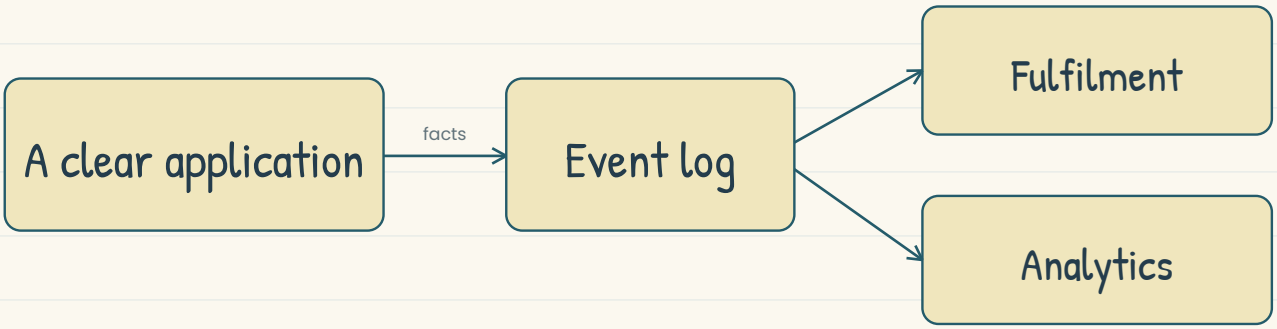
A NOTEBOOK FOR BUILDING SYSTEMS THAT MAKE SENSE

Software Architecture

Monoliths, microservices, Kafka and the decisions between them.

84 pages / 24 chapters / illustrated cases / practical labs



Start with the problem. Draw the boundary. Test the failure.

PRIVATUS COSMAS / PRIVER

depriver.tech / First edition / October 2026

A practical way to learn

This book is for developers who can build a small application and now want to understand how larger systems are organised. You do not need previous Kafka or cloud experience. Basic Python and SQL help with the optional labs.

THREE PAGES PER CHAPTER

First understand the idea and read its diagram. Next study a concrete case and compare the choices. Finally attempt a design task before reading the worked answer. Several designs can be reasonable; the explanation matters.

REAL-WORLD PROBLEMS, FICTIONAL CASES

Soko Market, the school portal, ticketing service and logistics platform are teaching scenarios. They model common engineering problems, but they are not reports of named companies. All workload figures are illustrative unless explicitly described as a lab result.

WHAT IS TESTED

The companion Python/SQLite lab has 16 passing automated tests. It models crash windows, duplicate handling, offset gaps and saga transitions. It is not a running distributed platform. The optional real Kafka exercise is documented separately and was not executed here.

HOW TO READ THE PICTURES

The small diagrams are conceptual views. Arrows show the relationship described by each caption; they are not complete network or deployment specifications. The Kafka and saga chapters include more specific partition, group and failure-branch sketches.

Use the clickable contents to navigate. Source keys link to official documentation or original authors; the complete bibliography is on pages 83-84. Keep a separate notebook for your own decisions.

Your route through the book

Foundations before infrastructure. Decisions before diagrams become deployments.

FOUNDATIONS

01 Architecture starts with a decision 6

02 Quality attributes you can measure 9

03 Draw systems people can understand 12

04 The layered monolith 15

05 The modular monolith 18

06 Ports, adapters and dependency direction 21

SERVICES & EVENTS

07 Choosing microservice boundaries 24

08 Synchronous APIs and contracts 27

09 Data ownership and consistency 30

10 Events, commands and queues 33

11 Kafka: topics, partitions and offsets 36

12 Kafka consumers, groups and replay 39

Glossary 78-79 / Local lab 80 / Kafka lab 81 / Decision template 82 / Sources 83-84

Your route through the book

Foundations before infrastructure. Decisions before diagrams become deployments.

DISTRIBUTED WORK

13 Delivery guarantees and idempotency 42

14 The transactional outbox 45

15 Sagas and business compensation 48

16 CQRS, projections and event sourcing 51

17 Timeouts, retries and overload 54

18 Caching, capacity and scaling 57

OPERATIONS & EVOLUTION

19 Security and tenant isolation 60

20 Observability and operations 63

21 Testing an architecture 66

22 Deployment, containers and serverless 69

23 Migration without a big-bang rewrite 72

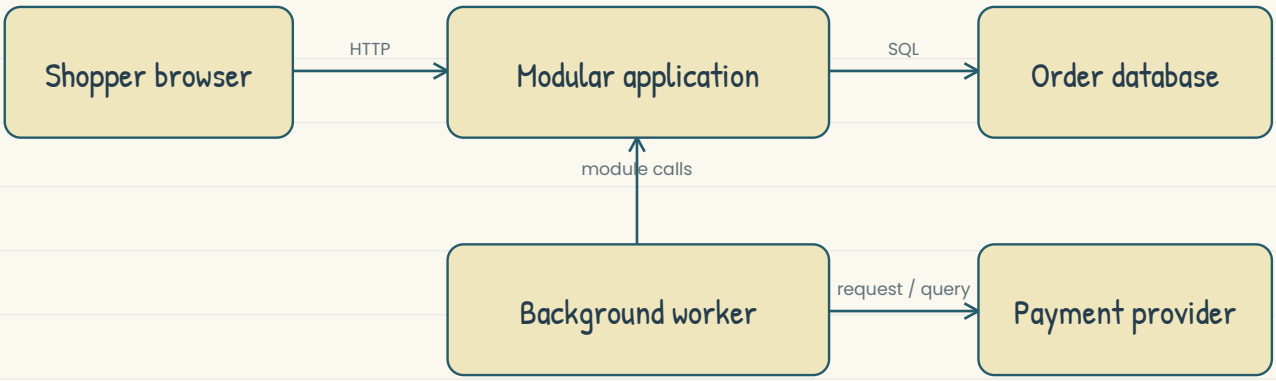
24 Capstone: choose the architecture 75

Glossary 78-79 / Local lab 80 / Kafka lab 81 / Decision template 82 / Sources 83-84

Soko Market: the journey

A shopper buys a router from a marketplace in Tanzania. The app must accept the order, reserve stock, reconcile payment and arrange fulfilment. A mobile connection can fail at any point. Support still needs to explain what happened.

A FIRST-RELEASE RUNTIME VIEW



The provider is external. Its authenticated callback returns to the application through a separate HTTP route, omitted here for clarity. The worker loads durable work; it is not merely an untracked thread.

THE INVARIANTS

Stock never becomes negative. A logical payment attempt must not create unintended duplicate charges. A confirmed order has durable evidence. A timeout preserves uncertainty until reconciliation establishes the result.

THE EVOLUTION

Begin with explicit modules and a relational store. Introduce asynchronous delivery when slow or independent work warrants it. Extract a service when ownership, scaling or release evidence justifies the new network boundary.

The book repeatedly asks: what changes, who owns it, and what happens when it fails?

FOUNDATIONS

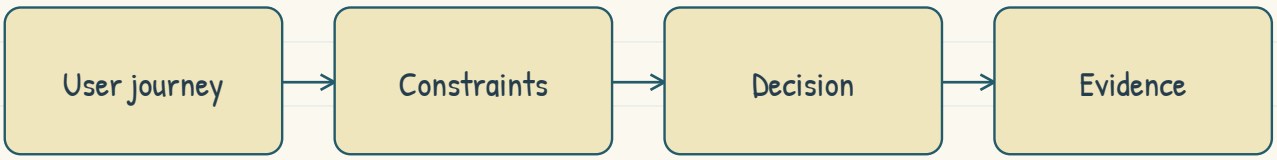
Architecture starts with a decision

Connect a technical choice to a user need, a constraint and a consequence.

Software architecture is the set of important structural decisions that shape how a system changes and operates. It includes responsibilities, dependencies, data ownership and deployment boundaries. A framework name describes only part of that picture. Two applications using FastAPI can have very different architectures.

Begin with a concrete journey. A shopper chooses an item, submits an order and waits for payment confirmation. Ask what must remain true at each step. Stock must not become negative; the same payment attempt must not charge twice; an uncertain payment must not be shown as a confirmed failure.

Constraints narrow the choices. A team of three with one operational owner should value a system it can diagnose and restore. A large budget does not remove coordination costs. Record assumptions, alternatives and the evidence that would cause you to revisit a choice. Architecture is a continuing decision process.



Read left to right: a decision should be traceable to the problem it serves.

The decision in practice

ARCHITECTURE STARTS WITH A DECISION

Soko Market is our fictional Tanzanian marketplace. Three developers maintain a mobile-friendly storefront, stock records and orders. Payment confirmation arrives from an external provider and can be delayed. The first release serves one warehouse. A developer proposes six services because the company hopes to grow. The architect asks which current problem requires six independent deployments. No team needs that independence yet, and local stock/order transactions would become harder.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
One application	Fast local changes	Shared release and runtime
Six services	Independent deployments	Network and operational work
Start modular	Clear boundaries now	Requires dependency discipline

FAILURE TO THINK THROUGH

The wrong optimisation is to design for an imaginary future while missing today's recovery needs. Soko could ship many containers yet still have no tested backup. Count the work required to operate each option, not only the amount of code needed to create it.

READ FURTHER

[C4] C4 model: diagram levels

Make the decision yourself

YOUR TASK

Write a five-line decision record for Soko's first release: context, options, decision, consequence and review trigger. Include one reason your preferred choice could later become unsuitable.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

A defensible decision is one modular application with one relational database and a background worker. Orders, inventory and payments have explicit interfaces. This fits the current team and keeps local invariants manageable. The worker handles slow provider interactions outside the browser request.

The consequence is a shared release and some shared failure exposure. Revisit extraction when a measured bottleneck, ownership boundary or release conflict persists after simpler fixes. A user-count milestone alone is weak evidence: a million mostly cached page views and a thousand contested stock writes stress different resources.

REVIEW YOUR DESIGN

[ ] The decision names an actual constraint.

[ ] At least one rejected option has a fair benefit.

[ ] The review trigger can be observed.

My assumption to test next:

FOUNDATIONS

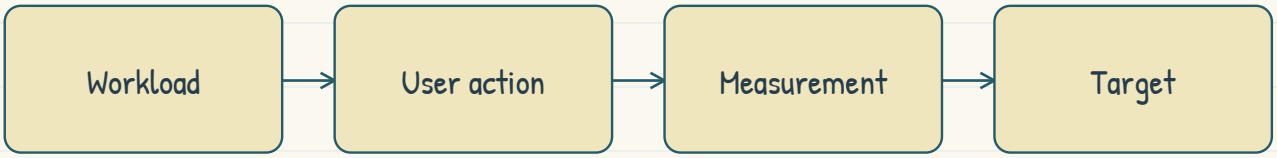
Quality attributes you can measure

Turn words such as fast, reliable and scalable into testable scenarios.

Functional requirements say what the system does. Quality attributes describe how well it does it under specified conditions. Performance, availability, security, changeability and recoverability often compete. A design cannot maximise them all without cost.

Use a scenario: under a stated workload, a named user performs an action, and the system responds within a measurable boundary. Define the measurement point and time window. A server response time is different from the time a shopper waits for payment confirmation.

An SLI is an observed indicator. An SLO is a target for that indicator. An SLA is a separate agreement and may have contractual consequences. A recovery time objective describes acceptable restoration time; a recovery point objective describes acceptable data loss measured in time. A backup schedule alone proves neither.



Every target needs a denominator, a window and a clear measurement location.

