Software & System Architecture Fieldbook

% Architecture Atlas: Software & System Architecture Fieldbook

Independent educational material. Versioned standards and external guidance should be verified at the linked primary sources.

\newpage

Architecture roadmap: from purpose to running evidence

Architecture is the set of important structures and decisions that shape a system. It connects stakeholder outcomes to boundaries, interactions, data, deployment, quality attributes, and evolution. A diagram alone is not an architecture.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Purpose	Defines the outcome and value the system must create	A vague purpose makes every later choice arbitrary
Context	Shows people, external systems, laws, and physical environment	Missing context hides dependencies and trust boundaries
Quality attributes	Make speed, safety, reliability, changeability, and cost measurable	Words such as scalable or secure are not testable
Structures	Describe modules, runtime processes, data, and deployment	One view cannot answer every stakeholder question
Decisions	Record consequential choices and rejected alternatives	A decision without forces and consequences becomes dogma
Evidence	Connects design claims to tests, telemetry, exercises, and review	Architecture drifts when evidence is not automated

How to apply it

Start with a one-page architecture brief: mission, users, business capabilities, constraints, top risks, quality-attribute scenarios, system context, data sensitivity, and explicit non-goals. Generate two or three viable options. Compare them against the same scenarios. Choose the smallest option that satisfies the evidence thresholds. Record the decision and a date or signal that will trigger reconsideration.

Evidence, not opinion

A useful architecture package contains a context view, major runtime and data flows, deployment/failure domains, ADRs, threat model, capacity model, service objectives, ownership map, and executable checks. Review it when the code, topology, risk, regulation, workload, or team boundaries change.

Small example

A ticket marketplace needs buyers to see purchases within two seconds and must never sell one seat twice. Those two scenarios drive consistency boundaries, reservation expiry, idempotency, partition ownership, and operational alarms before any framework is selected.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Software, system, solution, and enterprise architecture

The disciplines overlap but answer different scopes. Software architecture organizes code and runtime elements. System architecture includes software, hardware, networks, people, procedures, facilities, and external environments. Solution architecture shapes one business change. Enterprise architecture aligns portfolios, capabilities, information, technology, and governance.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Software architecture	Structures one software-intensive system	Can optimize locally while harming the wider operating system

Idea	What it solves	Cost, limit, or warning
System architecture	Coordinates technical and human subsystems end to end	Requires lifecycle, physical, safety, and operational thinking
Solution architecture	Integrates products and systems for a bounded outcome	May become a one-off if enterprise reuse is ignored
Enterprise architecture	Guides capabilities, portfolios, standards, and roadmaps	Becomes bureaucracy when detached from delivery evidence
Platform architecture	Creates paved roads and reusable internal capabilities	A platform without product ownership becomes mandatory friction
Systems of systems	Coordinates independently managed systems with emergent behavior	No single team controls all change or failure

How to apply it

Name the entity of interest and its boundary. List what is inside, outside, shared, and independently owned. Identify lifecycle authorities: who funds, builds, deploys, operates, secures, supports, and retires each part. Use capability and context views for wide scope, then zoom into containers, components, data, and deployment only where a decision requires detail.

Evidence, not opinion

Trace every high-level capability to owned services, information, controls, and operational measures. A system boundary is credible only if contracts, responsibility, budget, escalation, and failure handling match it in reality.

Small example

An airport baggage capability includes kiosks, conveyors, scanners, labels, networks, operators, physical zones, airline systems, safety procedures, and recovery teams. Designing only the tracking API misses most of the system architecture.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Systems thinking, boundaries, feedback, and emergence

Systems produce behavior through interactions. Optimizing a component can make the total system worse. Architecture therefore studies feedback loops, delays, constraints, queues, incentives, and emergent effects.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Boundary	Defines what the model includes and excludes	A convenient boundary can hide the actual cause
Feedback loop	Shows how an output changes future input	Delayed feedback causes oscillation and over-correction
Bottleneck	Limits total throughput	Improving non-bottlenecks increases work in progress
Emergence	Names behavior created by interactions rather than one component	Cannot be understood from static parts alone
Leverage point	Finds a small intervention with system-wide effect	High leverage can also create high unintended impact
Requisite variety	Matches control responses to environmental variation	Over-simple governance cannot manage complex exceptions
Socio-technical system	Treats teams, incentives, process, and technology together	Ignoring people moves failure outside the diagram

How to apply it

Draw a causal loop or stock-and-flow sketch for the critical outcome. Add queues, retry loops, cache expiry, batch windows, human approvals, and supplier delays. Ask what happens when demand doubles, one feedback signal is late, or each team locally maximizes its own metric. Move the system boundary until important causes are inside the analysis.

Evidence, not opinion

Use end-to-end lead time, total failure demand, queue depth, rework, and customer outcome—not just component utilization. Run game days or simulations that include people and procedures.

Small example

Autoscaling a slow consumer increases database contention, which increases latency, which triggers more retries, which creates more load. The correct leverage point may be admission control and retry budgets, not more replicas.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Stakeholders, requirements, constraints, assumptions, and risks

Architecture begins by turning conflicting stakeholder concerns into explicit, prioritized forces. Functional requirements describe behavior; quality requirements describe how well; constraints restrict the option space; assumptions need validation; risks combine uncertainty with impact.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Stakeholder map	Identifies affected people and decision authority	Loud stakeholders can hide operators and end users
Functional requirement	Describes observable capability	Feature lists do not expose system qualities
Constraint	Marks a non-negotiable limit	Treating preference as constraint blocks better options
Assumption	Makes uncertain beliefs testable	Untracked assumptions silently become design facts
Risk	Prioritizes uncertainty by likelihood and impact	A risk register without owners and actions is theatre
Non-goal	Prevents accidental scope expansion	Must be revisited when context changes
Acceptance criterion	Defines observable success	Implementation-shaped criteria can prescribe the wrong solution

How to apply it

Interview users, operators, security, data owners, compliance, finance, support, delivery teams, and external partners. Rewrite wishes as scenarios with stimulus, environment, artifact, response, and measure. Separate must, should, could, and will-not. Give every constraint a source and expiry/review condition.

Evidence, not opinion

Maintain a trace from concern to scenario, decision, implementation owner, test, telemetry, and review date. For the top assumptions, run spikes, load tests, supplier checks, or prototypes before committing.

