measured value, e.g., % of checkout requests succeeding within 2 s.
- **SLO** (objective): the target, e.g., 99.9% over 30 days.
- **SLA** (agreement): a contractual promise with consequences, usually looser than the SLO.

- **Error budget:** 100% − SLO. At 99.9% per 30 days the budget is **43.2 minutes**; if it's spent, prioritise reliability over features.

> **From your course notes (Extra topic 13)**
>
> Define alert thresholds and owners before release; maintain an incident timeline; run a **blameless review** focused on what happened, contributing conditions, impact and prevention; the PM helps quantify user impact and coordinate communication.

9.16 Technical trade-offs a PM should recognise

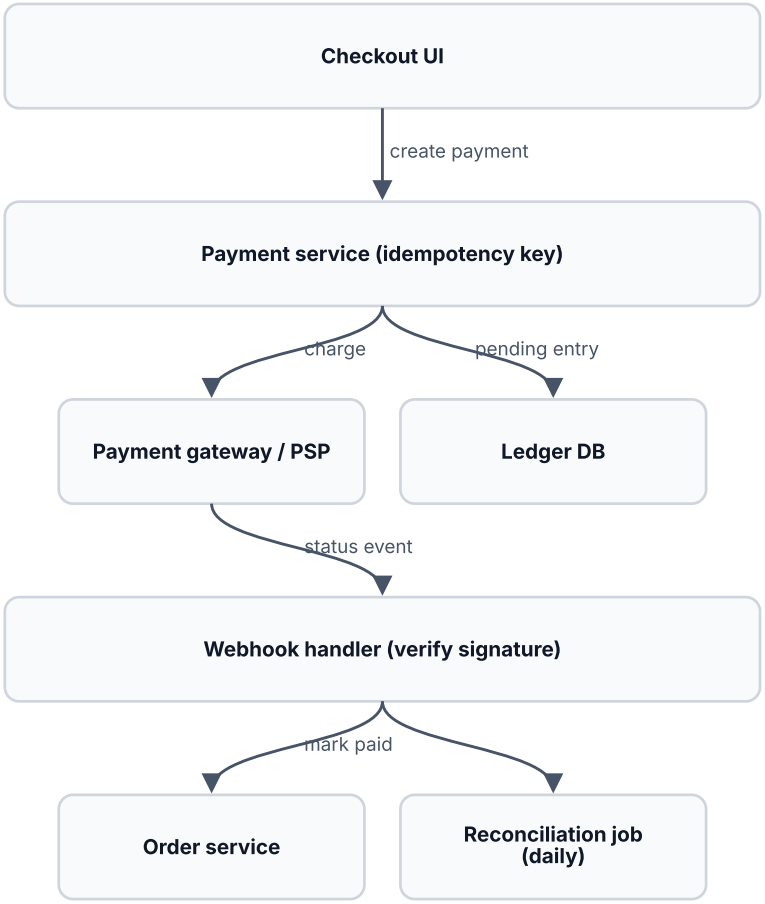
Trade-off	Example decision
Consistency vs availability	During a network partition, show possibly stale menu (available) vs refuse orders (consistent)? (CAP theorem)
Latency vs cost	Add caching/replicas (faster, costlier)
Build vs buy	Own the search engine or use a managed service
Sync vs async	Confirm payment instantly vs queue and notify
Flexibility vs simplicity	Generic rules engine vs hard-coded rules
Speed vs quality	Ship behind a flag with known gaps vs harden first

---

9.17 System design examples

The format for each: **requirements → core components → data flow → key risks → PM decisions/metrics.**