The decision in practice

QUALITY ATTRIBUTES YOU CAN MEASURE

Soko expects an illustrative launch peak of 40 checkout attempts per second. The team proposes that 99% of valid order-submission requests return a durable order reference within two seconds over a rolling 30-day window. Provider confirmation is measured separately: showing pending is acceptable when the provider is slow. A response claiming success before the order is durably stored would make the metric look good while breaking the journey.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Fast acknowledgement	User gets a reference	Final state may remain pending
Wait for provider	Immediate final answer sometimes	Long tail and dependency exposure
Separate indicators	Failures are easier to locate	More instrumentation to maintain

FAILURE TO THINK THROUGH

Averages hide painful tails. Ten very slow checkouts can disappear inside a low average. Report a latency distribution and count failures, including rejected overload where your definition requires it. State planned exclusions explicitly instead of silently removing inconvenient requests.

READ FURTHER

[SRE] Google SRE: implementing SLOs

Make the decision yourself

YOUR TASK

For 100,000 eligible requests and a 99.9% success target, calculate the allowed bad requests. Define one separate indicator for delayed payments and one recovery drill.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

The request-based allowance is 100 bad requests: 100,000 multiplied by 0.001. This is not a duration-based downtime allowance. If 70 requests have already failed, 30 remain under that simple window calculation.

For payments, measure the age distribution of pending attempts and the fraction resolved within the chosen business deadline. For recovery, restore a backup into an isolated environment, verify order counts and reconcile provider references. Record actual restore duration and the latest recoverable timestamp. Those observations test the recovery objectives.

REVIEW YOUR DESIGN

[ ] The target specifies eligible traffic.

[ ] Pending payment is distinguished from lost order.

[ ] The recovery objective is exercised, not assumed.

My assumption to test next:

FOUNDATIONS

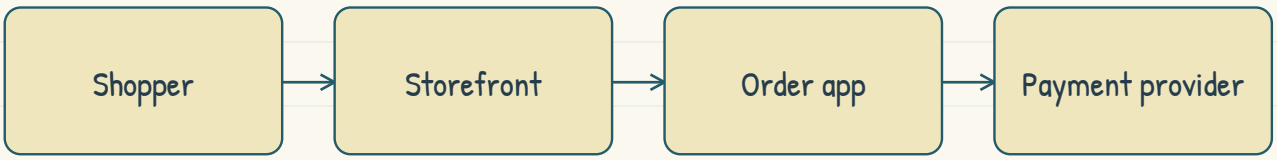
Draw systems people can understand

Use diagrams to explain boundaries, protocols and ownership.

A useful diagram answers a particular question. A context diagram places the system among its users and external dependencies. A container view shows major running applications and data stores. In the C4 vocabulary, a container is a deployable or executable unit, not necessarily a Docker container.

A component view looks inside a unit. A dynamic view shows a sequence of interactions. A deployment view shows where units run. Avoid putting every level into one picture: a user, a Python method and a database replica rarely belong at the same level of abstraction.

Name each arrow with an action or protocol. State who initiates the interaction and whether the caller waits. Mark external systems and trust boundaries. A simple labelled sketch is more informative than a wall of cloud logos. Keep the diagram beside a decision record so changes have a reason.



Context sketch: the provider is outside Soko's control. Detailed callback arrows belong in a sequence view.

The decision in practice

DRAW SYSTEMS PEOPLE CAN UNDERSTAND

A support engineer asks why a shopper has a pending order. A static box labelled Payments does not answer the question. The team draws a sequence: browser submits; app stores pending; worker sends payment request; provider later posts a callback; app records the result. The callback can arrive before the worker receives its original response. Showing this timing exposes the need for stable attempt identifiers and idempotent state transitions.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Context view	Explains scope	Hides runtime detail
Sequence view	Exposes timing and failure	Covers only selected scenarios
Deployment view	Shows failure domains	Can become vendor-specific

FAILURE TO THINK THROUGH

An arrow marked data tells the reader almost nothing. A direct line from browser to database may accidentally imply that clients hold database credentials. Diagrams should reveal security and ownership assumptions rather than conceal them behind ambiguous symbols.

READ FURTHER

[C4] C4 model: diagram levels

Make the decision yourself

YOUR TASK

Draw the pending-payment sequence with six steps. Add a lost-response branch and a duplicate-callback branch. Identify the durable state after each step.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Store an order and a payment-attempt reference before asking the provider to act. If the request times out, keep the attempt unresolved and reconcile using that reference. A callback or a status query may later establish the result.

For a duplicate callback, validate its authenticity and use a unique provider event or attempt identity to recognise work already applied. The final state should not oscillate because an old callback arrives late. Your diagram should name the transition guard and the database transaction that protects it.

REVIEW YOUR DESIGN

[ ] All external actors are labelled.

[ ] Waiting calls and later callbacks are distinguishable.

[ ] A reader can find the durable state after a timeout.

My assumption to test next:

FOUNDATIONS

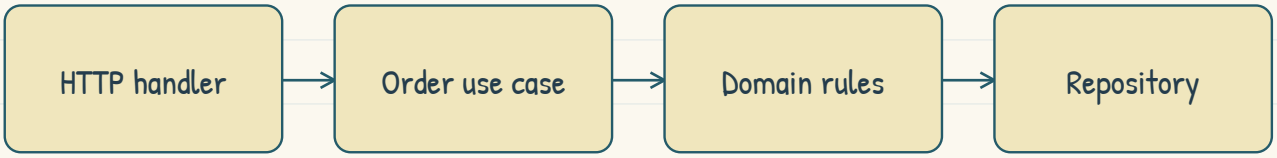
The layered monolith

Understand a single deployable application without equating it with tangled code.

A monolith packages its application capabilities into one deployable unit. It can have many modules, background jobs and database tables. It can also run multiple identical instances behind a load balancer. Monolith does not mean one source file, one machine or an inability to scale.

A layered design separates transport, application workflow, domain rules and persistence. A controller parses a request; a use case coordinates work; domain logic protects business rules; a repository stores data. The exact names matter less than the direction of dependencies and the responsibilities they express.

The convenience is local calls and potentially one local transaction. The risk is accidental coupling: a controller reaches directly into another feature's tables, or shared helpers become a dumping ground. Layering alone does not establish business boundaries. Organise features as well as technical responsibilities.



Call flow descends through responsibilities; domain rules should not require an HTTP request object.

The decision in practice

THE LAYERED MONOLITH

Soko's first checkout stores an order and reserves stock in one relational transaction. The HTTP handler validates the incoming shape, then calls place_order. The use case checks the price snapshot, reserves stock and records a pending payment request. It does not keep the database transaction open while waiting on the payment provider. A worker performs that external work after the local commit.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Single transaction	Protects local invariants	Limited to participating store
One release	Simple deployment path	Features share release risk
Multiple instances	Adds request capacity	Shared database may bottleneck

FAILURE TO THINK THROUGH

A slow provider call inside an open transaction can hold locks and connections. More application instances then increase database pressure instead of helping. Separate durable acceptance from remote completion before treating service extraction as the cure.

READ FURTHER

[MICRO] Microsoft: microservices architecture

Make the decision yourself

YOUR TASK

Split this handler into responsibilities: read JSON, check quantity, compute total, reserve stock, write order, call provider, render response. Which work belongs outside the transaction?

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

The handler reads JSON and maps errors to responses. Domain/application code checks positive quantities, computes authoritative totals and coordinates reservation plus order insertion. The repository executes persistence operations under the use case's transaction boundary.

The provider call runs after commit through a durable job or outbox path. Merely starting an in-memory background task after returning a response can lose work on process restart. The response contains the durable order reference and pending status; it does not promise that the payment has completed.

REVIEW YOUR DESIGN

[ ] Business rules can be tested without HTTP.

[ ] Remote waiting does not hold the local transaction.

[ ] The response reflects committed state.

My assumption to test next:

FOUNDATIONS

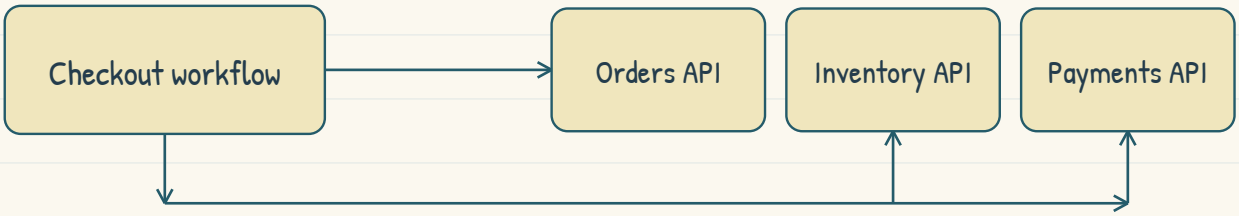
The modular monolith

Keep one deployment while enforcing boundaries around business capabilities.

A modular monolith divides the application into coherent modules with explicit interfaces. Orders owns order state; inventory owns availability and reservations; payments owns payment attempts. Other modules call those interfaces rather than updating private tables or importing internal classes.

Boundaries are enforced through code organisation, dependency checks and review. A folder named services does not enforce anything by itself. Document allowed imports and expose small public interfaces. Keep shared code narrow: identifiers and technical utilities are safer candidates than a giant shared business model.

Modules can share a database engine and still have distinct table ownership. Cross-module transactions are possible in one process, but record where they exist. That coupling affects later extraction. The purpose is understandable change today, not guaranteeing a free migration tomorrow.



Internal APIs are ordinary calls in this design. The diagram does not imply network services.

The decision in practice

THE MODULAR MONOLITH

A developer wants to build a warehouse screen by directly joining and updating orders, stock and payment tables. Reading a reporting projection may be acceptable under a documented contract, but writing payment state from the warehouse module breaks ownership. Soko introduces confirm_dispatch on Orders and reserve on Inventory. Payments exposes payment status and transition operations, not its private persistence model.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Module interfaces	Local, explicit collaboration	Needs design discipline
Shared write access	Fast first shortcut	Hidden ownership and migrations
Boundary checks	Catches illegal imports	Rules need ongoing maintenance

FAILURE TO THINK THROUGH

The distributed monolith starts before distribution: features cannot change without coordinated edits everywhere. Moving those same dependencies behind HTTP adds latency without creating independence. Improve the boundary while calls are still cheap to change.

READ FURTHER

[MICRO] Microsoft: microservices architecture

Make the decision yourself

YOUR TASK

Create an ownership table for orders, stock, reservations and payment_attempts. Name one allowed operation for each and one forbidden cross-module write.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Orders owns orders and exposes place_order and cancel_order. Inventory owns stock and reservations and exposes reserve, release and confirm_allocation. Payments owns payment_attempts and exposes initiate and record_provider_result. A warehouse action must not directly mark a payment as settled.

An order workflow may coordinate the interfaces, but module internals remain private. Add a check that Orders cannot import Payments' repository implementation. Review any transaction spanning module tables: it may be a valid current choice, but it is also evidence to consider before extraction.

REVIEW YOUR DESIGN

[ ] Every writable table has one owning module.

[ ] Public operations use business language.

[ ] A boundary violation is detectable in review or tooling.

My assumption to test next:

FOUNDATIONS

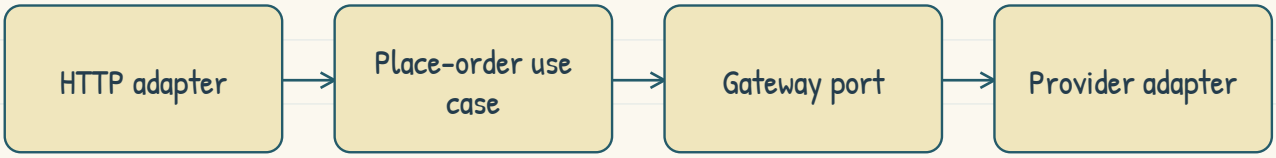
Ports, adapters and dependency direction

Keep business decisions independent of delivery and storage details.