Small example

“Must use Kafka” is usually a proposed solution. “Orders must survive a regional interruption with no duplicate fulfillment and recover within 15 minutes” is an architectural requirement that allows options.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Quality attributes and measurable scenarios

Quality attributes are the bridge between business risk and structure. ISO/IEC 25010:2023 provides a product-quality vocabulary, while practical architecture work expresses priorities as concrete scenarios.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Functional suitability	Checks completeness, correctness, and appropriateness	A correct feature can still be unusable or unsafe
Performance efficiency	Balances latency, throughput, capacity, and resources	Averages hide tail latency and overload
Compatibility	Covers interoperability and coexistence	Shared environments create invisible coupling

Idea	What it solves	Cost, limit, or warning
Interaction capability	Makes user-system interaction effective and inclusive	Accessibility cannot be patched at the end
Reliability	Covers availability, fault tolerance, recoverability, and maturity	Redundancy without independence creates common-mode failure
Security	Protects confidentiality, integrity, authenticity, accountability, and resistance	Controls must match threats and data class
Maintainability	Supports analysis, modification, testing, and reuse	Too many abstractions can reduce changeability
Flexibility	Supports adaptation, scalability, installability, and replaceability	Unused flexibility is permanent complexity
Safety	Avoids unacceptable harm	Safety needs hazards, controls, and assurance evidence

How to apply it

For each priority, write: source of stimulus, stimulus, environment, affected artifact, expected response, and response measure. Resolve conflicts explicitly—for example stronger consistency can increase latency; more isolation can increase cost; more generic extensibility can slow simple changes.

Evidence, not opinion

Use service-level indicators, recovery exercises, capacity tests, accessibility audits, security tests, change lead time, defect escape rate, and cost per business transaction. Measure percentiles and failure modes, not only happy-path averages.

Small example

When a payment provider times out during peak traffic, 99.9% of checkout requests return a stable pending state within two seconds, no charge is duplicated, and reconciliation completes within 30 minutes.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Architecture and design principles

Principles are decision heuristics, not laws. Apply them to the forces in context and record exceptions. A principle that cannot change a decision or be checked is only a slogan.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Separation of concerns	Places independently changing concerns behind boundaries	Excessive separation creates indirection
High cohesion	Keeps related behavior and data together	Cohesion depends on the reason for change
Low coupling	Limits knowledge and change propagation	Some coupling is essential; make it explicit
Information hiding	Hides volatile decisions behind stable interfaces	Leaky abstractions export hidden complexity
Single responsibility	Gives a unit one primary reason to change	Tiny classes/services can fragment a cohesive model
Open/closed and dependency inversion	Allows extension around stable policies	Premature abstraction increases cost
DRY	Maintains one authoritative representation of knowledge	Similar code may represent different domain rules
KISS and YAGNI	Prefer the simplest sufficient design	Simple is measured against required qualities, not line count
Least privilege and secure defaults	Restrict authority and start safe	Operational escape paths still need governance
Fail fast and graceful degradation	Expose invalid state early while preserving bounded service	Fail-fast at the wrong boundary creates outages
Idempotency	Makes repeated intent safe	Requires stable identity and retention policy
Mechanism versus policy	Separates reusable capability from changing rules	Policy still needs clear ownership

How to apply it

Turn principles into local questions: What changes together? Who owns this truth? What authority is needed? What happens on repetition? Which policy will vary? What is the smallest boundary that contains

the invariant? Apply principles at code, service, data, infrastructure, and organization levels.

Evidence, not opinion

Use dependency checks, architecture tests, ownership rules, API compatibility tests, privilege reviews, duplicate-request tests, and change-coupling metrics. Record justified exceptions in ADRs.

Small example

A shared “customer” library violates DRY less than it appears: billing and support may use the same fields but different meanings and lifecycles. Separate domain models preserve knowledge even when code repeats.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Architecture decisions, trade-offs, ADRs, and technical debt

Architecting is deciding under uncertainty. Good records preserve the context, forces, alternatives, decision, consequences, evidence, ownership, and conditions for change.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
ADR	Records one consequential decision and its status	A template does not replace reasoning
Option matrix	Compares alternatives against the same criteria	Weighted scores can hide weak evidence
Reversibility	Distinguishes one-way and two-way doors	Reversible still has migration and coordination cost
Technical debt	Makes a deliberate shortcut and its interest visible	Calling every weakness debt removes priority

Idea	What it solves	Cost, limit, or warning
Prototype/spike	Buys evidence before commitment	A prototype is not automatically production quality
Real-options thinking	Defers irreversible choice while preserving options	Delay has opportunity cost
Decision trigger	Defines when to revisit a choice	Calendar-only reviews miss operational signals

How to apply it

State the decision question before discussing products. List forces and rank them. Generate at least two credible alternatives including “do nothing.” Collect evidence proportional to irreversibility. Choose, record consequences, assign follow-up actions, and publish the ADR beside the system.

Evidence, not opinion

Link ADRs to code, diagrams, controls, tests, dashboards, and migration plans. Track superseded decisions rather than rewriting history. Use drift checks for dependencies, network paths, data ownership, and forbidden coupling.

Small example

Choose synchronous settlement now because the volume is low and correctness is dominant. Trigger a review when p99 exceeds 800 ms, settlement traffic exceeds 200 requests/s, or the provider availability objective falls below the product SLO.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Decomposition, modularity, cohesion, coupling, and boundaries

A useful module owns a coherent capability, protects invariants, exposes an intentional contract, and can change without coordinating with unrelated parts. Distribution does not create modularity.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Functional decomposition	Groups steps or technical functions	Often scatters one business change across layers
Domain decomposition	Groups business language, rules, and data	Needs domain knowledge and careful boundary discovery
Volatility-based decomposition	Isolates things likely to change	Volatility predictions can be wrong
Afferent/efferent coupling	Shows incoming responsibility and outgoing dependency	Counts do not reveal semantic strength
Temporal coupling	Requires operations in an order or time window	Hidden timing creates brittle workflows
Data coupling	Connects modules through shared schema/state	A shared database can be stronger than API coupling
Connascence	Names what multiple parts must agree on	Move strong forms closer and weaken across boundaries
Bounded context	Protects a domain model and language	Not every context deserves a distributed service

How to apply it

Map business capabilities, invariants, transactions, vocabulary, data ownership, change history, and team ownership. Cluster things that change and fail together. Make cross-boundary commands and events explicit. Start with in-process modules unless independent deployment, scaling, isolation, or governance has proven value.

Evidence, not opinion

Analyze repository co-change, cross-team coordination, cyclic dependencies, shared tables, runtime call graphs, incident propagation, and release coupling. A boundary should reduce at least one measurable form of coupling.