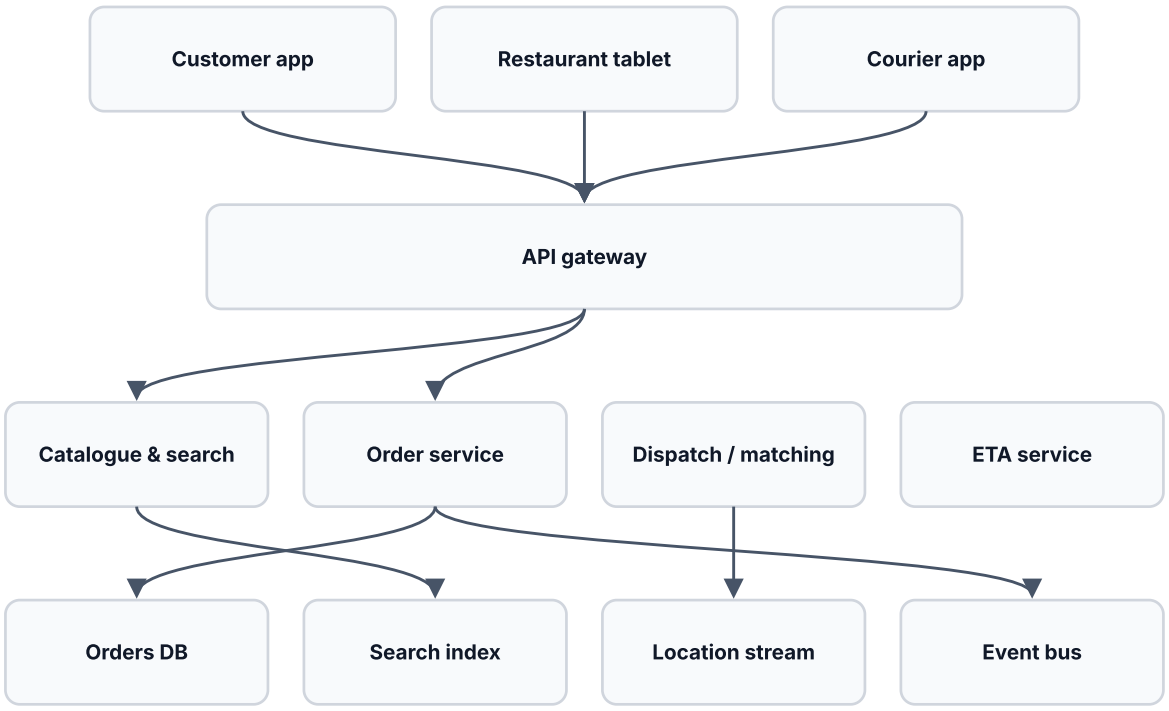
9.17.1 1 · Payment processing



Payment flow with idempotency and reconciliation (simplified). The PSP (payment service provider) holds card data so the app stays out of most PCI scope. Idempotency keys prevent double charges on retries; reconciliation catches missed or duplicate webhooks.

Key risks: double charges on retry; webhook lost/duplicated/out of order; timeouts leaving status unknown (“pending”); fraud; refunds. **PM decisions:** show “payment pending” states honestly; retry UX (another method, saved cart); refund SLA; success-rate SLO by method; reconciliation ownership. **Metrics:** payment success rate (by method/bank), time-to-confirm, duplicate-charge reports, refund time, chargeback rate.

9.17.2 2 · Food delivery



Food-delivery platform (simplified). Three apps, one order lifecycle. Events (OrderPlaced, Accepted, PickedUp, Delivered) drive notifications and analytics.

Key risks: restaurant doesn't accept; ETA errors; courier shortages at peak; menu/price staleness. **PM decisions:** order state machine and customer messaging; batching policy; ETA confidence ranges; cancellation rules. **Metrics:** order acceptance time, on-time rate, ETA error (p50/p90), cancellations by cause, contribution per order.