A port is an interface through which application logic communicates with the outside. An adapter implements that interface for a particular technology. A payment gateway port could have a provider adapter for deployment and a deterministic fake for tests. Ports are not TCP ports in this vocabulary.

Hexagonal and clean architecture approaches emphasise dependency direction: policy should not be controlled by replaceable infrastructure. You can apply that idea inside a monolith or a service. It does not require a large framework or an interface for every function.

Test doubles make business scenarios easy to exercise, but they can lie. A fake gateway that always returns immediately misses timeouts, malformed responses and provider-specific limits. Pair fast domain tests with adapter contract tests and a small number of realistic integration checks.



The use case knows the port contract. The adapter knows the provider's protocol.

The decision in practice

PORTS, ADAPTERS AND DEPENDENCY DIRECTION

Soko adds a second payment provider. The application asks a gateway to initiate an attempt using amount, currency and a stable reference. Provider A calls it merchant_reference; provider B uses another field. Adapters translate those details and map responses into a small internal result model: accepted, rejected or uncertain. The use case does not inspect vendor-specific JSON.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Provider port	Separates business policy	Contract must cover uncertainty
Fake adapter	Fast deterministic tests	Can miss integration behaviour
Real adapter tests	Checks protocol mapping	Needs fixtures or sandbox access

FAILURE TO THINK THROUGH

An abstraction that maps every failure to False loses information. A rejected payment can safely be shown as rejected; a timeout may mean the provider accepted the request. Preserve uncertainty so later logic can reconcile rather than create another charge.

READ FURTHER

[HEX] Alistair Cockburn: hexagonal architecture

Make the decision yourself

YOUR TASK

Design a gateway result with three states and a provider reference. Explain which states permit immediate cancellation and which require reconciliation.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Use accepted, rejected and uncertain, with structured reason information and a stable attempt reference. Accepted means the provider accepted the request according to the defined contract; it may still be different from settled. Document that distinction instead of using a vague success flag.

A definitive rejection can move the attempt to failed when the provider contract supports that conclusion. An uncertain attempt stays pending and is queried or resolved by an authenticated callback. The application should not create a new attempt merely because the adapter could not receive the response.

REVIEW YOUR DESIGN

[ ] The core does not import a provider SDK.

[ ] The port preserves meaningful failure states.

[ ] Adapter tests cover malformed and delayed responses.

My assumption to test next:

SERVICES & EVENTS

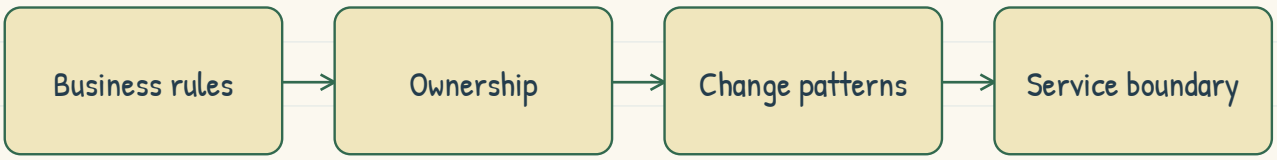
Choosing microservice boundaries

Split where independent ownership and change justify distributed operation.

Microservices organise capabilities as independently deployable services communicating over explicit contracts. Useful boundaries follow business responsibilities and ownership, not a rule that every database table needs a service. A small codebase can still be badly coupled.

Look for cohesive rules and change patterns. Inventory allocation may need different scaling and operational ownership from product descriptions. But splitting orders from order lines usually separates things that must change together. Start with the invariant: which facts must be checked atomically?

Independence has a price. Each service needs deployment, telemetry, access control, incident ownership and recovery. Network calls create partial failure. Teams must evolve contracts without forcing every consumer to upgrade at the same moment. Service count is not a maturity score.



Validate the proposed boundary against real collaboration before creating infrastructure.

The decision in practice

CHOOSING MICROSERVICE BOUNDARIES

Two years into the illustrative Soko journey, warehouse allocation has a dedicated team and release schedule. Its workloads include long-running imports and stock reconciliation. Those jobs repeatedly disrupt storefront releases. Inventory now has a plausible extraction case. Payments might remain a module until its own ownership and operational needs justify moving it. There is no obligation to extract every module together.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Extract inventory	Independent team and workload	Reservation becomes remote
Keep orders together	Preserves order invariants	One release for order features
Service per table	Small code units	Chatty workflows and coupling

FAILURE TO THINK THROUGH

If checkout must synchronously call eight services before accepting an order, its latency and failure exposure include the whole chain. A service map that looks clean organisationally may be unreliable operationally. Trace the critical journey before approving the split.

READ FURTHER

[MICRO] Microsoft: microservices architecture

Make the decision yourself

YOUR TASK

Choose whether orders and order lines should be separate services. Then propose one better boundary and specify the new failure your design introduces.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Keep orders and order lines together when totals, state transitions and line edits belong to one invariant boundary. Separating them adds coordination without a clear independent capability. Inventory is a stronger candidate when a dedicated team owns reservations and stock rules.

The new failure is an unavailable or uncertain reservation response. Define a reservation identity, expiry policy and reconciliation path. Order acceptance may become pending rather than immediately confirmed. That product behaviour must be agreed with support and users, not hidden inside infrastructure work.

REVIEW YOUR DESIGN

[ ] The service has a named operational owner.

[ ] Independent deployment is meaningful.

[ ] Cross-boundary invariants have an explicit strategy.

My assumption to test next:

SERVICES & EVENTS

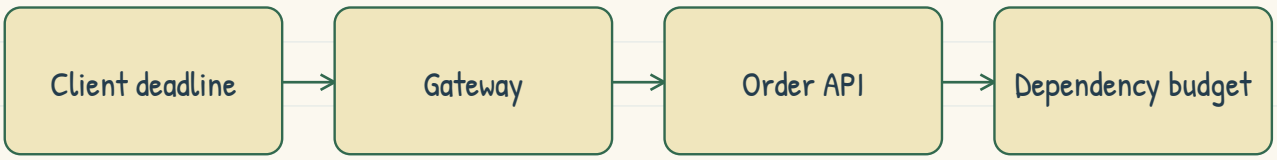
Synchronous APIs and contracts

Design requests that remain understandable when networks are slow or unreliable.

Synchronous communication means the caller waits for a response in the interaction being considered. HTTP and gRPC can both support request-response APIs. Choose according to clients, tooling, contract needs and operating experience rather than assuming one protocol creates better architecture.

A contract includes more than fields: status semantics, timeouts, authentication, pagination, compatibility and retry behaviour matter. Returning 202 Accepted acknowledges processing under the stated contract; it does not mean the business action has finished. Provide a reference or status endpoint for pending work.

Bound latency with a total deadline and per-dependency budgets. Connection and read timeouts may cover different phases; a read timeout alone is not necessarily an end-to-end limit. Propagate cancellation where supported, but do not assume cancellation reverses a remote side effect.



Time spent earlier in the request reduces the budget available to later calls.

The decision in practice

SYNCHRONOUS APIS AND CONTRACTS

Soko's app submits POST /orders with a client-generated idempotency key. The server stores the accepted order and returns an order URL. If the mobile network drops the response, the app retries the same logical request with the same key. A later GET returns the current state. A different basket using the same key is rejected as a conflict instead of silently returning an unrelated order.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Wait for completion	Simple success path	Long waits and uncertain outcomes
Accept then poll	Short durable acknowledgement	Client must handle pending
Webhook update	Less polling	Authentication and retry contract

FAILURE TO THINK THROUGH

Renaming a response field can break deployed mobile apps that cannot upgrade immediately. Even adding an enum value can break exhaustive client logic. Compatibility tests should represent real consumers, not merely prove that the new server accepts its own requests.

READ FURTHER

[HTTP] IETF RFC 9110: HTTP semantics
[RETRY] AWS: timeouts, retries and jitter

Make the decision yourself

YOUR TASK

Write responses for a newly accepted order, a repeated identical request and a repeated key with a different payload. State what the client does after a timeout.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

A new request returns a durable order identity and pending state. The identical retry returns the same logical outcome under the idempotency contract; its status code may follow the documented replay policy. A changed payload with the same key returns a clear conflict.

After a timeout, reuse the key or query by the stored reference. Do not interpret silence as rejection. Keep the key scoped to the caller and operation, store a request fingerprint, and define retention. Expiring the record too early can permit a late duplicate.

REVIEW YOUR DESIGN

[ ] Timeout behaviour is documented.

[ ] Retries preserve logical request identity.

[ ] Compatibility includes errors and enum values.

My assumption to test next:

SERVICES & EVENTS

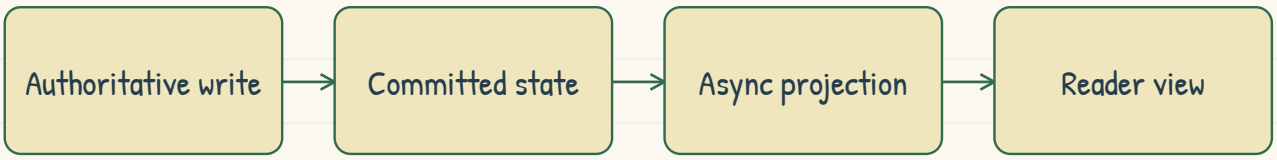
Data ownership and consistency

Locate the source of truth and protect invariants under concurrent work.

A source of truth is the authority for a particular fact, not necessarily one database for the whole company. Inventory may own available stock while Orders owns the customer's order state. Copies for search or reporting do not gain authority just because they are convenient to query.

Local transactions can protect multiple changes within a database. Distributed workflows require additional coordination or a deliberate relaxation of when all views agree. Eventual consistency means replicas or derived views can converge under stated conditions; it is not a promise that every failure fixes itself.

The CAP tradeoff concerns consistency and availability during a network partition under specific definitions. It is not a general instruction to choose any two of three labels for an entire application. Different operations can make different choices. A stale product description may be acceptable while a stock reservation requires an authoritative check.



The reader view can lag. Define when the application must return to the authoritative owner.

The decision in practice

DATA OWNERSHIP AND CONSISTENCY

One unit of a popular router remains in Soko's warehouse. Two shoppers read a cached availability of one. Both attempt checkout. A read-then-write sequence without concurrency control can oversell. Inventory instead performs a conditional update that decrements stock only if enough remains, inside the reservation transaction. Exactly one of the competing reservations can succeed under the database's enforced condition.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Authoritative reservation	Protects stock invariant	Adds latency and contention
Cached availability	Fast browsing	Cannot authorise a reservation
Async read model	Efficient queries	Needs lag and repair handling

FAILURE TO THINK THROUGH

Giving every service write access to the same tables creates hidden coupling even if each runs in a separate container. Database-per-service means logical ownership and access boundaries; it does not necessarily require a separate physical database server for every service.

READ FURTHER

[CAP] Eric Brewer: CAP twelve years later
[SQL] SQLite: isolation

Make the decision yourself

YOUR TASK

For a ticket-booking system, identify which reads may be stale and which write must be exclusive. Describe the outcome when the seat authority cannot be reached.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

A venue description, image or approximate availability count may be stale within an agreed tolerance. Confirming ownership of seat A12 must be protected by the booking authority using a constraint, conditional update or appropriate locking strategy.