Small example

Inventory reservation owns available quantity and reservation expiry. Checkout may request a reservation but cannot update inventory tables. The boundary protects the no-oversell invariant.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Layered, N-tier, client-server, and broker styles

Structural styles arrange responsibilities and allowed dependencies. They are starting shapes, not complete solutions. Logical layers and physical tiers are different decisions.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Layered	Separates presentation, application, domain, and infrastructure concerns	Strict traversal can add pass-through code
N-tier	Deploys processing into separate physical/network tiers	Network hops and partial failure appear
Client-server	Centralizes service behind client requests	Server capacity and availability become critical
Broker	Mediates discovery and interaction between distributed components	The broker adds operational and semantic dependency
Three-tier	Separates UI, application, and data tiers	Does not guarantee domain modularity
Closed layers	Only allow dependency on the next lower layer	Can force irrelevant indirection
Relaxed layers	Permit intentional layer skipping	Without rules, layering erodes

How to apply it

Define each layer’s responsibility and dependency rule. Keep business policy independent of delivery and storage mechanisms where changeability matters. Decide separately whether a logical layer needs a process or network tier. Avoid mapping every folder to a deployable unit.

Evidence, not opinion

Use build-time dependency rules, API tests, latency budgets per hop, network-policy checks, and deployment topology validation. Test a whole use case to expose layers that only forward data.

Small example

A web application can have four logical layers in one process and two physical tiers. That is not the same as four network services.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Hexagonal, clean, onion, ports-and-adapters architecture

These related styles put business policy inside and volatile delivery or infrastructure details outside. Dependencies point toward stable policy; ports express required or offered capabilities; adapters connect technology.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Hexagonal architecture	Separates application core from external actors	Too many generic ports can obscure the domain
Clean architecture	Organizes dependency direction around entities and use cases	Rigid rings can create ceremony
Onion architecture	Places domain model at the center of dependency layers	Does not decide distributed boundaries
Inbound port	Defines an application use case	A CRUD-shaped port can leak persistence thinking
Outbound port	Expresses a capability the core requires	One port per library call over-abstracts stable tools

Idea	What it solves	Cost, limit, or warning
Adapter	Translates HTTP, messaging, persistence, or provider protocols	Adapters must preserve semantics, not only syntax

How to apply it

Model use cases and invariants without framework types. Define ports in domain language. Build adapters for REST, CLI, messages, databases, clocks, identity, and suppliers. Keep transactions and authorization at the boundary that owns the rule. Use dependency inversion only where substitution, isolation, or testing matters.

Evidence, not opinion

Compile the core without infrastructure frameworks. Run the same use-case tests through in-memory and real adapters. Use contract tests to verify semantic equivalence. Check that framework annotations do not leak into central policy unnecessarily.

Small example

A TransferFunds use case depends on AccountRepository, ExchangeRate, Clock, and Audit ports. PostgreSQL, a bank API, and Kafka are adapters; none defines the transfer invariant.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Modular monolith and plugin architecture

A modular monolith keeps strong in-process operational simplicity while enforcing business modules. It is often the best default when distribution has no proven benefit.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Modular monolith	Combines one deployment with explicit modules	Boundaries need automated enforcement
Plugin/microkernel	Extends a stable core through contracts	Versioning and trust of plugins become platform concerns
Package by feature	Keeps a business slice together	Shared technical packages can recreate horizontal coupling
Internal API	Limits what one module can call	Language visibility alone may be insufficient
Module-owned schema	Assigns table ownership inside one database	Database permissions and migration discipline are needed
In-process event	Decouples modules without a broker	Transaction and ordering semantics must be explicit

How to apply it

Define modules from domains and change boundaries. Give each module an API, data ownership, tests, and owner. Ban direct access to internals and foreign tables. Use in-process calls for immediate consistency and events for notification. Extract a service only when evidence shows independent scaling, failure isolation, deployment, or governance value.

Evidence, not opinion

Run architecture tests for module dependencies, migration ownership, cyclic calls, and forbidden imports. Measure build/release lead time, incident blast radius, and cross-module changes.

Small example

A commerce monolith contains Catalog, Pricing, Basket, Ordering, Inventory, and Fulfillment modules. Ordering requests inventory through its API; it never joins inventory tables directly.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Microservices, SOA, service-based, and cell architecture

Service styles distribute capabilities across independently running units. They buy isolation and autonomy by accepting networks, distributed data, observability, deployment, and governance complexity.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Microservices	Enable independently owned, deployable domain services	Distributed transactions and operations become difficult
Service-oriented architecture	Integrates reusable business services across an enterprise	Central governance or ESB can become a bottleneck
Service-based architecture	Uses a smaller number of coarse services	Coarse data ownership can constrain autonomy
Cell/stamp architecture	Replicates isolated service/data slices	Routing, placement, and fleet management grow
Shared-nothing	Minimizes runtime resource sharing	Duplication and eventual reconciliation increase
Service mesh	Standardizes transport security and telemetry	It cannot repair bad service semantics
API gateway	Centralizes edge routing and cross-cutting policy	Business logic at the gateway creates coupling

How to apply it

Split by business capability and invariant, not technical layer. Give services data ownership and clear contracts. Budget every remote dependency. Design identity, authorization, discovery, compatibility, delivery, observability, incident ownership, and reconciliation before multiplying services.

Evidence, not opinion

Prove independent deployment, failure isolation, ownership, SLOs, consumer compatibility, and data reconciliation. Compare the operational cost against a modular monolith baseline.

Small example

A tenant cell contains application, cache, queue, and database partitions for a bounded tenant cohort. A bad deployment or hot tenant affects one cell, while the control plane manages placement and rollout.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Event-driven architecture, messaging, and streaming

Events state that something happened. Commands request intent. Queries ask for information. Messaging decouples time and availability, but creates delivery, ordering, duplication, schema, and observability work.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Publish-subscribe	Fans an event to independent consumers	Consumers can lag or interpret semantics differently
Message queue	Distributes work among competing consumers	Poison messages and fairness need handling
Event notification	Signals change with minimal data	Consumers must fetch state and race with later changes
Event-carried state transfer	Lets consumers maintain local read state	Duplicates data and raises privacy/retention issues
Event streaming	Maintains an ordered log for replay and processing	Partitioning limits global order
Competing consumers	Scales independent work	Ordering per entity requires partition strategy
Dead-letter channel	Quarantines repeatedly failing messages	A DLQ is not a recovery process

How to apply it

Define message intent, owner, schema, identity, partition key, ordering scope, delivery guarantee, retention, privacy, retry, expiry, and recovery. Assume at-least-once delivery and build idempotent consumers unless stronger semantics are proven end to end. Version for semantic compatibility.

Evidence, not opinion

Track consumer lag, age of oldest message, duplicates, handler latency, retry/DLQ volume, schema compatibility, replay outcome, and business reconciliation.