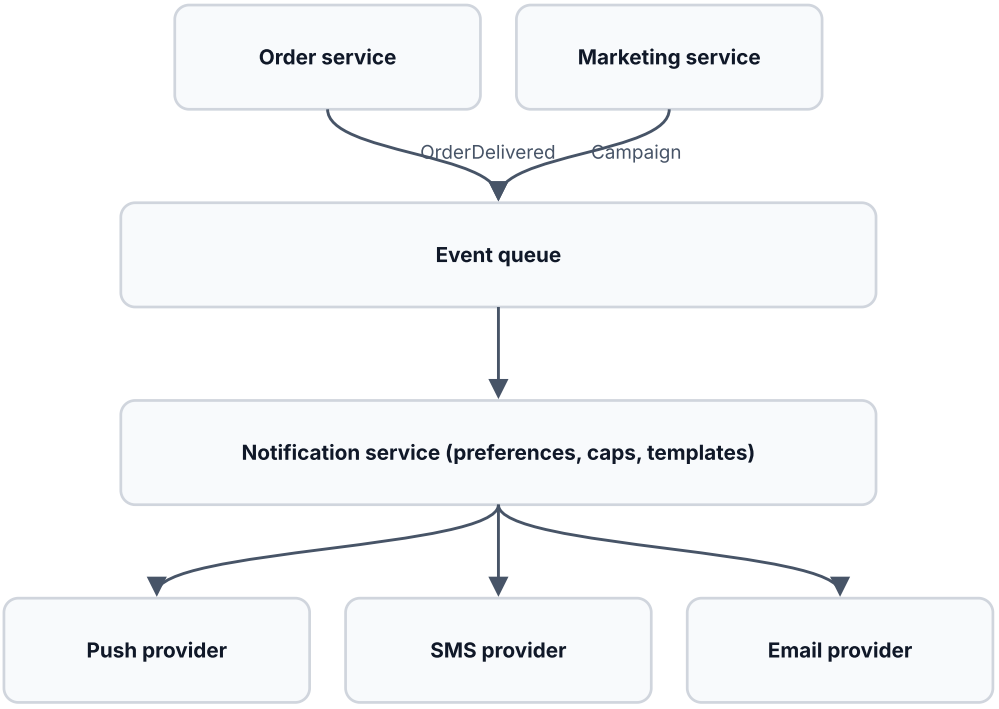
9.17.3 3 · Ride-hailing

Components: rider app, driver app (location every few seconds), **location service** (geospatial index), **matching/dispatch**, **pricing** (base + surge), trip service, payments, safety (SOS, trip sharing), ratings. **Key challenges:** real-time location at scale, fair and fast matching, dynamic pricing with guardrails, fraud, safety. **PM metrics:** ETA to pickup, match rate, cancellation rate (rider and driver), surge frequency, driver utilisation and earnings, safety incidents.

Ride request lifecycle. Every transition has a timeout and a fallback (re-offer to another driver, cancel with explanation). Cancellation reasons are critical product data.

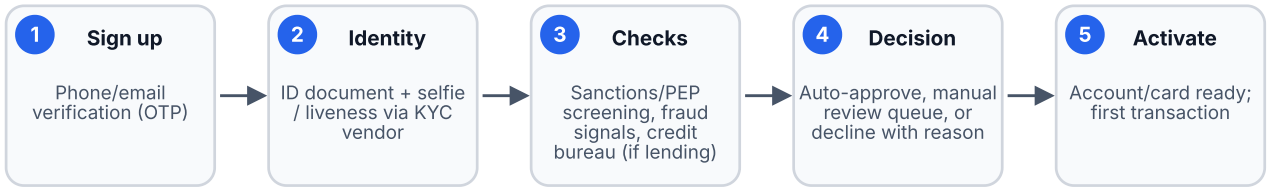
9.17.4 4 · Notifications

Components: event producers → **notification service** (templates, user preferences, rate limits, quiet hours) → channel providers (push, SMS, email, WhatsApp) → delivery receipts. **Key risks:** spam/fatigue, duplicate sends, wrong user, timezone errors, provider outages, consent. **PM decisions:** priority tiers (transactional vs marketing), frequency caps, opt-outs, fallbacks (push → SMS for critical). **Metrics:** delivery rate, open/click, opt-out rate, complaints, time-to-deliver for transactional messages.



Notification pipeline (simplified). Centralising preferences and caps prevents each team from spamming users independently. Transactional messages bypass marketing caps but still respect consent and quiet hours where appropriate.

9.17.5 5 · Customer onboarding (fintech KYC)



Digital onboarding with KYC (simplified). Asynchronous steps (vendor checks, manual review) need clear status pages and notifications. Each step is a drop-off point and a compliance obligation.

Key risks: high drop-off at document upload; vendor latency/outages; false rejections (bias, poor image quality); fraud; data protection. **PM decisions:** progressive onboarding (collect only what is needed now), save-and-resume, camera guidance, manual-review SLAs, adverse-action explanations. **Metrics:** step completion, time to decision, auto-approval rate, manual review backlog, fraud rate, complaints.

9.18 Exercises with model answers

Exercise 1 · Status codes

Users report “Something went wrong” on checkout. Logs show many 429 and some 504 responses from the pricing API. Explain each and propose product actions.

Model answer

429 = rate limit exceeded — our calls to the pricing API exceed the quota (perhaps at peak or due to retries). Actions: cache fee estimates for a short TTL, batch requests, add exponential back-off, negotiate a higher tier. **504** = gateway timeout — the upstream didn't respond in time. Actions: shorter timeout with the “Estimate unavailable” fallback, monitor p95 latency, alerting. Product: replace the generic error with specific, honest copy and a retry.

Exercise 2 · Availability maths

A booking flow depends on four services at 99.9% each. What's the combined availability and yearly downtime? Suggest two mitigations.