If that authority is unreachable, return pending or unavailable rather than inventing confirmation from a cache. This sacrifices immediate completion to protect the seat invariant. A later reconciliation process must resolve pending attempts and release expired holds according to a documented policy.

REVIEW YOUR DESIGN

[ ] Each important fact has an owner.

[ ] Staleness tolerance is specified per operation.

[ ] Concurrent writes cannot bypass the invariant.

My assumption to test next:

SERVICES & EVENTS

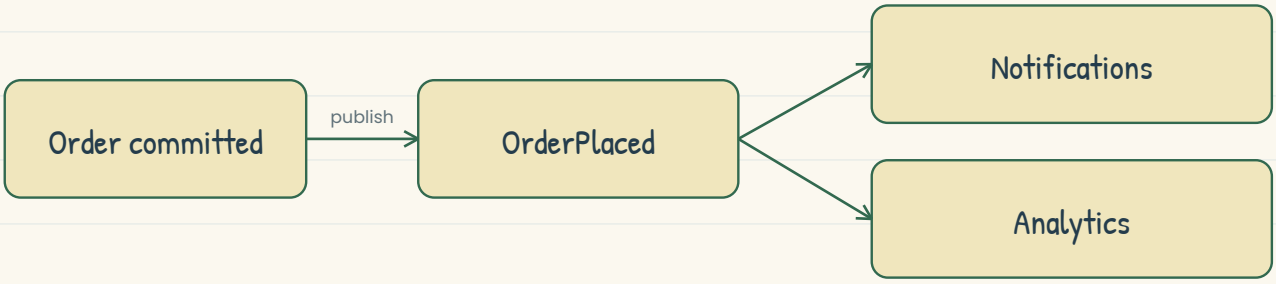
Events, commands and queues

Choose asynchronous messages for a specific coordination problem.

A command asks a responsible receiver to do something: ReserveStock. An event reports a fact that has happened: StockReserved. Both can travel through messaging infrastructure, but their meanings and ownership differ. Name events in the past tense and include enough context to interpret them.

A work queue distributes tasks among workers. Publish-subscribe lets independent subscribers react to the same event. A retained event log supports consumers reading positions over time. Products can support more than one model, so compare required semantics rather than relying only on a product category.

Asynchronous processing decouples timing, not every dependency. Consumers still depend on contracts, identifiers and business meaning. Plan for duplicate messages, delayed processing, invalid payloads and recovery. A message accepted by infrastructure is not proof that every business effect has completed.



Notifications and analytics are independent reactions; a slow report need not block checkout.

The decision in practice

EVENTS, COMMANDS AND QUEUES

Soko wants confirmation emails and daily sales analysis. Neither should delay order acceptance. Orders publishes OrderPlaced after the order is committed. Notifications turns that fact into a delivery task; analytics updates a report. Inventory reservation is a different requirement: if checkout needs an immediate reservation decision, merely broadcasting an event does not make the invariant disappear.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Command queue	Clear work ownership	Receiver and retry policy required
Event subscription	Independent reactions	Contracts and lag still couple systems
Direct API	Immediate decision	Caller waits on receiver health

FAILURE TO THINK THROUGH

An event called ProcessEverything is really an ambiguous command. Consumers guess what must happen, and no owner knows when the workflow is complete. Separate facts from requests and document who is responsible for each state transition.

READ FURTHER

[KINTRO] Apache Kafka 4.3: introduction

Make the decision yourself

YOUR TASK

Classify SendReceipt, PaymentSettled and RebuildSearchIndex. For each, name the owner and decide whether multiple independent subscribers make sense.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

SendReceipt is a command owned by the notification capability. PaymentSettled is a fact owned by Payments and may interest Orders, accounting and analytics. RebuildSearchIndex is an operational command owned by search; broadcasting it as a business fact would be misleading.

An email worker may have multiple competing instances while accounting and Orders each independently receive PaymentSettled. Those are different subscription relationships. Give each logical consumer its own progress tracking, failure policy and replay rules.

REVIEW YOUR DESIGN

[ ] Messages distinguish requests from facts.

[ ] Each required effect has an accountable owner.

[ ] Queue acceptance is not labelled business completion.

My assumption to test next:

SERVICES & EVENTS

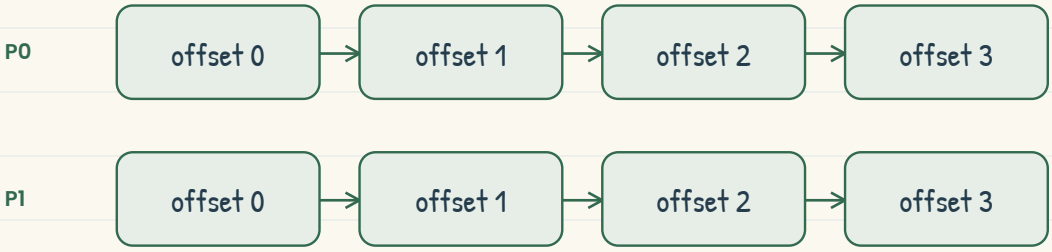
Kafka: topics, partitions and offsets

Understand Kafka as a retained event log rather than a magic service connector.

Kafka stores records in topics divided into partitions. Each partition is an ordered log, and records have positions called offsets. There is no single total order across all partitions in a topic. An offset is a position within one partition, not a global business identifier.

A producer can use a key to route related records consistently when its partitioning strategy and partition count remain suitable. Choosing order_id can preserve per-order locality; choosing customer_id creates a different ordering boundary. Increasing partition count can change key placement, so migration and ordering assumptions need review.

Retention determines how long records remain available; consumption does not inherently delete them. Replication supports durability and availability under configured conditions. Log compaction is a separate retention mechanism centred on keys; it does not mean every historical update is preserved forever.



Ordering is per partition. The key choice should match the entity whose sequence matters.

The decision in practice

KAFKA: TOPICS, PARTITIONS AND OFFSETS

Soko publishes order lifecycle events using order_id. Order O-42 receives OrderPlaced, PaymentSettled and DispatchConfirmed. A consumer can reason about that order's sequence when production, partitioning and processing preserve it. Events for O-43 may be in another partition and processed first. The daily report must not infer global chronology from offsets across partitions.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Key by order	Per-order locality	A hot order can concentrate work
More partitions	More potential parallelism	Ordering and operations need review
Longer retention	More replay history	Higher storage and privacy burden

FAILURE TO THINK THROUGH

A producer acknowledgement is governed by producer and broker configuration; it is not a universal guarantee against all loss. Set replication, acknowledgements and in-sync replica policies together. A single local broker demonstrates APIs but provides no multi-broker resilience.

READ FURTHER

[KDESIGN] Apache Kafka 4.3: design and guarantees
[KINTRO] Apache Kafka 4.3: introduction

Make the decision yourself

YOUR TASK

Given partition 0 offsets 7 and 8 and partition 1 offset 2, identify which ordering claims are valid. Choose a key for delivery-driver location updates and explain the tradeoff.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Within partition 0, offset 7 precedes offset 8 in the log. Offset 2 in partition 1 cannot be compared as a global position. A timestamp is useful context but does not automatically create trustworthy global ordering across producers.

Driver_id is a plausible key when the latest trajectory of each driver matters. It keeps one driver's events together under a stable partition strategy, but a very active driver can create uneven load. For an analytical workload that does not need per-driver sequence, a different distribution may be acceptable.

REVIEW YOUR DESIGN

[ ] The ordering claim names a partition or entity.

[ ] Retention and replay needs are explicit.

[ ] Key choice is reviewed before changing partition count.

My assumption to test next:

SERVICES & EVENTS

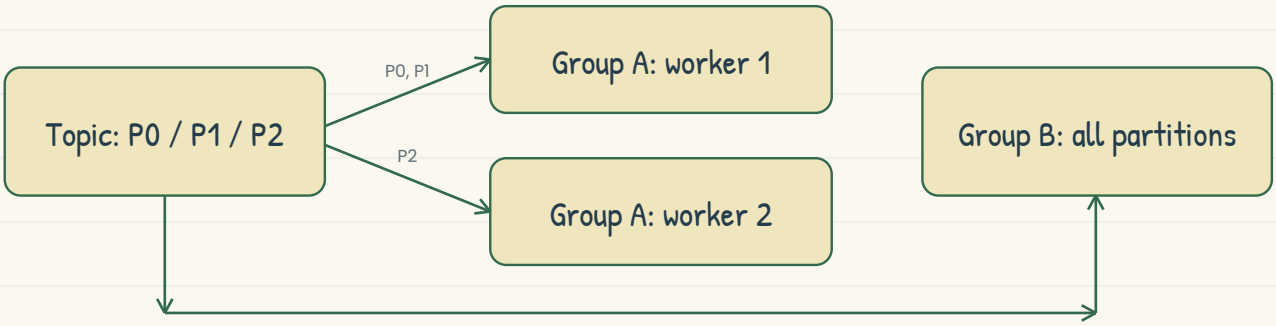
Kafka consumers, groups and replay

Scale consumption while keeping progress and recovery understandable.

In a conventional Kafka consumer group, partitions are assigned among group members; one member at a time owns a partition assignment within that group. Separate groups can independently read the same topic. This chapter covers consumer groups, not the distinct share-group consumption model.

The committed offset records where a group should resume, conventionally the next record to read. Processing progress and committed progress can differ. Rebalances move assignments as membership changes, so consumers need careful shutdown, polling and offset handling.

Replay is powerful only when side effects are controlled. Rebuilding a search index can be safe in an isolated destination. Replaying an email consumer may resend old receipts. Record lag and event age, but remember that an offset gap measures records, not elapsed time or business impact.



A different consumer group has its own progress. Adding workers does not create more partitions.

The decision in practice

KAFKA CONSUMERS, GROUPS AND REPLAY

Soko's orders topic has six partitions. Its search group has three consumers, which can share those assignments. Growing the group to eight conventional consumers leaves at least two without a partition assignment at that moment. A separate analytics group reads all partitions independently. The search team rebuilds a new index under a new group identity, compares results, then switches readers.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
One group, many members	Distributes work	Parallelism bounded by partitions
Separate groups	Independent reactions	Each adds processing and lag
Replay to new destination	Safe comparison and cutover	Needs retained history and capacity

FAILURE TO THINK THROUGH

Committing before durable processing can skip work after a crash. Processing first and crashing before commit can repeat work. A worker that launches untracked parallel tasks can also commit past unfinished records. Completion and offset advancement must be coordinated per partition.

READ FURTHER

[KDESIGN] Apache Kafka 4.3: design and guarantees
[KCONSUMER] Apache Kafka 4.3: consumer configuration

Make the decision yourself

YOUR TASK

A worker processes offsets 10, 11 and 12 concurrently. Only 10 and 12 finish. What is the safe next-offset commit boundary? What would committing 13 risk?

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

The safe contiguous boundary is 11, meaning records below 11 are complete and offset 11 remains to be read. Finishing 12 does not make the gap at 11 disappear. A restart from 11 may repeat 12, so effects need idempotency.

Committing 13 would tell the group to resume after all three records. If the process then dies, unfinished work at 11 may be skipped. Prefer simple sequential processing per partition until the throughput need justifies the complexity of tracking concurrent completions.

REVIEW YOUR DESIGN

[ ] Committed progress cannot pass unfinished work.

[ ] Replay side effects have been reviewed.

[ ] Consumer count is compared with partition count.