Small example

OrderPlaced contains stable order facts and version. Inventory reserves idempotently using event ID and order ID. A duplicate delivery changes nothing; a late incompatible version is quarantined and alerted.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

CQRS, event sourcing, saga, and transactional messaging

These patterns address different problems and are often confused. CQRS separates write and read models; event sourcing stores state transitions; a saga coordinates distributed business work; outbox/inbox bridge database and messaging.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
CQRS	Optimizes command and query models independently	Creates synchronization and consistency lag
Event sourcing	Persists an append-only sequence of domain facts	Schema evolution, privacy, replay, and debugging are hard

Idea	What it solves	Cost, limit, or warning
Saga choreography	Coordinates through events without a central controller	Flow and failure policy become hard to see
Saga orchestration	Centralizes process state and compensation decisions	The orchestrator becomes a critical domain component
Transactional outbox	Atomically stores state and an outgoing message	Relay lag and cleanup need operations
Inbox/deduplication	Makes message handling idempotent	Retention bounds determine duplicate protection
Compensating transaction	Semantically reverses completed work	Not every action is reversible

How to apply it

Use CQRS only when read/write needs truly diverge. Use event sourcing only when the event history itself is the authoritative business record. Model saga states, timeouts, retries, compensation, human intervention, and irreversibility. Keep the local state change and outbox record in one transaction.

Evidence, not opinion

Test replay from version zero, upcasters, duplicate and reordered messages, timeout transitions, compensation failures, relay restart, and reconciliation against source systems.

Small example

A travel booking saga reserves flight, then hotel, then payment. If hotel fails, it cancels the reversible flight hold. If refund needs manual review, the saga enters an explicit intervention state instead of pretending rollback succeeded.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Enterprise integration and migration patterns

Integration architecture protects meaning across ownership, protocol, lifecycle, and failure boundaries. Translation is usually more important than transport.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Anti-corruption layer	Protects a domain from another model	Translation code must own semantic differences
Strangler fig	Replaces legacy capability incrementally	Routing and duplicated behavior need exit criteria
Facade	Provides a simpler stable interface	Can hide but not remove underlying coupling
Adapter	Converts one interface or protocol to another	Syntax conversion is insufficient when concepts differ
Canonical data model	Standardizes exchange across many systems	Enterprise-wide models become slow and lowest-common-denominator
Aggregator	Combines related messages or responses	Completeness and timeout rules are domain decisions
Content-based router	Routes by message content	Rules can become hidden centralized business logic
Claim check	Moves large payload out of a message	Payload storage lifecycle and authorization become critical
Pipes and filters	Composes independent transformation stages	Error context and end-to-end tracing can fragment

How to apply it

Define system of record, source of truth per field, semantic mapping, contract, data classification, authorization, retry, reconciliation, and retirement path. Prefer bounded context maps over a universal model. Give temporary migration components deletion criteria.

Evidence, not opinion

Use consumer/provider contract tests, reconciliation totals, semantic examples, lineage, transformation error queues, and cutover/rollback rehearsals.

Small example

A strangler routes customer search to a new service while legacy remains authoritative for updates. Change data capture feeds the new read model; parity reports gate each migration cohort.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Domain-driven design: strategic and tactical patterns

DDD aligns software boundaries and language with business knowledge. Strategic patterns decide where models differ; tactical patterns express rules inside a boundary. DDD does not require microservices.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Ubiquitous language	Creates shared precise domain terms	Language must be contextual and continually refined
Bounded context	Defines where one model and language apply	Boundaries discovered too early can freeze misunderstanding
Core/supporting/generic subdomain	Directs investment by differentiation	Classification changes with strategy
Context map	Names relationships and influence between models	Political reality may differ from desired integration
Entity	Models identity across change	Entity-heavy models hide value semantics
Value object	Models immutable descriptive value	Equality and units must be precise
Aggregate	Protects invariants inside a transaction boundary	Large aggregates cause contention
Domain event	Records a meaningful fact in domain language	Technical events should not masquerade as business facts

Idea	What it solves	Cost, limit, or warning
Repository	Provides aggregate-oriented persistence abstraction	Generic CRUD repositories leak persistence semantics
Domain service	Holds domain behavior not owned by one entity	Can become an anemic dumping ground

How to apply it

Run event storming or domain storytelling with experts. Discover commands, facts, policies, hotspots, invariants, and terminology conflicts. Draw context relationships. Model aggregates around consistency needs, not object graphs. Keep application orchestration separate from domain policy.

Evidence, not opinion

Use example-based tests in domain language, invariant/property tests, context contract tests, and change coupling. Review the model when experts stop using its terms.

Small example

CreditLimit is a value with currency. CustomerAccount is an aggregate that protects the approved-credit invariant. CreditChecked is a domain fact; “row inserted” is not.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Distributed systems fundamentals

A distributed system adds independent failure, uncertain time, partial knowledge, and network cost. Correct architecture starts by rejecting the fallacies of distributed computing.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Partial failure	Recognizes one component can fail while others run	Status becomes uncertain, not simply up/down
Timeout	Bounds waiting for remote work	Too short creates false failure; too long consumes capacity
Consistency model	Defines what observations are allowed	Weak guarantees move complexity to users and code
Consensus	Agrees on ordered state despite failures	Requires quorum and sacrifices availability under partition
Leader election	Assigns temporary authority	Split brain and fencing must be prevented
Logical clock/version	Orders causally related changes	Does not provide physical wall time
Quorum	Uses overlapping read/write sets for consistency	Latency and availability depend on quorum placement
Fencing token	Rejects stale lock holders	Every protected resource must enforce the token
Gossip	Spreads state without central coordination	Convergence is eventual and bandwidth is probabilistic

How to apply it

State the required consistency per invariant. Define partitions, authority, ordering scope, clock assumptions, timeout and retry budgets, and behavior under partition. Prefer local transactions and explicit asynchronous coordination over pretending remote calls are local.

Evidence, not opinion

Use fault injection, clock skew, packet loss, process pause, duplicate/reorder tests, Jepsen-style history analysis where justified, and reconciliation invariants.

Small example

A lease alone cannot prevent an old worker writing after a pause. A monotonically increasing fencing token lets storage reject the stale writer even if it wakes later.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Data architecture, ownership, consistency, and analytics

Data architecture defines meaning, ownership, lifecycle, quality, access, lineage, placement, movement, and consistency. Database selection comes after workload and invariant analysis.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
System of record	Names authoritative state for a business fact	Different fields may have different authorities
Polyglot persistence	Matches stores to workload needs	Multiplies skills, operations, and consistency models
Data warehouse	Integrates historical structured analytics	Batch freshness and central modeling can limit speed
Data lake/lakehouse	Stores broad raw and curated analytical data	Governance and quality are not automatic
Data mesh	Applies domain ownership and product thinking to analytical data	Platform and governance maturity are prerequisites
Materialized view	Precomputes read results	Refresh lag and rebuild policy matter
Change data capture	Streams committed database changes	Database logs expose physical rather than domain semantics
Sharding/partitioning	Distributes data and load	Cross-partition operations become expensive
Schema evolution	Changes structure without breaking consumers	Backward/forward compatibility needs discipline

How to apply it

Classify data and assign semantic owner, steward, system of record, retention, residency, access policy, quality rules, lineage, recovery objective, and deletion process. Model transactions and queries. Choose

partition keys from access and growth, not convenience. Version contracts and migrate expand-contract.

Evidence, not opinion

Measure freshness, completeness, accuracy, uniqueness, reconciliation, lineage coverage, restore results, query latency, skew, and unauthorized access. Test deletion across replicas, caches, indexes, and analytics.

Small example

Customer contact preference is authoritative in the preference service, copied to campaign projections with version and timestamp, and reconciled daily. Campaign cannot overwrite the source record.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

API, contract, and interface architecture

An interface is a behavioral promise across an ownership boundary. Good API architecture includes semantics, identity, authorization, compatibility, failure, idempotency, quotas, observability, and lifecycle.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
REST	Models resources over standard HTTP semantics	Poor resource modeling becomes RPC with URLs
RPC/gRPC	Provides typed efficient operation calls	Tighter client/server coupling and proxies need management
GraphQL	Lets clients select a composed graph	Cost, authorization, caching, and N+1 need controls
Webhook	Pushes change to external consumers	Delivery security, retries, ordering, and replay are hard
Async API	Defines message channels and event contracts	Consumer behavior is less immediately visible

Idea	What it solves	Cost, limit, or warning
Backend for frontend	Shapes APIs for a specific client experience	Duplicated policy across BFFs is a risk
API composition	Aggregates several services for one use case	Latency and failure multiply
Pagination/cursor	Bounds large result retrieval	Mutable datasets require stable ordering semantics
Idempotency key	Makes retried commands safe	Scope, collision, response replay, and expiry need definition

How to apply it

Design from consumer tasks and domain language. Define success and every failure state, authorization at resource/action level, concurrency control, idempotency, pagination, rate policy, and version compatibility. Avoid leaking database structures. Publish examples and machine-readable contracts.

Evidence, not opinion

Run schema linting, consumer-driven contracts, backward-compatibility checks, authorization matrix tests, fuzz/property tests, quota tests, and synthetic probes. Track deprecation consumers and removal dates.

Small example

POST /transfers accepts an idempotency key and expected account version. A timeout can be retried safely; a version conflict returns a stable problem document instead of silently overwriting.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Cloud, containers, serverless, edge, and deployment architecture

Deployment architecture maps logical workloads and data to physical failure domains, trust zones, regions, capacity pools, and lifecycle mechanisms. Cloud services change responsibility; they do not remove it.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
IaaS/PaaS/SaaS	Trade control for managed responsibility at different layers	Shared-responsibility boundaries must be explicit
Containers/orchestration	Package processes and automate placement/lifecycle	Kubernetes adds a distributed control plane
Serverless/FaaS	Scales event-triggered functions with managed runtime	Cold start, limits, state, and provider coupling apply
Edge computing	Moves work near users/devices	Fleet, consistency, security, and disconnected operation grow
Sidecar	Adds a colocated supporting capability	Resource and lifecycle coupling are hidden
Ambassador	Proxies outbound connectivity for an application	Proxy failure and configuration affect the workload
Gateway aggregation	Reduces client round trips	Gateway can become a latency and ownership hotspot
Deployment stamp/cell	Replicates an isolated scale unit	Fleet automation and placement are required
Multi-region active-active	Serves and writes from multiple regions	Conflict, cost, and operational complexity are high

How to apply it

Draw production deployment nodes, zones, regions, network paths, identity boundaries, data replication, autoscaling inputs, quotas, secrets, and control-plane dependencies. Calculate capacity and recovery at each failure domain. Define immutable delivery, progressive rollout, and rollback.

Evidence, not opinion

Test zone/region loss, quota exhaustion, image/supply-chain policy, cold starts, rescheduling, secret rotation, backup restore, DNS failure, and control-plane degradation. Measure cost per transaction.

Small example

A stamp serves 2,000 tenants with its own application pool, cache, queue partition, and database shard. Routing metadata assigns tenants; new stamps add capacity and contain noisy-neighbor impact.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Reliability and resilience patterns

Resilience is the ability to preserve or recover required outcomes under disturbance. Patterns must be composed against a failure model; blindly stacking retries and redundancy can amplify failure.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Retry with jitter	Handles transient failure	Retries multiply load and can repeat unsafe work
Circuit breaker	Stops calls likely to fail	Thresholds and half-open behavior need tuning
Bulkhead	Partitions capacity to contain exhaustion	Unused reserved capacity can reduce efficiency
Timeout/deadline	Bounds resource occupation	Nested timeouts must fit one end-to-end budget
Rate limiting	Protects scarce capacity and fairness	Limits need identity and priority semantics
Load shedding	Rejects low-priority work before collapse	Business degradation order must be agreed
Fallback	Provides a bounded alternate result	Stale or misleading fallbacks can be worse than failure
Health endpoint	Exposes readiness/liveness state	Shallow checks create false confidence
Redundancy	Survives component loss	Common dependencies defeat independence
Backup/restore	Recovers lost or corrupted state	A backup is unproven until restored

How to apply it

Create a failure-mode table: trigger, scope, detection, containment, degraded behavior, recovery, data impact, and owner. Set an end-to-end deadline and allocate budgets. Retry only classified transient errors with idempotency and a retry budget. Design overload behavior before normal scaling.

Evidence, not opinion

Use SLOs/error budgets, fault injection, game days, restore drills, dependency maps, saturation alarms, reconciliation, and incident learning. Measure recovery time distribution, not the runbook estimate.

Small example

Checkout has a 1.5-second deadline. Inventory gets 400 ms with one jittered retry only for connection reset. Payment gets no automatic retry without an idempotency key. Recommendation load is shed first.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Performance, capacity, scalability, and efficiency

Performance architecture connects workload shape to latency, throughput, concurrency, resource use, queues, and cost. Scalability means maintaining objectives as a dimension grows—not merely adding servers.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Scale up/out	Adds resources to one node or more nodes	State and coordination can limit horizontal scale
Caching	Trades freshness and memory for lower latency/load	Invalidation, stampede, and authorization are hard
Read replica	Scales read workloads	Replication lag changes observed consistency
Partitioning	Spreads state and processing	Skew and cross-partition work dominate
Queue-based load leveling	Buffers bursts for bounded consumers	Adds delay and backlog risk
Backpressure	Signals producers to slow or bound work	Requires propagation across the whole path

Idea	What it solves	Cost, limit, or warning
Batching	Amortizes overhead across items	Increases wait time and partial-failure complexity
CDN	Caches content near consumers	Purge, personalization, and security constraints apply
Precomputation	Moves work before the request	Freshness and storage cost increase

How to apply it

Define workload model: arrival rate, burst, payload, mix, growth, locality, concurrency, and dependency limits. Budget end-to-end latency. Apply Little’s Law to queues. Find the bottleneck with measurement, then change one constraint at a time. Preserve load-test realism and production safeguards.

Evidence, not opinion

Track p50/p95/p99, throughput, utilization, saturation, queue depth/age, hit rate, eviction, skew, garbage collection, cost, and quality under overload. Test beyond the knee to verify graceful failure.

Small example

At 500 requests/s and 200 ms average in-system time, average concurrency is about 100. A 50-connection database pool cannot support every request holding a connection for the full path.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Security, privacy, trust, and threat modeling

Secure architecture makes trust boundaries, assets, identities, authority, data flows, threats, controls, and verification explicit. Security is a quality attribute and risk decision, not a perimeter product.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Least privilege	Grants only necessary authority for bounded time	Requires usable identity and policy lifecycle
Defense in depth	Uses independent preventive/detective/recovery controls	Duplicated controls with one failure mode add little
Zero trust	Continuously verifies identity, device, context, and policy	Does not mean trusting nothing or adding proxies everywhere
Threat modeling/STRIDE	Systematically asks how design can be abused	Checklists must include business and domain threats
Trust boundary	Marks where assumptions or authority change	Undrawn boundaries lead to implicit trust
Token exchange/delegation	Narrows authority across service calls	Audience, lifetime, and impersonation need control
Encryption/tokenization	Protects confidentiality or replaces sensitive values	Keys, metadata, and access patterns remain
Privacy by design	Minimizes collection, use, retention, and exposure	Consent does not justify unnecessary data
Audit trail	Supports accountability and investigation	Logs must be tamper-aware and privacy-safe

How to apply it

Inventory assets and data classes. Draw data flows and trust boundaries. Enumerate threats and abuse cases, rank risk, select preventive/detective/recovery controls, and map each to an owner and verification. Design identity for users, workloads, devices, and automation. Separate authentication, authorization, and business invariants.

Evidence, not opinion

Use architecture review, ASVS/control mapping, authorization matrix tests, secret rotation, attack simulation, dependency/supply-chain controls, audit queries, incident exercises, and deletion verification.

Small example

The API gateway authenticates a user, but the order service still authorizes the requested order against tenant and ownership. It sends a narrow token to payment; it does not forward a universal bearer credential.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Observability, operability, SRE, and incident architecture

A system is operable when teams can understand state, control change, handle failure, recover data, and learn. Observability is designed into contracts and context propagation, not added after deployment.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Logs	Record discrete contextual events	High volume, secrets, and inconsistent fields reduce value
Metrics	Aggregate numeric behavior over time	Cardinality and averages can hide incidents
Traces	Connect work across distributed boundaries	Sampling can miss rare critical paths
SLO/error budget	Balances reliability promises and change	Poor indicators optimize the wrong experience
Correlation/causation ID	Links related work and business intent	One trace ID may not cover long workflows
Runbook	Provides tested operational action	Stale prose can be dangerous
Feature flag	Separates release from exposure	Flags create state space and need retirement
Progressive delivery	Limits exposure while evaluating evidence	Bad metrics can promote bad releases
Post-incident learning	Improves system conditions after failure	Blame suppresses useful evidence

How to apply it

Define user journeys and SLIs before dashboards. Propagate technical and business correlation across sync calls, messages, and jobs. Make state transitions and retries visible. Design safe control actions: drain,

pause, replay, quarantine, fail over, roll back, and reconcile.

Evidence, not opinion

Prove alert precision, trace continuity, log redaction, dashboard usefulness, runbook execution, rollback time, restore, and incident roles. Use synthetic transactions and controlled failure.

Small example

An order has request ID, trace ID, order ID, saga ID, tenant ID, and message ID. Operators can follow one business outcome across minutes without searching by customer PII.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Architecture documentation: ISO 42010, views, C4, UML, SysML, and arc42

ISO/IEC/IEEE 42010:2022 distinguishes architecture from its architecture description. Documentation should answer stakeholder concerns through viewpoints and models. One giant diagram mixes scopes and becomes ambiguous.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Stakeholder/concern	Explains who needs which architectural answer	Missing stakeholders create missing views
Viewpoint/view	Defines conventions for a concern and its resulting representation	A view without a question becomes decoration
Model kind	Defines how a model is constructed and interpreted	Notation must have explicit semantics
C4 context/container/component/code	Provides hierarchical static structure zoom	C4 does not replace behavior, data, or threat views

Idea	What it solves	Cost, limit, or warning
C4 dynamic/deployment	Shows runtime interaction or physical mapping	Keep scenario and environment explicit
UML	Models structure and behavior with standardized diagrams	Over-detailed class diagrams age quickly
SysML	Models multidisciplinary system requirements, behavior, structure, and parametrics	Needs trained audience and model governance
arc42	Organizes architecture documentation into practical sections	Template completion is not architectural quality
ADL/model as code	Makes architecture queryable, diffable, and automatable	Generated diagrams still need clear purpose and layout

How to apply it

For each stakeholder concern choose the smallest useful view: context, container, component, domain, sequence, state, data flow, deployment, threat, or operations. Title the scope, declare notation, label relationships and protocols, include legend, owner, date, and link to decisions. Store near source where possible.

Evidence, not opinion

Run diagram review: scope and abstraction are clear; every element has name/type/responsibility; every relationship is directional and labeled; technologies and boundaries are explicit; diagrams match deployment. Automate drift checks where the architecture model can be compared with code or infrastructure.

Small example

The context view answers executives and external integrators. A container view answers developers. A deployment view answers operations. A threat/data-flow view answers security. They share element names but do not force every detail into one picture.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Architecture evaluation, ATAM, risk, and fitness functions

Evaluation tests whether architecture claims satisfy prioritized scenarios and exposes sensitivity points, trade-offs, risks, and non-risks. Review quality depends on concrete scenarios and evidence.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
ATAM	Evaluates trade-offs through business drivers, scenarios, approaches, and analysis	A workshop is not a substitute for experiments
Quality-attribute workshop	Elicits and prioritizes measurable scenarios	Needs broad stakeholder representation
Risk storming	Uses views and scenarios to find architectural risk collaboratively	Risk severity needs follow-up ownership
Sensitivity point	Finds a decision where small change strongly affects quality	Unknown sensitivities create surprise
Trade-off point	Finds one decision affecting multiple qualities	Optimization requires business priority
Fitness function	Continuously checks an architectural property	Only automatable properties are covered
Threat model	Evaluates adversarial failure paths	Must be updated as boundaries and assets change
Load/fault prototype	Tests uncertain quality claims	Prototype environment must represent the force being tested

How to apply it

Present mission, scope, drivers, constraints, views, and approaches. Collect scenario candidates from diverse stakeholders, prioritize them, and walk each through the design. Record risk, sensitivity, trade-off, and evidence gaps. Assign experiments or changes. Re-evaluate major deltas, not every line of code.

Evidence, not opinion

Turn recurring claims into architecture tests: dependency rules, API compatibility, latency/capacity budgets, availability exercises, recovery, data residency, security controls, cost ceilings, and ownership metadata.

Small example

A cache TTL is a sensitivity point for latency and freshness. The evaluation measures both under failure and defines a bounded stale mode instead of arguing that caching is generally good or bad.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Evolutionary architecture, governance, platform, and organization

Architecture is a living capability. Evolutionary architecture supports guided incremental change through fitness functions, reversible decisions, observability, and governance close to delivery.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Conway's law	Connects communication structures to system structure	Reorg charts alone do not create good boundaries
Inverse Conway maneuver	Shapes teams to encourage desired architecture	Forced boundaries without domain fit create coordination
Team Topologies	Clarifies stream-aligned, platform, enabling, and subsystem interactions	Labels without cognitive-load reduction add little
Paved road	Provides a supported secure delivery path	A mandatory road that ignores users becomes a blockade
Architecture runway	Builds enabling capability ahead of near-term demand	Too much runway becomes speculative platform work
Fitness function	Guards important qualities continuously	Needs business-aligned thresholds
Architecture guild	Shares practice without centralized command	Advice needs accountable owners and adoption evidence
Standard/exception process	Balances reuse with justified variation	Slow exceptions drive shadow architecture

Idea	What it solves	Cost, limit, or warning
Deprecation/retirement	Removes obsolete contracts and technology safely	Inventory and consumer evidence are prerequisites

How to apply it

Assign clear decision rights. Keep organization-wide principles few and enforceable. Provide golden paths, reference implementations, self-service platforms, and automated controls. Allow time-bound exceptions with risk owner and exit plan. Review portfolios for duplication, obsolete dependencies, and systemic concentration risk.

Evidence, not opinion

Measure platform adoption and satisfaction, cognitive load, lead time, change failure, dependency age, exception duration, architecture-test pass rate, service ownership, and retirement progress.

Small example

A platform offers identity, service templates, telemetry, delivery, and secrets as self-service. A team may deviate for a proven need, but owns extra operations and records a review trigger.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Architecture anti-patterns and failure smells

An anti-pattern is a recurring response that looks attractive but produces harmful consequences in context. Names are useful only when they lead to diagnosis and a safer migration.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Big ball of mud	Signals boundaries and ownership have eroded	A rewrite without domain learning repeats it

Idea	What it solves	Cost, limit, or warning
Distributed monolith	Services deploy or fail together despite network boundaries	More infrastructure does not create autonomy
Shared database	Lets services bypass owned contracts	Sometimes valid for modules, dangerous for claimed autonomy
Chatty interface	Requires many fine-grained remote calls	Latency and partial failure multiply
God service/object	Centralizes unrelated policy and knowledge	Splitting by method count misses domain cohesion
Golden hammer	Uses one familiar technology for every force	Standardization can hide mismatch
Resume-driven architecture	Optimizes learning novelty over system need	Long-term operational cost remains
Vendor lock-in panic	Builds abstractions for imaginary portability	The abstraction can cost more than migration
Cargo-cult microservices	Copies topology without its forces	Teams inherit distributed complexity
Lava flow	Leaves dead or half-understood paths in place	Removal requires usage evidence and ownership
Architecture astronaut	Creates abstractions far above concrete needs	Vocabulary grows while delivery slows
Single point of human failure	Depends on undocumented expert knowledge	Redundancy must include people and procedure

How to apply it

Describe the observed consequence before applying a label. Map dependencies, co-change, failure propagation, ownership, data access, and release coordination. Identify the smallest boundary or contract that can be recovered. Migrate with characterization tests, telemetry, strangling, and explicit deletion.

Evidence, not opinion

Measure cross-service release coupling, cyclic dependencies, shared-table access, remote call count, incident blast radius, bus factor, unused code paths, and time to understand a change.

Small example

Ten “microservices” share one schema, release train, library version, and synchronous chain. The corrective goal is independent capability ownership; merging some services may be the fastest path.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

A repeatable system-design and architecture interview method

A good design process moves from purpose and numbers to boundaries, data, interactions, failure, security, and evolution. Product names come after forces.

Core ideas and patterns

Idea	What it solves	Cost, limit, or warning
Clarify scope	Prevents solving a different system	Time-box questions and state assumptions
Estimate workload	Sizes traffic, storage, bandwidth, and growth	Order-of-magnitude is enough to expose constraints
Define SLOs/invariants	Turns quality into design drivers	Avoid percentages without user journey
Context and APIs	Defines actors and external contracts	Premature endpoints can freeze weak boundaries
Data model	Exposes identity, ownership, transactions, and queries	A schema is not the full consistency model
Critical flow	Explains the normal end-to-end path	Also model timeout, duplicate, and partial failure
Scale and resilience	Adapts the bottleneck and failure model	Do not add every pattern by default
Security/operations	Makes trust, telemetry, rollout, and recovery real	Often omitted under interview pressure

How to apply it

Use this sequence: outcome and non-goals; actors/context; workload estimates; invariants and quality scenarios; logical boundaries; APIs/events; data ownership and consistency; critical sequence;

deployment/failure domains; capacity; resilience; security/privacy; observability; delivery/migration; risks and ADRs. Revisit numbers when the design changes.

Evidence, not opinion

End with explicit trade-offs, bottleneck, cost drivers, unresolved risks, validation experiments, and what would trigger a different architecture. A strong answer is coherent, not a catalog dump.

Small example

For a URL shortener, clarify redirect/read dominance, custom aliases, expiry, abuse, geographic latency, and availability. Estimate key volume and read rate before choosing partitioning, cache, and multi-region behavior.

Feynman check

Explain the design to a new engineer without using the pattern names. State the problem, the forces that conflict, the chosen boundary or mechanism, what can fail, and the evidence that would prove the choice still works.

\newpage

Real-life scenario: AtlasMarket global commerce modernization

AtlasMarket operates a marketplace in twelve countries. Its ten-year-old application shares one database, has six-hour releases, duplicate charges during provider timeouts, oversold stock during campaigns, and no reliable way to trace an order across support, warehouse, and finance.

Outcomes and measurable scenarios

- a customer sees a confirmed order within two seconds at p99 during 5× demand;
- the system never creates two charges for one payment intent;
- inventory never confirms more units than are available;
- a zone failure loses no confirmed order and recovers traffic within five minutes;
- support can trace one order without reading raw personal data;
- a normal change reaches production in one hour with independently reversible rollout;
- a regional payment or recommendation outage degrades checkout safely.

Context and domains

The context includes customer web/mobile clients, seller portal, identity provider, payment providers, tax, fraud, warehouse, carrier, notifications, analytics, finance ledger, support, privacy operations, and human fulfillment.

Domain discovery produces Catalog, Pricing, Basket, Ordering, Inventory, Payment, Fulfillment, Customer, Seller, and Finance bounded contexts. Ordering owns the order lifecycle; Inventory owns available-to-promise; Payment owns payment attempts and provider idempotency; Finance owns immutable accounting.

Target architecture

Begin with a modular monolith for Catalog, Pricing, Basket, Ordering, and Customer. Extract Payment and Inventory because they need strict isolation, specialized scaling, data ownership, and independent risk controls.

```
Clients → CDN/WAF → BFF/API gateway → commerce application

Ordering → Inventory reservation API

Ordering → transactional outbox → event log → Payment/Fulfillment/Analytics

Payment orchestrator → provider adapters

Operational stores → CDC/curated events → analytical lakehouse
```

Each tenant cohort runs in a deployment stamp containing application workers, cache, message partitions, and database partitions. The control plane assigns cohorts and manages rollout. A regional ledger remains authoritative for financial facts; cross-region replication has explicit recovery semantics.

Critical order flow

1. Client submits order with idempotency key and basket version.
2. Ordering validates price snapshot and asks Inventory for a timed reservation.
3. Ordering commits PendingPayment plus an outbox record atomically.
4. Payment saga creates a provider intent using the order payment key.
5. Provider webhook is authenticated, deduplicated, and reconciled.
6. PaymentAuthorized moves the order to Confirmed and starts fulfillment.
7. Expiry or permanent failure releases inventory; uncertain payment enters reconciliation rather than blind retry.

Data and consistency

Each service owns its schema. No cross-service joins occur on the write path. Read models compose order status for customers and support. Events have stable identity, owner, schema, partition key, privacy

classification, and retention. Outbox/inbox handles at-least-once delivery. Daily reconciliation compares orders, payment provider records, inventory reservations, and finance entries.

Resilience and capacity

End-to-end checkout has a two-second deadline. Recommendation and nonessential enrichment are shed first. Remote calls have allocated timeout and retry budgets. Payment retries require stable provider idempotency. Bulkheads isolate providers and tenant stamps. Queues absorb campaigns but enforce backlog-age objectives. Restore, zone loss, provider outage, broker pause, duplicate webhook, and poison message are exercised.

Security and privacy

Trust boundaries are shown on data-flow diagrams. User identity terminates at the edge but every domain authorizes its resource. Workload identity and narrow-audience tokens protect service calls. Card data is tokenized; secrets use managed rotation; logs exclude payment and personal values. Retention, export, and deletion propagate to caches, projections, search, and analytics. STRIDE and business-abuse analysis cover price manipulation, seller fraud, inventory hoarding, replay, webhook forgery, and cross-tenant access.

Observability and delivery

Request, trace, order, saga, payment intent, message, tenant, and stamp identifiers preserve causality without logging PII. SLIs measure checkout success/latency, duplicate charge invariant, oversell invariant, queue age, reconciliation gaps, stamp saturation, and cost per order.

The migration uses a strangler route, anti-corruption layer, CDC-backed parity projections, shadow reads, cohort canaries, expand-contract schema changes, and explicit rollback. Every extracted capability has entry and exit criteria.

Architecture package

- ISO 42010 stakeholder/concern map and viewpoint definitions;
- C4 context, container, component, dynamic, and deployment views;
- domain/context map, critical sequence and state models;
- data ownership, lineage, retention, and consistency matrix;
- quality-attribute scenarios and ATAM/risk results;
- threat model and control-to-evidence map;
- capacity model, SLOs, dashboards, game-day and restore evidence;
- ADRs for modular monolith, extraction, messaging, consistency, cells, and regions.

Glossary and abbreviations

Term	Plain meaning
AD / ADL / ADF	Architecture description / language / framework
ADR	Architecture Decision Record
API / RPC	Application interface / remote procedure call
ATAM	Architecture Tradeoff Analysis Method
BFF	Backend for a Frontend
CAP	Consistency, availability, partition-tolerance trade-off framing
C4	Context, Container, Component, Code architecture views
CDN / WAF	Content delivery network / web application firewall
CDC	Change Data Capture
CQRS	Command Query Responsibility Segregation
DDD	Domain-Driven Design
DLQ	Dead-letter queue/channel
DRY / KISS / YAGNI	One knowledge source / keep simple / avoid speculative work
ESB / SOA	Enterprise service bus / service-oriented architecture
FaaS / PaaS / IaaS / SaaS	Function/platform/infrastructure/software as a service
HATEOAS	Hypermedia controls guide REST clients
IdP / IAM	Identity provider / identity and access management
MTBF / MTTR	Mean time between failure / to restore or repair
NFR	Non-functional or quality requirement
PII	Personally identifiable information
RPO / RTO	Acceptable data loss / recovery time objectives
SAGA	Long-running distributed transaction with explicit steps and compensation
SLI / SLO / SLA	Indicator / internal objective / contractual agreement
SOLID	Five object/module design heuristics
STRIDE	Spoofing, tampering, repudiation, disclosure, denial, elevation threats
SysML / UML	Systems / Unified Modeling Language

Term	Plain meaning
TTL	Time to live
Two-phase commit	Atomic commit protocol across participants
Zero trust	Verify each access using identity, context, and policy

Official and primary references

- ISO/IEC/IEEE 42010:2022
- ISO/IEC 25010:2023
- C4 model
- C4 diagram checklist
- arc42
- SEI quality attributes
- Azure architecture styles
- Cloud design patterns
- OWASP Secure by Design
- OWASP ASVS

Final Feynman challenge

Trace one AtlasMarket order from purpose, stakeholder and quality scenarios through context, domain boundaries, APIs, data ownership, consistency, deployment, failure, security, telemetry, delivery, evaluation, and evolution. At each boundary explain what is owned, what can fail, and what evidence proves the design claim.

\newpage