Model answer

$0.999^4 \approx$ **0.996 → 99.6%**, about 0.004 × 8,760 ≈ **35 hours/year**. Mitigations: make non-critical dependencies asynchronous (e.g., send confirmation email via queue), add redundancy/fallback for the most critical one, cache read-only data.

Exercise 3 · Explain a system in an interview

In 90 seconds, explain how an order moves through a food-delivery system.

Model answer

"The customer app sends the cart to our API gateway, which routes it to the order service. The order service validates items and prices from the catalogue, asks the payment service to charge via the payment provider using an idempotency key, and saves the order. It publishes an OrderPlaced event; the restaurant tablet gets the order, and the dispatch service matches a courier using live locations and an ETA model. Status events update the customer through the notification service. Key risks are payment failures, restaurant non-acceptance and ETA errors; we track acceptance time, on-time rate and cancellations."

9.19 Interview perspective

"Design a notification system" (technical PM) — what interviewers test and a structure

Tests: requirements thinking, trade-offs, user empathy, scale awareness — not code. *Structure:* clarify use cases (transactional vs marketing), scale (users, messages/day), channels, latency needs → components (producers, queue, notification service with preferences/caps/templates, channel providers, receipts) → data (preferences, logs) → failure handling (retries, dedup, fallbacks) → abuse/spam controls and consent → metrics (delivery, opt-outs, complaints) → phased rollout. *Mistakes:* jumping into tech names without requirements; ignoring user fatigue and consent.

“What happens when you type a URL and press enter?” — 60-second answer

“The browser resolves the domain via DNS to an IP address, opens a TCP connection and a TLS handshake for HTTPS, then sends an HTTP GET request. The request may be served by a CDN cache; otherwise it reaches a load balancer and an application server, which may query databases or other services and return HTML or JSON with a status code. The browser parses and renders the page, fetching CSS, JavaScript and images — often from a CDN. Each hop adds latency and is a potential failure point, which is why we use caching and monitor percentiles.”

9.20 Key takeaways

- Think in **requests and responses**; every network hop adds latency and failure modes.
- Know **HTTP methods/codes**, **authN vs authZ**, **OAuth scopes** and **idempotency**.
- Pick databases by **access pattern and consistency needs**; vector DBs complement, not replace, transactional stores.
- Reliability maths: **dependencies multiply**; report **p95/p99** latency; manage **error budgets**.
- **Caches, queues, webhooks and CDNs** trade freshness, complexity and cost for speed and resilience.
- Security and privacy are product requirements: least privilege, no secrets in clients, no PII in analytics.
- In system-design interviews: **requirements → components → data flow → failures → metrics.**

9.21 Check your understanding

CHECK YOUR UNDERSTANDING

A user is logged in but tries to view another user's order and gets blocked. Which status code fits?

A. 401

B. 403

C. 404

D. 500

Why: 403 Forbidden: authenticated but not authorised. (Some APIs return 404 to avoid revealing existence.)

Why use an idempotency key on payment requests?

A. To encrypt card numbers

B. So retries don't create duplicate charges

C. To speed up the request

D. To bypass rate limits

Why: Idempotency ensures repeating the same request has the same effect as doing it once.

Three services at 99.9% availability in series give roughly...

A. 99.9%

B. 99.7%

C. 99.99%

D. 97%

Why: $0.999^3 \approx 0.997 = 99.7\%$.

Which metric best reflects the slow experiences some users have?

A. Mean latency

B. p95 / p99 latency

C. Minimum latency

D. Number of servers

Why: Tail percentiles capture the worst experiences that averages hide.

OAuth 2.0 is primarily for…

A. Encrypting databases

B. Delegated authorisation via scoped tokens

C. Load balancing

D. Sending emails

Why: OAuth lets an app access resources on a user's behalf without their password; OIDC adds identity.

Feature flags are useful because they…

A. Replace testing

B. Let you turn features on/off or roll out gradually without redeploying

C. Make code faster

D. Remove the need for monitoring

Why: Flags enable staged rollouts, experiments and kill switches.

9.22 Recommended resources

- MDN Web Docs: HTTP — methods, status codes, caching.
- OWASP Top 10.
- OAuth 2.0 (oauth.net) and OpenID Connect.
- Google SRE Book: Service Level Objectives.
- GitHub Docs: Hello World (from your course notes).
- System Design Primer (GitHub).