My assumption to test next:

DISTRIBUTED WORK

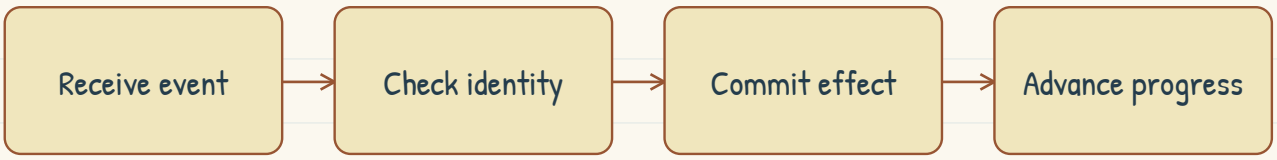
Delivery guarantees and idempotency

Protect business effects when the same request or event appears again.

At-most-once processing can lose work when progress is recorded before the effect. At-least-once processing permits repetition when an effect succeeds but progress is not recorded. These labels describe a defined boundary; they do not automatically describe the outcome in every external system.

Idempotency means repeating a logical operation does not add another unintended effect. A unique operation identity and a durable record of its outcome are common ingredients. The record and the business update must be coordinated; storing a processed flag separately can leave another crash gap.

Kafka producer idempotence and transactions address particular Kafka operations. Transactional read-process-write can coordinate Kafka output and consumer offsets when configured correctly. An email, bank API or unrelated database is not automatically part of that transaction. Define the end-to-end boundary before saying exactly once.



The identity check and durable effect belong together. Progress is acknowledged after that commit.

The decision in practice

DELIVERY GUARANTEES AND IDEMPOTENCY

Soko receives PaymentSettled twice because the publisher retried after losing an acknowledgement. Orders must not dispatch two packages. It records the event identity and the order transition in one database transaction. A repeated identity becomes a no-op. Separately, a dispatch operation has its own stable identity because two distinct events can still refer to the same logical fulfilment action.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Deduplicate event ID	Handles transport repeats	Does not detect all semantic duplicates
Business uniqueness	Protects logical effect	Needs carefully chosen scope
Kafka transaction	Coordinates Kafka work	Does not cover arbitrary external effects

FAILURE TO THINK THROUGH

A set of processed IDs in process memory disappears on restart. A durable set without an atomic effect update is also incomplete. For external side effects, use the provider's idempotency contract where available and reconcile uncertain results instead of pretending a local flag proves completion.

READ FURTHER

[KDESIGN] Apache Kafka 4.3: design and guarantees
[IDEMP] Chris Richardson: idempotent consumer

Make the decision yourself

YOUR TASK

An event changes an order, then the process crashes before acknowledging it. Explain what the next delivery should do. What changes if the effect is an external email send?

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

If the order update and processed identity committed together, redelivery finds the identity and leaves the order unchanged. If the transaction rolled back, redelivery can apply it normally. The companion SQLite lab exercises both cases.

Email sending cannot usually be committed atomically with the local database. Use a durable send task and a provider idempotency facility if offered. Otherwise acknowledge the residual duplicate risk and build delivery tracking. A local database transaction alone cannot guarantee one physical email at the recipient.

REVIEW YOUR DESIGN

[ ] Identity survives process restart.

[ ] The deduplication record and local effect are atomic.

[ ] The guarantee names the systems it covers.

My assumption to test next:

DISTRIBUTED WORK

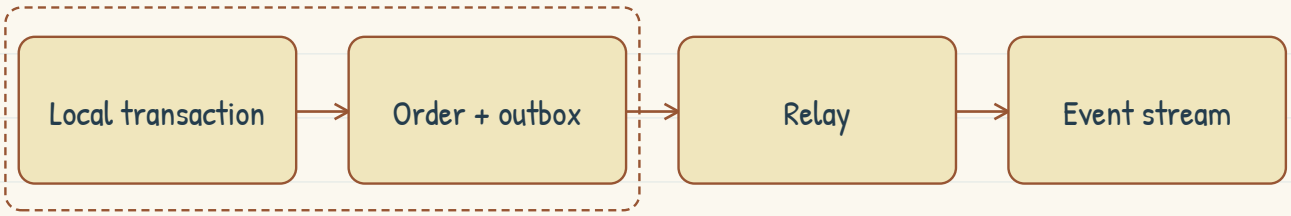
The transactional outbox

Keep a database change and its publication intent together.

The dual-write problem appears when code updates a database and separately publishes a message. Either action can succeed while the other fails. Reversing their order simply changes which inconsistent state is possible.

An outbox stores the business change and a publication record in the same local transaction. A relay later publishes committed outbox records. This makes publication retryable from durable state. The relay may publish more than once if it crashes after publishing but before recording completion.

An outbox therefore does not remove the need for idempotent consumers. Decide how records are claimed, ordered, retried, monitored and eventually removed. A polling relay and change-data-capture relay have different operational costs. Start with the simplest mechanism that meets the measured workload.



The first two boxes describe one atomic write boundary; the relay is an independent retrying process.

The decision in practice

THE TRANSACTIONAL OUTBOX

Soko commits order O-42 and event E-42 in one transaction. The relay publishes E-42, but loses its connection before it can mark the row sent. On restart it publishes E-42 again. The order is not missing and the publication intent was not lost; the consumer handles the repeated event identity. A dashboard tracks the age of the oldest unpublished row.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Database then publish	Easy to write	Can lose publication intent
Outbox + polling	Durable, inspectable backlog	Polling and cleanup work
Outbox + CDC	Streams committed changes	Connector and schema operations

FAILURE TO THINK THROUGH

A relay that marks a row sent before publishing can lose the event. A relay that never expires or archives completed rows can fill the database. Monitor both pending age and total growth. For multiple relays, use a deliberate claiming protocol rather than letting every worker race over the same rows.

READ FURTHER

[OUTBOX] Chris Richardson: transactional outbox

Make the decision yourself

YOUR TASK

Run the outbox companion lab. Inject a failure after publication but before marking sent. Predict the number of message deliveries and the final order count.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

The simulated broker receives the same event twice after the retry. The consumer's transactional identity check applies the order effect once. The publisher still records the row as sent only after its publish call returns successfully.

The lab models the crash window using SQLite and an in-memory transport; it is not a Kafka broker or a production relay. Its purpose is to make the failure ordering visible. A deployed relay also needs concurrent claims, durable transport acknowledgements, retry limits and operational recovery.

REVIEW YOUR DESIGN

[ ] Business state and publication intent commit together.

[ ] The relay tolerates repeated publication.

[ ] Old pending rows trigger investigation.

My assumption to test next:

DISTRIBUTED WORK

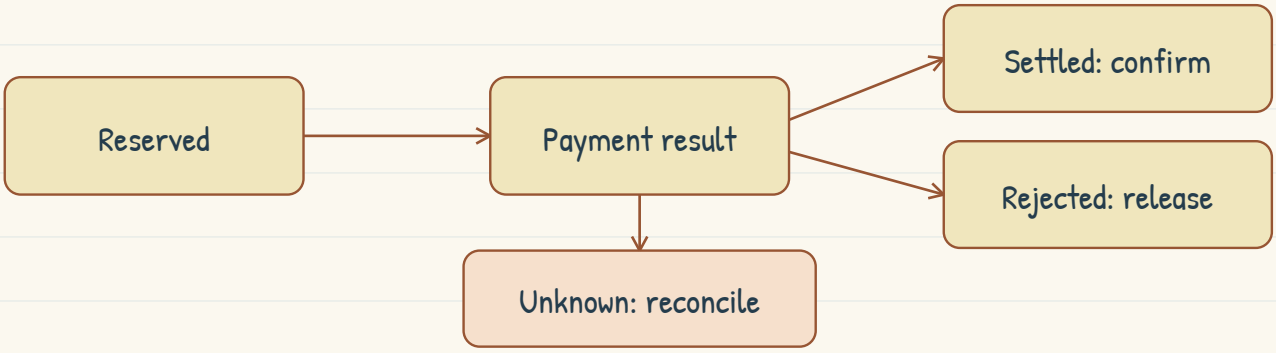
Sagas and business compensation

Coordinate a long workflow without pretending it is one local transaction.

A saga is a sequence of local steps with a policy for failure and compensation. Each service commits its own work. Compensation is a new business action, such as releasing a reservation or initiating a refund; it is not a time machine that erases every consequence.

Orchestration puts workflow coordination in an explicit component. Choreography lets participants react to events. Either can work, but visibility, ownership and complexity differ. A workflow with many timeouts and exceptional paths often benefits from an explicit durable state machine.

Sagas do not automatically provide isolation. Another workflow may observe intermediate state. Use reservations, version checks or other business rules to protect conflicting operations. Compensation itself may fail, so persist its state and provide retry and manual recovery paths.



Success, rejection and uncertainty take different branches. Unknown payment outcomes need reconciliation.

The decision in practice

SAGAS AND BUSINESS COMPENSATION

Soko reserves one router, starts payment and waits. A definitive payment rejection releases the reservation. A timeout is different: the provider might have accepted payment, so the saga records payment_unknown. If a late settlement arrives after a reservation has expired, the business must choose re-reservation, substitution or refund. That decision cannot be delegated to Kafka.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Orchestration	Visible workflow state	Coordinator must be reliable
Choreography	Local reactions	End-to-end flow harder to follow
Local transaction	Strong local boundary	Cannot include every remote effect

FAILURE TO THINK THROUGH

Automatically refunding on every timeout can refund an unknown charge, race with late callbacks or create inconsistent support messages. Write transitions for definitive rejection, uncertainty, late success and failed compensation separately. Include an operator-visible terminal exception state.

READ FURTHER

[SAGA] Microsoft: saga pattern

Make the decision yourself

YOUR TASK

Design states for reserved, payment_unknown, confirmed, release_pending and cancelled. Which event permits confirmed? What happens if release fails?

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

A verified settlement plus a valid reservation permits confirmed under this example's policy. A timeout moves to payment_unknown, not cancelled. A definitive rejection requests release and enters release_pending. Successful release allows cancelled.

If release fails, retain release_pending and retry the same logical release identity. Escalate when age exceeds the operational threshold. The companion state-machine lab checks that uncertain payment does not release stock and that repeated confirmations do not create a second transition.

REVIEW YOUR DESIGN

[ ] Every uncertain outcome has a durable state.

[ ] Compensations can be retried safely.

[ ] Support can locate workflows needing intervention.

My assumption to test next:

DISTRIBUTED WORK

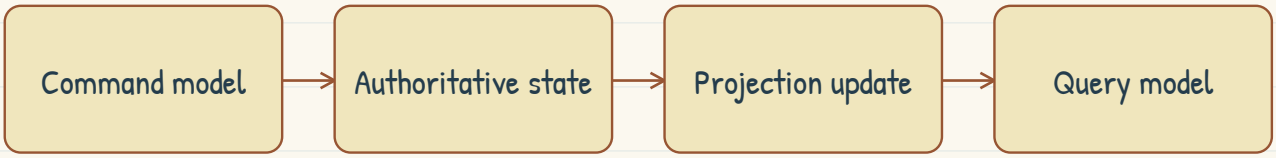
CQRS, projections and event sourcing

Separate three ideas that are often incorrectly treated as one package.

CQRS separates command-side responsibilities from query-side responsibilities. It can begin with different models in one application and one database. It does not require Kafka, microservices or event sourcing.

A projection is a derived view built for a query: orders by day, available products or a customer timeline. It may lag behind authoritative writes. Define its rebuild source and how readers handle that lag. A projection that cannot be repaired becomes another fragile source of truth.

Event sourcing stores the authoritative history of domain changes as events and derives current state from that history. Publishing events from an ordinary CRUD database is not automatically event sourcing. Event sourcing adds demands around event evolution, replay, snapshots and deletion policies; adopt it for a clear domain need.



This diagram shows CQRS with a projection. The authoritative state need not be an event store.

The decision in practice

CQRS, PROJECTIONS AND EVENT SOURCING

Soko needs a dashboard of daily settled totals. Orders and Payments remain ordinary database-backed modules. A reporting projection consumes settlement facts and builds a daily summary. The UI displays the last processed timestamp. Rebuilding into a new table and comparing totals is safer than overwriting the only report in place. Financial reconciliation still checks provider records and authoritative payment state.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
CRUD + queries	Simple operating model	Complex reports can burden writes
Separate projection	Query-specific performance	Lag and rebuild work
Event sourcing	Rich domain history	Long-lived event contracts

FAILURE TO THINK THROUGH

A replay that blindly sums duplicate settlement events inflates revenue. A snapshot without a clear event position can also produce double counting or omissions. Use stable identities, version rules and reconciliation totals; test a rebuild against a known small history.

READ FURTHER

[CQRS] Microsoft: CQRS pattern
[ES] Microsoft: event sourcing

Make the decision yourself

YOUR TASK

For a school portal, decide whether a student timetable needs full event sourcing. Design a projection freshness indicator and a rebuild check.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

A timetable usually needs a current schedule and perhaps an audit trail; those requirements alone do not require full event sourcing. A normal authoritative schedule database with a derived student view can be sufficient.

Show last refreshed time and, where useful, a version identifier. Rebuild a candidate view from a controlled snapshot plus subsequent changes under a consistent boundary. Compare student/course assignments and conflict counts before switching readers. A stale or incomplete view should be visible, not silently treated as authoritative.

REVIEW YOUR DESIGN

[ ] CQRS is distinguished from event sourcing.

[ ] The projection has a reproducible rebuild path.

[ ] Duplicate and late events have defined handling.

My assumption to test next:

DISTRIBUTED WORK

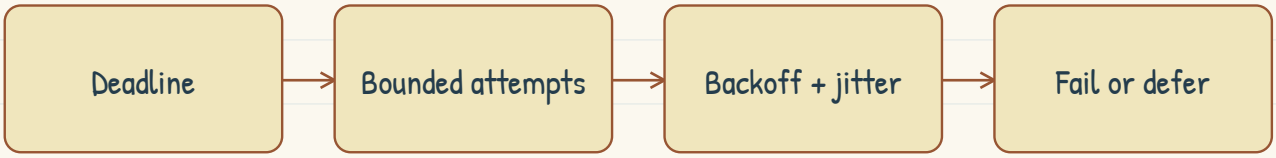
Timeouts, retries and overload

Prevent a small dependency failure from consuming the whole system.

A timeout bounds waiting; a retry repeats an attempt; backoff spaces attempts; jitter spreads them. These mechanisms solve different problems. Retrying validation failures usually wastes work, while retrying an uncertain side effect without idempotency may duplicate it.

A circuit breaker can stop repeated calls to an unhealthy dependency and later allow probes. A bulkhead limits how much capacity one dependency or workload can consume. Backpressure signals that producers must slow down. A queue absorbs bursts only within its capacity and acceptable delay; it cannot create processing capacity.

Retry budgets and deadlines should apply across a user journey. If every layer retries independently, attempts multiply. Decide which layer owns retries and expose the final outcome clearly. Fallbacks must preserve business meaning: cached product text may be fine, fabricated payment success is not.



Every repeated attempt consumes time and capacity. Stop when either budget is exhausted.

The decision in practice

TIMEOUTS, RETRIES AND OVERLOAD

A provider slowdown causes Soko checkout workers to occupy all connections. The team caps concurrent provider calls, separates receipt workers from checkout workers, and uses a bounded retry policy for transient failures. Pending payment attempts remain durable for reconciliation. The storefront can still serve catalog pages while the payment path degrades.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Unlimited retries	May eventually succeed	Can amplify overload indefinitely
Bounded retries	Limits extra work	Some requests need later recovery
Queue + backpressure	Absorbs temporary bursts	Adds delay and capacity limits

FAILURE TO THINK THROUGH

Assume two layers each allow three total attempts. One user action can drive nine downstream calls, not six. If requests already exceed capacity, extra attempts worsen the queue. Measure retry volume separately from original traffic.

READ FURTHER

[RETRY] AWS: timeouts, retries and jitter
[CIRCUIT] Microsoft: circuit breaker

Make the decision yourself

YOUR TASK

A worker completes 80 tasks per second while arrivals remain at 100 per second for five minutes. Estimate backlog growth. Then choose one action that reduces demand and one that increases effective capacity.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

The backlog grows by 20 tasks per second for 300 seconds: 6,000 tasks, ignoring initial backlog and variability. A queue length limit without an overload response only changes where failure occurs.

Demand can be reduced by rejecting or deferring nonessential work and limiting retries. Capacity can improve through a measured bottleneck fix or additional independent workers when the downstream system can sustain them. Adding workers against a saturated database can reduce effective capacity instead.

REVIEW YOUR DESIGN

[ ] Retries are bounded and owned by one layer.

[ ] Side-effect retries preserve identity.

[ ] Overload has an explicit user-visible behaviour.

My assumption to test next:

DISTRIBUTED WORK

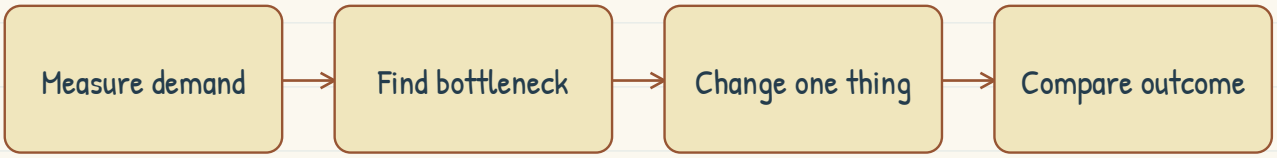
Caching, capacity and scaling

Optimise the resource that measurements identify as limiting useful throughput.

Vertical scaling gives a unit more resources. Horizontal scaling adds units. Neither removes a serial bottleneck or a shared hot record. First identify whether CPU, storage latency, locks, connection limits or an external service constrains the journey.

A cache stores a reusable result closer to the reader. Define keys, expiry, invalidation and acceptable staleness. Include tenant and authorisation scope where relevant. Cache-aside reads the cache first and loads from the source on a miss; simultaneous misses can create a stampede.

Partitioning distributes data or work across units. Replication creates copies for availability or reads. These are different tools. Both introduce coordination and recovery work. Capacity estimates should state units and assumptions, then be checked with representative workloads and headroom.



Scale from evidence. Include cost and tail latency in the comparison, not only average throughput.

The decision in practice

CACHING, CAPACITY AND SCALING

Soko's catalog has 95% cache hits under an illustrative test at 1,000 requests per second. About 50 requests per second reach the source if each miss causes one source read. A full expiry can briefly send far more traffic to the database. The team adds staggered expiry, request coalescing and a stale-read policy for descriptions. Stock reservation still checks the authority.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Cache descriptions	Lower read load	Stale content and invalidation
Read replica	Moves some queries	Replication lag and routing
More app instances	More request concurrency	May overload shared dependencies

FAILURE TO THINK THROUGH

A global cache key such as dashboard can leak one tenant's data to another. A fast response that violates authorisation is not a performance improvement. Key design and access checks belong in the architecture review, not only in an implementation detail.

READ FURTHER

[CACHE] Microsoft: cache-aside

Make the decision yourself

YOUR TASK

Calculate source reads at 80% and 99% hit rates for the same 1,000 requests per second. Name one reason the simple estimate may understate database load.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

At 80% hits, 200 requests per second miss; at 99%, 10 miss. These are request counts, not necessarily SQL query counts. A miss that performs five queries multiplies the database work.

Retries, stampedes, background refresh and write traffic also add load. A benchmark should include realistic hit distribution, hot keys and failure recovery. Measure cache warm-up separately from steady state and verify that stale data never authorises a protected write.

REVIEW YOUR DESIGN

[ ] Units and workload assumptions are stated.

[ ] Cache keys include the relevant security scope.

[ ] The benchmark includes cold or expired cache behaviour.

My assumption to test next:

OPERATIONS & EVOLUTION

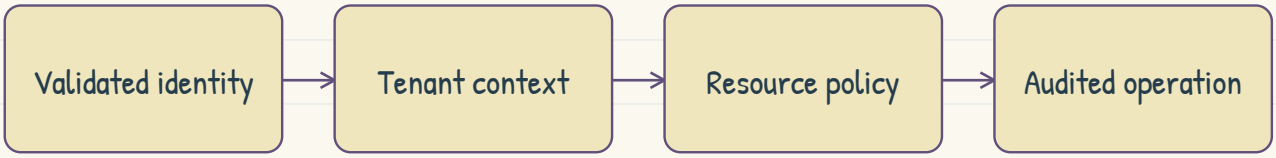
Security and tenant isolation

Make trust boundaries and data access part of the structure of the system.

Authentication establishes an identity; authorisation decides what that identity may do. Encryption protects data in transit or storage under its threat model, but it does not decide whether a user may read another customer's order. Each service must enforce the access rules relevant to its resources.

Use least-privilege service identities and narrow credentials. A gateway can centralise some checks, but internal services should not blindly trust caller-supplied tenant headers. Derive security context from validated identity and preserve it through requests and messages.

Events are durable data distribution. Publishing addresses, tokens or unnecessary personal details spreads the security and retention burden to every consumer and backup. Minimise payloads, classify topics and restrict producer/consumer permissions. Design credential rotation and incident response before an emergency.



Authorisation is evaluated for the resource and action, not merely for successful login.

The decision in practice

SECURITY AND TENANT ISOLATION

Soko later serves multiple merchants. Merchant A requests order 500, which belongs to merchant B. The order API checks both the requested ID and the authenticated merchant scope. The cache key includes that scope. A warehouse event contains only the fields required for fulfilment; analytics receives a narrower representation. Internal network location alone does not grant access.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Gateway checks	Consistent entry policy	Cannot replace resource checks
Service authorisation	Protects owned resources	Requires shared identity contracts
Minimal events	Reduces exposure	Consumers may need authorised lookups

FAILURE TO THINK THROUGH

Filtering only the list endpoint while leaving GET /orders/{id} unscoped leaks data. So can exporting reports through a less-protected worker. Include object-level access tests across HTTP, jobs, files and caches.

READ FURTHER

[OWASP] OWASP: object-level authorisation

Make the decision yourself

YOUR TASK

Design three tests for tenant isolation: direct lookup, cached response and asynchronous export. Explain what must happen when the tenant context is absent.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Each test creates data for A and B, authenticates as A and attempts to access B's result. Direct lookup must not return the record; cache reuse must remain scoped; export generation and download must enforce the same ownership rules.

An absent or invalid tenant context should fail closed for tenant-protected operations. Do not substitute an all-tenants default. Logs can record a safe request identifier and denial reason without exposing the protected record or credentials.

REVIEW YOUR DESIGN

[ ] Tenant context comes from validated identity.

[ ] Object-level authorisation covers all delivery paths.

[ ] Messages contain only necessary data.

My assumption to test next:

OPERATIONS & EVOLUTION

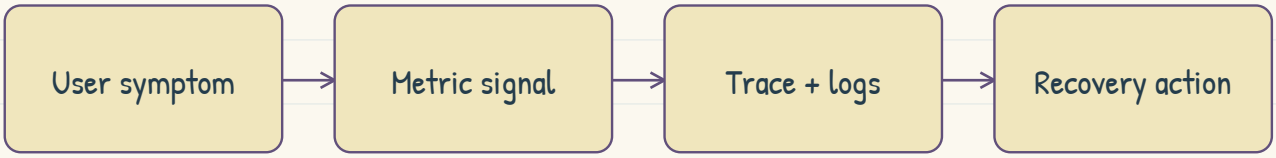
Observability and operations

Make it possible to explain a user-visible failure across boundaries.

Metrics summarise behaviour over time; logs record discrete details; traces connect operations across a request or workflow. They complement each other. A trace does not replace business reconciliation, and a dashboard of CPU graphs does not explain why a customer has no order confirmation.

Propagate correlation identifiers across requests and events, while avoiding secrets in telemetry. Use bounded labels in metrics: putting every order ID into a metric label creates high cardinality. Put detailed identifiers in appropriately controlled logs or traces instead.

Alert on meaningful symptoms and sustained risk. Queue age, failed transitions and payment uncertainty may matter more than raw queue size. Every alert should have an owner and an initial action. A runbook should state how to diagnose, contain and recover, including how to verify that recovery worked.



Start with the affected journey, then use technical evidence to locate the cause.

The decision in practice

OBSERVABILITY AND OPERATIONS

Soko sees normal web response times but an increase in orders pending for more than ten minutes. A workflow-age metric alerts the on-call engineer. Correlation IDs connect the order, payment attempt and provider response. The cause is an expired provider credential used by the worker. Rotating it restores new work; the team then reconciles the existing pending attempts rather than assuming the backlog fixed itself.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Infrastructure metrics	Show resource health	May miss business failure
Workflow metrics	Show customer impact	Need domain-specific definitions
Distributed traces	Connect dependency calls	Sampling and storage cost

FAILURE TO THINK THROUGH

Logging every request body can expose credentials and personal data while creating expensive noise. Redact sensitive fields, control retention and test redaction. Observability should help operators without creating another data leak.

READ FURTHER

[OTEL] OpenTelemetry: signals
[SRE] Google SRE: implementing SLOs

Make the decision yourself

YOUR TASK

Write an alert and a five-step runbook for growing outbox age. Include a step that distinguishes a blocked publisher from a slow consumer.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Alert when the oldest unsent outbox record exceeds the agreed threshold for a sustained interval. First confirm impact and current age; then inspect relay health and publish errors; check broker reachability and permissions; apply the documented repair; finally verify that unsent age falls and downstream effects reconcile.

A healthy relay with growing consumer lag is a different incident: publication may be succeeding while processing is stalled. Compare outbox progress, broker acknowledgements and consumer event age before changing the wrong component.

REVIEW YOUR DESIGN

[ ] Alerts name a user or workflow impact.

[ ] Correlation survives asynchronous boundaries.

[ ] Recovery includes a correctness check.

My assumption to test next:

OPERATIONS & EVOLUTION

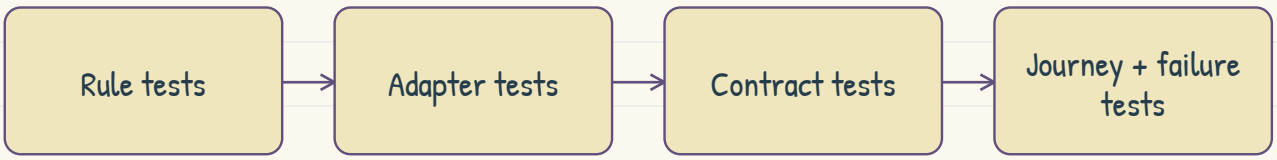
Testing an architecture

Test contracts and failure boundaries, not only individual functions.

Unit tests check local rules quickly. Integration tests exercise real adapters and stores. Contract tests check what consumers expect from an interface. End-to-end tests cover selected journeys through the assembled system. These layers answer different questions; one cannot replace all the others.

Architecture tests can detect forbidden imports, missing ownership rules or undocumented public endpoints. Failure tests explore crashes, timeouts, duplicates and delayed messages. Make the injection point explicit: before commit, after commit, after publication or before acknowledgement produce different outcomes.

A passing local simulation proves only its model. Real infrastructure adds configuration, permissions, concurrency and network behaviour. Keep a small representative integration suite and record environment versions. Test restores and rollback paths as part of release readiness, not only after a production incident.



Test depth grows across boundaries. Use each level for a distinct kind of evidence.

The decision in practice

TESTING AN ARCHITECTURE

Soko's checkout unit tests pass, but a schema change removes currency from a payment message. The local publisher and its tests changed together, so they remain green. A consumer contract fixture from the payment service catches the missing field. Another test replays the same settlement event and verifies that the dispatch count remains one.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Unit tests	Fast rule feedback	Cannot validate broker configuration
Contract tests	Protect independent consumers	Need representative contracts
Failure injection	Exercises recovery assumptions	Must control scope and cleanup

FAILURE TO THINK THROUGH

An end-to-end happy-path test can pass while every recovery path is broken. Conversely, thousands of unit tests can coexist with a misconfigured network policy. Map each major architectural claim to the type of test that can actually falsify it.

READ FURTHER

[OUTBOX] Chris Richardson: transactional outbox
[KDESIGN] Apache Kafka 4.3: design and guarantees

Make the decision yourself

YOUR TASK

Create a test matrix for order creation and event publication with crashes before commit, after commit and after publish. State the invariant for every row.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Before commit, neither order nor outbox intent should persist. After commit, both must exist even if the relay has not run. After publish but before marking sent, retry may produce another delivery, but the downstream business effect must remain singular.

The companion tests exercise these boundaries with SQLite and a simulated transport. Extend the matrix in a real deployment with broker outage, consumer restart, database restore and multiple relay workers. Treat those as additional evidence, not as results already established by the local lab.

REVIEW YOUR DESIGN

[ ] Each guarantee has an appropriate test boundary.

[ ] Consumer contracts are represented independently.

[ ] Failure tests check final state, not only exceptions.

My assumption to test next:

OPERATIONS & EVOLUTION

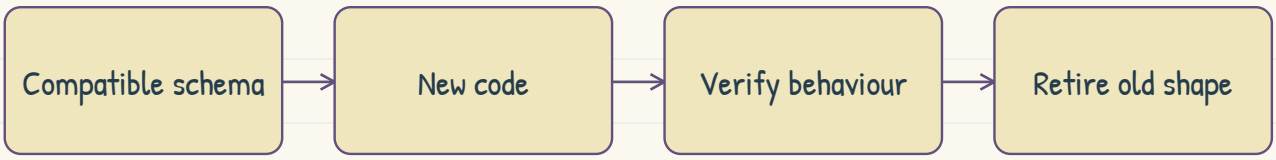
Deployment, containers and serverless

Choose an operating model and release strategy that the team can support.

A container packages an application with a defined runtime environment. An orchestrator schedules and manages running units. Serverless platforms manage more of the execution infrastructure for you. None of these choices determines whether your application is a monolith or microservices.

Choose according to workload duration, scaling pattern, state, networking, cost and operating experience. A managed application platform may be enough for a small team. Kubernetes can offer useful controls, but those controls also require expertise. Serverless execution can simplify bursty tasks while introducing limits and lifecycle assumptions to understand.

Releases change code and often data. Use backward-compatible migrations where possible: expand the schema, deploy compatible code, migrate data, then remove obsolete fields after old readers are gone. Rolling back a binary does not automatically undo a destructive database migration.



Expand and contract separates compatibility changes from irreversible cleanup.

The decision in practice

DEPLOYMENT, CONTAINERS AND SERVERLESS

Soko renames a payment status field. Instead of deleting the old column immediately, the team adds the new representation, supports the transition, backfills data and verifies counts. Only after all readers have moved does it remove the old field. A small canary release compares error rates and pending-age behaviour before broader rollout.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Managed app platform	Lower operational setup	Platform constraints
Containers + orchestration	Flexible runtime control	Cluster and policy operations
Serverless functions	Useful for bursty tasks	Execution and state constraints

FAILURE TO THINK THROUGH

A readiness check that says process alive may send traffic to an application that cannot serve requests. A liveness check tied to a brief database outage can cause a restart storm. Define startup, readiness and liveness according to what each signal controls.

READ FURTHER

[K8S] Kubernetes: startup, readiness and liveness
[SERVERLESS] AWS Lambda: operational practices

Make the decision yourself

YOUR TASK

Plan a release that adds an optional delivery_note field used by one service and ignored by older consumers. Include a rollback and a data verification step.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Add the nullable or otherwise compatible field first, then deploy producers that may send it and consumers that tolerate its absence. Verify old consumers also tolerate unknown fields under their actual schema rules. Compare record counts and representative payloads during the canary.

Rollback restores the prior application version while leaving the compatible schema in place. Do not drop the field as part of an emergency rollback. Remove transitional code only after consumers and retained events have been considered. Compatibility includes data still waiting in queues.

REVIEW YOUR DESIGN

[ ] The platform choice names an operational reason.

[ ] Old and new versions can coexist during rollout.

[ ] Rollback accounts for persisted data and queued events.

My assumption to test next:

OPERATIONS & EVOLUTION

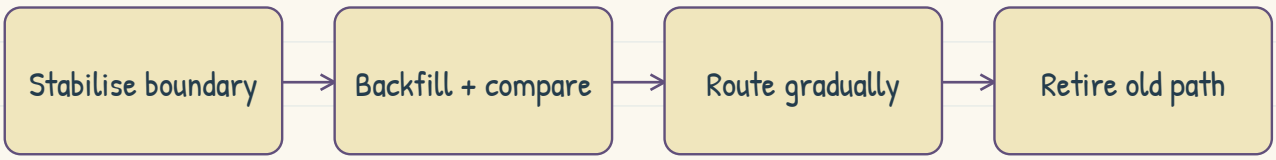
Migration without a big-bang rewrite

Extract one capability with a controlled cutover and a measurable exit criterion.

A strangler migration gradually routes selected behaviour to a new implementation while the old system continues serving the rest. Start with a capability whose boundary is understood. Separate refactoring the boundary from changing its deployment whenever possible.

Data migration needs a source-of-truth policy. Backfills, ongoing changes and cutover can race. Record a consistent starting point and a way to capture later updates. Shadow reads can compare results without serving them, but shadow writes with real side effects are dangerous.

Define success and rollback before traffic moves. A migration is not complete when the new service answers a demo request. It is complete when ownership, monitoring, recovery, data correctness and operational responsibility are established, and the old path can be retired safely.



Each stage has an exit check. Reversibility decreases after data ownership changes.

The decision in practice

MIGRATION WITHOUT A BIG-BANG REWRITE

Soko extracts catalog search first because it is read-heavy and has a rebuildable index. The monolith remains the product authority. The new search service builds an index from a snapshot and subsequent changes, compares result samples, then serves a small traffic share. Checkout still validates price and stock with their owners. Search becoming unavailable must not corrupt orders.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
Read-only extraction	Lower side-effect risk	Does not prove write migration readiness
Gradual traffic shift	Limits exposure	Requires routing and comparison
Big-bang replacement	One visible cutover	Large rollback and discovery risk

FAILURE TO THINK THROUGH

Dual writing to old and new databases without a reconciliation plan creates two competing truths. During a rollback, deciding which one contains the valid recent writes becomes a business problem. Prefer one authoritative write path and explicit propagation while the migration is in progress.

READ FURTHER

[STRANGLER] Microsoft: strangler fig pattern

Make the decision yourself

YOUR TASK

Create a cutover checklist for search. Then explain why the same checklist is insufficient for moving stock reservations.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

Search needs index completeness, query quality checks, freshness bounds, capacity evidence, routing rollback and an owner for repair. If it fails, traffic can return to the old search path while products remain authoritative elsewhere.

Stock reservations also require exclusive write ownership, active-reservation transfer, concurrency guarantees and a policy for in-flight requests. Rolling traffic back while both systems can reserve the same units risks overselling. The migration must coordinate the authority transition, not merely copy rows.

REVIEW YOUR DESIGN

[ ] One authority remains clear during the transition.

[ ] Cutover includes in-flight work.

[ ] The old path has a defined retirement condition.

My assumption to test next:

OPERATIONS & EVOLUTION

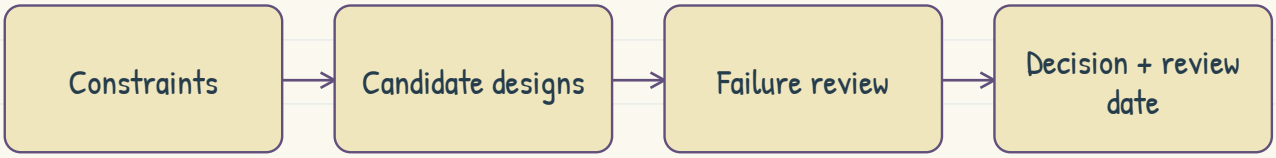
Capstone: choose the architecture

Defend a complete design using workload, risk, ownership and recovery evidence.

Architecture review is a comparison of alternatives under constraints. Explain the user journey, invariants, quality targets, data ownership and deployment approach before listing technologies. A good review names what the system will do when a dependency is unavailable.

Use cost as an engineering input. Count developer effort, operations, managed-service charges, storage, data transfer and incident recovery. A system that is inexpensive per request but requires constant specialist intervention may be a poor fit for a small organisation.

Your final design should contain a context diagram, runtime view, one failure sequence, an ownership table and a decision record. Include a staged evolution path. Separate required mechanisms from optional future improvements so the first release remains achievable.



The deliverable is an argued decision with evidence, not the largest possible box diagram.

The decision in practice

CAPSTONE: CHOOSE THE ARCHITECTURE

Compare three fictional workloads. A school portal has one small team, predictable enrolment peaks and sensitive student records. A ticketing system has short bursts and strict seat uniqueness. A logistics platform receives ongoing location events from many devices and serves independent operational and analytical consumers. All three need reliable software, but their dominant problems differ.

COMPARE THE CHOICES

CHOICE	WHAT IT HELPS	WHAT IT COSTS
School portal	Modular app + jobs fits early needs	Protect tenant data and backups
Ticket booking	Authoritative reservation boundary	Control contention and hold expiry
Logistics stream	Partitioned ingestion may fit	Plan keys, lag, retention and cost

FAILURE TO THINK THROUGH

Using Kafka for a school portal solely because it appears in a modern stack adds operational work without proving value. Conversely, forcing every logistics consumer into one synchronous ingestion request couples device acceptance to unrelated reporting. Match the mechanism to the requirement.

READ FURTHER

[MICRO] Microsoft: microservices architecture
[SRE] Google SRE: implementing SLOs

Make the decision yourself

YOUR TASK

Design Soko's next release for a five-person team, two warehouses and delayed payment callbacks. Submit the five architecture artefacts above. Include a duplicate settlement, a broker outage and a warehouse outage in your review.

WORKED ANSWER / COMPARE AFTER YOUR ATTEMPT

One defensible proposal keeps Orders and Payments in a modular application, gives Inventory a clear interface, and uses durable jobs or an outbox for asynchronous effects. Extract Inventory only if team ownership or measured workload justifies it. Use a broker with retention and independent consumers when those needs outweigh its operational cost; Kafka is an option, not a prerequisite.

A duplicate settlement is absorbed by transactional identity handling. A broker outage leaves committed outbox work pending and visible. A warehouse outage leaves fulfilment pending with an explicit reservation policy. Define who resolves old pending cases, how users see them and what evidence would trigger the next architectural change.

REVIEW YOUR DESIGN

[ ] The simplest viable design is compared fairly.

[ ] Failures have durable states and accountable owners.

[ ] Future extraction has an evidence-based trigger.

My assumption to test next:

The vocabulary, made plain

ADR

Architecture decision record: context, options, decision, consequences and review trigger.

BOUNDED CONTEXT

A boundary within which a domain model and its language have a specific meaning. It is not automatically one service.

INVARIANT

A rule that must remain true, such as a stock balance never becoming negative.

MONOLITH

One application deployment unit; it can contain modules and run as multiple instances.

MODULAR MONOLITH

One deployable application with explicit internal ownership and interface boundaries.

MICROSERVICE

An independently deployable capability with an explicit contract and operational ownership.

PORT / ADAPTER

An application-facing interface and a technology-specific implementation of that interface.

COMMAND / EVENT

A request to perform work / a fact describing work or change that has occurred.

SYNCHRONOUS / ASYNCHRONOUS

The caller waits in the interaction / completion can occur separately and be observed later.

PARTITION / OFFSET

An ordered Kafka log / a position within that partition.

The vocabulary, made plain

CONSUMER GROUP

A conventional group shares partition assignments and tracks progress independently of other groups.

IDEMPOTENCY

Repeating the same logical operation does not add an unintended extra effect.

OUTBOX

Publication intent stored atomically with local business changes, then delivered by a relay.

SAGA / COMPENSATION

A coordinated sequence of local steps / a business action that addresses an earlier step's consequences.

PROJECTION / CQRS

A derived read view / separation of command-side and query-side responsibilities or models.

EVENT SOURCING

Using a history of domain events as authoritative state, rather than only publishing notifications from CRUD state.

BACKPRESSURE / BULKHEAD

A mechanism to slow demand / a capacity boundary that limits the impact of one workload.

SLI / SLO

An observed service indicator / the target set for that indicator over a defined window.

RTO / RPO

Target time to restore service / acceptable data-loss interval. Both need recovery evidence.

RECONCILIATION

Comparing authoritative records and resolving missing, conflicting or uncertain outcomes.

See the failure, then fix it

Extract the companion ZIP and open its folder in a terminal. The local lab needs Python 3.11-compatible syntax and the standard library only; execution was checked on Python 3.14.7. No credentials or network are required.

TERMINAL

```
python architecture_lab.py
python -m unittest test_architecture_lab -v
```

EXPECTED DEMO OUTPUT

```
Deliveries: 2; new fulfilments: 1
Next safe offset: 11
After a payment timeout: payment_unknown
```

FOLLOW THE STATE

place_order commits an order and outbox intent together. relay_once publishes to a simulated transport. Its injected crash leaves the row pending, so retry delivers the same event again. consume commits its identity check and fulfilment effect together.

WHAT TO EXPERIMENT WITH

Open test_architecture_lab.py. Read the producer rollback, publish-crash and consumer rollback tests. Predict the final row counts before running them. Then add a lost-publish-response test and a policy for late settlement after reservation expiry.

THE LIMIT OF THE MODEL

All 16 tests passed locally. The transport is an in-memory list, the relay is single-worker, and saga transitions are pure functions. This does not validate Kafka failover, concurrent relays, network policy, production durability or load handling.

Observe a real event log

Use the official Apache quickstart to start a disposable local broker. The companion kafka-lab.md contains the complete commands. This exercise was checked against documentation but was not executed in the authoring environment.

1 / CREATE A TOPIC

Create architecture-orders-lab with three partitions. The local exercise uses replication factor one; it cannot demonstrate replicated availability. Describe the topic and confirm the partition count.

2 / PRODUCE WITH KEYS

Send O-42:OrderPlaced, O-42:PaymentSettled and O-43:OrderPlaced with key parsing enabled. Observe where the two O-42 records land under the unchanged keyed partitioning setup. Do not substitute Python's built-in hash for Kafka's partitioner.

3 / COMPARE CONSUMERS

Run two conventional consumers in architecture-a and inspect their partition assignments. Start architecture-b to observe independent progress. Print keys, partitions and offsets. Do not assume display order across partitions is a total event order.

4 / REPLAY DELIBERATELY

Use a new group identity for the replay exercise. --from-beginning does not override valid existing committed offsets. Before replaying a real consumer, identify which external effects could repeat and how they are controlled.

READ FURTHER

[KLAB] Apache Kafka: official quickstart
[KCONSUMER] Apache Kafka 4.3: consumer configuration

Record observations separately from expectations. A console demo is evidence about that setup, not proof of an end-to-end production guarantee.

A decision worth keeping

Copy adr-template.md from the practice kit. Use this structure for the capstone or a real project. Keep facts, assumptions and test results distinguishable.

CONTEXT

Who needs what? What workload, team size and constraints are known?

INVARIANTS

Which facts must remain true under concurrency and failure?

ALTERNATIVES

Compare at least two plausible options with fair benefits and costs.

DECISION

Explain why the chosen option fits the present problem.

OWNERSHIP

Name data owners, trust boundaries and operational responsibilities.

FAILURE & RECOVERY

Trace timeout, duplicate, partial commit, dependency loss and restore.

EVIDENCE

Attach test results, measurements and drills. Label untested assumptions.

REVIEW TRIGGER

State the observation that would justify changing the decision.

A useful architecture can be explained, tested and revised.

Read the original sources

Primary documentation and original authors, checked for this edition in October 2026. Links are clickable. Cases, diagrams and worked answers in this book are original teaching material.

[C4] C4 model: diagram levels
https://c4model.com/diagrams

[MICRO] Microsoft: microservices architecture
https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[HEX] Alistair Cockburn: hexagonal architecture
https://alistair.cockburn.us/hexagonal-architecture

[SRE] Google SRE: implementing SLOs
https://sre.google/workbook/implementing-slos/

[HTTP] IETF RFC 9110: HTTP semantics
https://www.rfc-editor.org/rfc/rfc9110.html

[CAP] Eric Brewer: CAP twelve years later
https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

[SQL] SQLite: isolation
https://www.sqlite.org/isolation.html

[KINTRO] Apache Kafka 4.3: introduction
https://kafka.apache.org/43/getting-started/introduction/

[KDESIGN] Apache Kafka 4.3: design and guarantees
https://kafka.apache.org/43/design/design/

[KCONSUMER] Apache Kafka 4.3: consumer configuration
https://kafka.apache.org/43/configuration/consumer-configs/

[KLAB] Apache Kafka: official quickstart
https://kafka.apache.org/quickstart/

[IDEMP] Chris Richardson: idempotent consumer
https://microservices.io/patterns/communication-style/idempotent-consumer.html

Read the original sources

Primary documentation and original authors, checked for this edition in October 2026. Links are clickable. Cases, diagrams and worked answers in this book are original teaching material.

[OUTBOX] Chris Richardson: transactional outbox
https://microservices.io/patterns/data/transactional-outbox.html

[SAGA] Microsoft: saga pattern
https://learn.microsoft.com/en-us/azure/architecture/patterns/saga

[CQRS] Microsoft: CQRS pattern
https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs

[ES] Microsoft: event sourcing
https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

[RETRY] AWS: timeouts, retries and jitter
https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/

[CIRCUIT] Microsoft: circuit breaker
https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

[CACHE] Microsoft: cache-aside
https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

[OWASP] OWASP: object-level authorisation
https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/

[OTEL] OpenTelemetry: signals
https://opentelemetry.io/docs/concepts/signals/

[K8S] Kubernetes: startup, readiness and liveness
https://kubernetes.io/docs/concepts/workloads/pods/probes/

[SERVERLESS] AWS Lambda: operational practices
https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[STRANGLER] Microsoft: strangler fig pattern
https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig