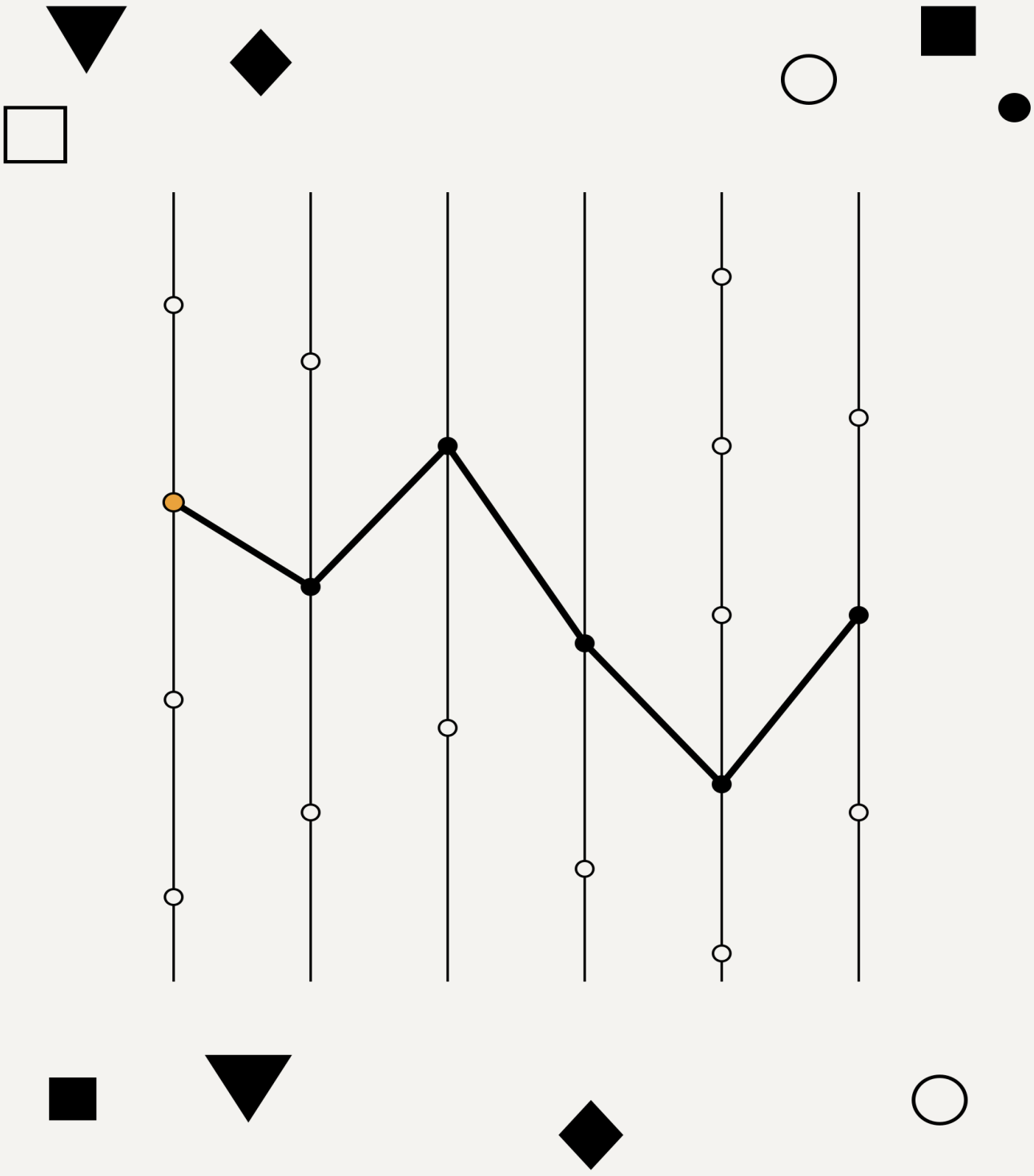
Architecture Synthesis

The Next Correct Step



Sergiy Yevtushenko

*

Architecture Synthesis

The Next Correct Step

Sergiy Yevtushenko

Version 1.0.0 — July 2026

Contents

Two Teams, Neither Wrong

Two teams build the same product: software that sells a ticket to a seat at an event. One team ships a single program talking to one database. The other ships a multi-region platform of subsystems trading events across a bus. They share almost nothing — not the topology, not the storage, not the way one part of the system talks to another.

Neither team made a mistake.

Everything in this book follows from that sentence, so it deserves a moment of pressure. If two architectures that share nothing can both be correct for the same domain, then the domain never determined the architecture. Something else did. In most teams, that something else is taste: the last conference talk, the case study about how a company a thousand times larger does it, the loudest voice in the design review. Architecture selection is the least standardized decision in backend engineering, and the industry has normalized this by giving it a dignified name. We call it trade-off analysis. Examined closely, much of it is preference with a vocabulary.

That "something else" has more candidates than taste. Team structure, the contracts that pay for the work, an appetite for risk — each genuinely bends what teams build, and an account that stops at "not the domain" owes them a place. This book's treatment is the same for all of them: written down, priced, and scoped, they become inputs like any other commitment; left unwritten, they draw boundaries nothing justifies — and Chapter 9 gives those boundaries a name — unforced positions — and a price.

There is a better account of the two teams. They had different **answers to the same questions**, and the architectures follow from the answers — mechanically. Both teams ran the same derivation; the inputs differed, so the outputs did. Architecture is not a choice you defend. It is an output you compute, and recompute when the inputs change.

That is the book's entire claim, stated as bluntly as it will ever be stated. The rest of the book earns it: this chapter runs the derivation once, in miniature, so you can watch it work before any part of it is formalized. The chapters after it take the machinery apart, run it on systems nobody asked us to derive, grade the results in public — misses included — and then turn to the harder problem: the system you inherited.

The debates are the tell

Consider what architecture discussion actually looks like where you work. Someone proposes services; someone counters with operational cost; someone cites an outage at a company neither of them works for; someone says "it depends" and the meeting ends. The same discussion reconvenes a quarter later with the same positions and no new information, because nothing in it was anchored to anything that could settle it. Architectural debate is the only technical activity we routinely conduct with no termination condition.

The debates are the symptom that exposes the disease. A debate without a termination condition is what you get when a decision has inputs nobody has written down and criteria nobody has agreed to. Strip the vocabulary away and most architecture

arguments reduce to two people defending different defaults — and often, without noticing it, arguing about entirely different properties of the system. One is worried about deployment independence, the other about consistency across a boundary; both say "microservices." The argument cannot converge because it is not about one thing.

Watch what happens to the same discussion when an input sheet exists. "We should split the checkout path" stops being a position and becomes a claim with a checkable premise: which answer does the current structure fail to contain? If the answer exists — a stated target the current position cannot meet — the split becomes a calculation, and the discussion moves to whether the answer is real. If no such answer exists, the proposal dies without a meeting. Either way, something terminated the debate, and it was not seniority.

That termination condition is what this book builds: a procedure that makes most of the advocacy unnecessary.

Three kinds of book, two of which exist

Almost everything written about software architecture is one of two kinds. The first is the **catalog**: styles, patterns, reference architectures, trade-off matrices (a well-organized menu of options, with guidance that ultimately reduces to *pick one and defend your pick*). The second is **physics**: how replication actually behaves, what a queue actually absorbs, what consensus actually costs (the mechanics of the materials). The catalogs are genuinely useful; so is the physics, and the best of it is superb. Neither tells you what to do on Monday. A catalog gives you options without a selection procedure. Physics gives you mechanisms without a decision path. Between them sits the actual daily question — *which of these, for us, now* — and on that question both kinds of book go quiet, at exactly the moment the loudest voice in the review does not.

The gap is visible even in the most disciplined corner of the field. The Software Engineering Institute — the people who took architecture evaluation more seriously than anyone — built a workshop for eliciting quality requirements from stakeholders (the QAW) and a method for evaluating a candidate architecture against them (ATAM). Elicitation on one side, evaluation on the other. Between them, where the candidate architecture actually comes from, the guidance says: expert judgment. The most rigorous framework in the industry has a mechanized front door and a mechanized back door, and in the middle, a room where the experienced architect does something no one has written down.

That middle room is this book's subject. The missing kind of book is a **procedure**: elicit these inputs, run this derivation, get an architecture out, re-derive when the inputs move. With the middle room mechanized, the SEI's two doors become more valuable, not less — elicitation feeds the derivation, and evaluation becomes a check on its output rather than a substitute for producing it.

If a procedure sounds too ambitious for something as contested as architecture, it is worth remembering that hardware engineering had this argument, and settled it, in the 1990s. Circuit design used to be schematic capture by expert hand: taste, experience, style, defended in review. Then logic synthesis arrived: a behavioral specification plus timing, area, and power constraints go in; a circuit comes out. Engi-

neers did not stop mattering. Their judgment moved upstream, into the specification and the constraints — the inputs — and out of the gate-level netlist, where it had never belonged. Nobody today argues a netlist by taste. Software architecture is a specification-and-constraints problem of precisely the same shape, and it is still being argued gate by gate.

The analogy carries a disanalogy, named here so it stays ahead of its critics. RTL is a formal language and a timing constraint is a number; an answer sheet is elicited from people, who can game it. That is why the next chapter spends half its length on an entry gate that refuses unpriced answers — the gate is this method's informal type-checker, and the method is exactly as strong as the discipline at its input. History also counsels patience about adoption curves: synthesis produced worse-than-expert netlists for years before it won. The bet is on where the curve ends, and the hardware precedent is what the bet stands on.

Answers in, vector out

A procedure needs a defined input and a defined output. Both deserve a preview here, because the miniature derivation below uses them; both get a full chapter of their own.

The input is an **answer sheet**: the system's commitments, stated as numbers with scopes. What must this operation return within, and for which percentile of requests? How much of this data class may be lost, ever? Which reads must see the writer's own write, and which may lag? What does the peak look like, on which path? Which regulator can demand what, replayed from when? Each answer is a business commitment, not a technical preference — and each answer only counts once its cost is acknowledged. "We need 99.99%" is not an answer until someone has seen what a fourth nine costs per year and still wants it. An SLO you haven't priced is a wish.

The output is not a style with a name. It is a position on **six axes**, each with a small set of values, each value carrying real capabilities and real, always-on costs:

Axis	Values
Deployment topology	single deployable / multiple / unified runtime / serverless
Composition substrate	direct calls / event-based / streaming
Read/write model	unified / separated
State storage	current-state / event-sourced
Persistence	single shared / distributed / sharded / per-component / polyglot
Recovery	compensate-by-inverse / degrade-and-continue / design-the-failure-out

"Microservices versus monolith" — the debate that consumes the industry — is one value on one axis. The other five axes exist whether or not you name them, and every one of them is being set, by someone or by default, every time a system is built. Most preference wars are two people arguing about different axes without noticing.

One more property of the output matters before you watch it get computed: values apply **at a scope**. A system does not "use event sourcing"; a data class within it does, or doesn't. A system is not "CQRS"; one read path is separated, or none is. The named styles that catalogs trade in are bundles — several axes fixed at whole-system scope and given a marketing name. The vector unbundles them, which is what makes positions checkable one at a time.

The derivation, in miniature

Three rules run the whole thing. They get a chapter of formal treatment later; here they are as they operate:

1. **Start at the cheapest position on every axis** — single deployable, direct calls, unified reads, current-state, one shared store.
2. **Move an axis only when an answer is not contained by the current position.** An answer the cheap position already satisfies **presses** on nothing; it is **inert**. Both words are load-bearing for the rest of the book, always in exactly this sense: a demand *presses* an axis when it escapes what the current position contains, and an answer that presses nothing is inert — recorded, and rightly ignored.
3. **When you must move, take the cheapest value that contains the demand, at the narrowest scope that contains it.** Event-source one data class, not the system. Split one read path, not the world.

(There is a fourth rule, for the day two answers press one axis in opposite directions. The team below never needs it — which is itself the point. It gets its treatment where it belongs.)

Now the smaller of the two teams from the opening. One venue runs its own box office: one team, one region, one country. The answer sheet, in full:

- Buying a ticket: about 2 seconds at P95. The availability check: under 1 second.
- Bookings: tens per second on a busy night. Availability checks: somewhat higher.
- Service availability: 99.5% — hours of downtime a year, survivable for one venue, priced and accepted.
- Booking data: strictly consistent, zero loss on confirmed sales. Availability and price reads: eventual is fine.
- One currency, one legal regime. Daily deploys. A tight cost ceiling. Read-moderate load.

Walk the axes and watch what presses.

Deployment topology. What would demand a second deployable? Independent release cadence between parts of the system; independent scaling for a path whose load is shaped unlike the rest; a blast-radius boundary a failure must not cross. The sheet contains none of these: one team releasing daily as one unit, load in the tens per second everywhere, an availability target that a restart budget satisfies. Nothing presses. **Single deployable.**

Composition substrate. An event substrate earns its cost across deployment boundaries — temporal decoupling, burst absorption, fan-out — and this system has no

boundary to cross. A queue here would add propagation lag and delivery semantics to paths that a function call serves in-process. Nothing presses. **Direct calls.**

Read/write model. Reads and writes share one shape; no read path carries its own contractual target; the read volume sits comfortably inside what the write model serves. A projection would be standing machinery — built, monitored, backfilled — against a problem no answer states. **Unified.**

State storage. One answer could move this axis, and it is worth naming because it so often gets bought by accident: a demand to *replay* history — to reconstruct what the system believed at a past moment, under the rules of that moment. No such answer exists here. Confirmed sales must survive; nothing must be replayed. **Current-state** — and should an auditor ever ask *who changed what, when*, an audit log written in the same transaction answers it without turning every read in the system into a projection.

Persistence. Two seconds at P95, tens of writes per second, strict consistency on bookings. A single store contains all three — and hands over the strict consistency for free, through ordinary transactions, the way single stores have done for fifty years. **Single shared store.**

Recovery. The one axis the domain itself forces, because buying a ticket moves money through sequenced steps: reserve the seat, authorize payment, confirm. Steps will fail mid-sequence; the axis asks what the domain offers for unwinding them — a question answered by the domain's shape rather than by the sheet's numbers, the second input class Chapter 2 makes formal. Here, every step has a defined inverse: release the seat, void the authorization. **Compensate by inverse.**

The vector, assembled: *single deployable / direct calls / unified reads / current-state / single shared store / compensation.* A boring program and one database.

Read the walk again and notice what did all the work. **The team didn't choose simplicity. The derivation found zero uncontained demands.** Every answer sat inside the cheapest position, so every axis stayed put. That is not minimalism as a virtue, or restraint, or good taste. It is arithmetic — and the same arithmetic cuts the other way. Copy the multi-region platform's architecture onto this answer sheet and every axis moved without a forcing answer becomes a standing bill: machinery to operate, failure modes to debug, staleness to explain — paid monthly, for capabilities no answer demands. That is how small platforms drown, and the drowning is usually described afterward as "we over-engineered," as if it were a mood. It was a computable error.

And the other team, the multi-region platform from the opening? Same procedure, different sheet: a contractual 200 milliseconds at P99 on the quote path across regions, on-sale bursts in the hundreds of thousands per minute against a fixed number of seats, a regulator requiring replayable price history, sovereignty rules pinning data by law. Each of those answers breaks containment somewhere specific, and only that axis moves, at only that scope. The full walk — and the systems between the two extremes — is Part II's work. What matters here is the shape of the explanation: **both architectures are outputs of one procedure, fed different answers.**

When the answers change

Systems do not hold still, and this is where the procedure stops being a design-day tool and becomes something else.

The venue grows. It joins a regional network; a partner contract now specifies availability checks under 500 milliseconds; the read load goes heavy. What happens to the architecture? Precisely one thing: the read path — and only the read path — now carries an answer the current position no longer contains. The derivation moves one axis, at one scope: read replicas behind the same unified model, because the staleness budget allows them and they cost less than projections. Everything else stays. No migration program, no target-state architecture, no two-year roadmap. **The next correct step, computed from the new answers.**

That phrase is the book's subtitle, and this is the reframe it names: an architecture is not a state a system is in. It is a **derivative**: the next correct step from where the system stands, given the forces acting on it now. A greenfield architecture is the derivative taken from a blank page. The system you inherited — running for a decade, carrying decisions nobody alive remembers making — is the same computation from a harder starting point. Part III takes that reframe seriously, all the way to a real inherited system with a public audit trail.

What the book owes you now

A claim like "derived, not chosen" creates obligations, and the book accepts all of them.

First: **where do the questions come from?** A derivation is only as honest as its inputs, and a question list handed down without justification is just taste moved upstream. The next chapter holds the questions to a membership criterion — every one has to demonstrate that its answer can force an axis on its own — and the sheet's count is that test's output, free to grow the day a new demand earns a seat.

Second: **why these axes, and what do the values actually buy?** Every axis value carries capabilities and always-on costs, and "cheapest containing" means nothing until the costs are on the table. That ledger is Chapter 3, and the derivation rules that consume it are Chapter 4, with the verification that closes the loop in Chapter 5.

Third, and this is the obligation most methodology books quietly decline: **proof against systems we do not control.** Part II re-runs ticketing at three scales with every step cited. Then it does something stricter: derives four real systems — Stack Overflow, Shopify, Discord, and a British government registrar — from their public commitments alone, with predictions registered before looking at the outcomes, and grades the results in the open. The procedure gets some things impressively right. It also misses, and the misses are kept, because a method that never fails is a parlor trick, and what a miss teaches is worth more than what a hit confirms.

Fourth: **naming where judgment remains.** A method that claims to mechanize everything is selling something; this one maintains an explicit inventory of the decisions it hands back — targets, recovery ties, contradiction choices, product picks — and Chapter 12 is that inventory, kept as carefully as the mechanics.

Two disciplines run underneath all four obligations, stated once here and honored throughout. Claims wear their grade: **derived** (follows from cited answers by named rules), **empirical** (graded against a documented outcome), **heuristic** (works across the validation record, mechanism plausible), or **contextual** (true in a scope, stated with it), and the provenance labels you will meet in the evidence chapters ([reconstruction], UNVERIFIED, registered-and-graded) are this legend at work. And the method's assumptions are on the table now, not discovered later: enterprise backend systems in the percentile regime; answers that can be elicited and priced from someone accountable for them; ledger entries that state mechanism physics, not fashion. Where an assumption fails — hard real-time correctness, unpriceable answers, novel mechanisms without operational record — the method says so instead of stretching.

The two teams from the opening never argued, because they never met. Your teams meet every quarter. The next chapter starts where every derivation starts — with the questions, and with the discipline that makes answers worth deriving from.

The Answer Sheet

The meeting is scheduled for an hour and runs ten minutes over. The product owner wants the new platform "highly available — let's say 99.99%," "real-time where it matters," and "scalable, obviously." The architect writes all three on the whiteboard. Everyone nods; the meeting ends; the requirements document says exactly what the whiteboard said. Six months later the platform carries a cross-region consensus store nobody can operate, a queue on paths that needed function calls, and a latency target no one can state from memory — and every one of those purchases traces back to that whiteboard, where three wishes were recorded as answers.

Chapter 1 claimed the architecture follows from the answers. That claim puts enormous weight on the answers, and this chapter is where the weight lands. A derivation is only as honest as its inputs: garbage answers derive garbage vectors with perfect mechanical integrity. So the input side needs two things a requirements meeting never produces on its own — the right questions, and a gate that keeps wishes from passing as answers.

It also needs something subtler, which this chapter does in the open: a justification for the questions themselves. A question list handed down without argument is taste moved upstream — the loudest voice replaced by the author's voice, which is no improvement. Every question below has to demonstrate its seat: its answer must be able to force an axis on its own. The list was not born this clean; it is the survivor of an audit against this method's own derivation record, and its count is an output of that audit rather than a choice.

What a question must do to earn a seat

The derivation gives questions a job description. An answer participates by *pressing* an axis toward a value or *pruning* values an axis can no longer hold. So the membership criterion writes itself: **a question earns its place only if its answer can press or prune at least one axis independently of every other question's answer.** A question whose answers never move anything is decoration. A question whose pressure always arrives bundled with another question's is a duplicate wearing different words.

Nine questions hold seats under that criterion, each audited against every derivation this book runs. The count deserves one sentence of standing law rather than a history: it is an output, and if a tenth demand ever demonstrates independent pressure — pressing or pruning an axis no current answer reaches — the sheet grows. The criterion outranks the number.

The questions organize by what their answers *do* in the derivation, because that is the property that matters when you hold the sheet. Five driver modes cover them.

Prune mode: the correctness answers

Prune-mode answers are binary. They do not trade against money or cleverness; a value that cannot honor them is struck, and no amount of hardware buys it back.

The consistency contract — per data class and per path: which reads must be strict, which may lag behind writes and by how much, who must see their own writes. This is the question people answer with a shrug and pay for with an incident. The contract is per data class because one system legitimately carries several: the booking that must be strict and the availability display that tolerates thirty seconds of lag are different contracts in one product.

The loss budget — per data class: how much of this data may be lost, ever, and how long must it be kept. A recovery-point objective of zero on confirmed sales is a prune-mode fact: values that cannot durably commit before acknowledging are simply gone from the board.

External constraints — the regulator's and the contract's reach: what must be auditable, what must be *replayable*, what must be *erasable*, where data must reside, which mandates strike values outright. One decomposition inside this question does more damage prevention than any other in the sheet, and it gets its own section below.

Multi-X — countries, currencies, tenants, regions, versions. Two different outputs hide here: partition *gifts* (tenants that never interact are a natural shard key — Part II shows one carrying a hundred-billion-dollar platform) and legal *pins* (data that must stay in-country by statute). One question, because both arrive in the same breath of business fact; two readings, because one gives capacity away and the other takes options off the table.

Select, isolate, split, bound

The time budget (select mode) — per operation, and *shaped*, never a scalar: percentiles and tails for request paths, soft maxima where a human waits, completion windows where a batch must land inside a business deadline. Latency selects among surviving values rather than striking them, and it presses only near a physics floor: a 2-second budget over a 2-millisecond path forces nothing, a 50-millisecond contract over a cross-region hop forces everything. The shape catalog doubles as a scope detector: an answer with a *hard* maximum as a correctness criterion — a deadline whose breach is a system failure, at hard-real-time strictness — is the tripwire that says this book's methods stop applying. Enterprise backends live in percentiles; anti-lock brakes do not, and the sheet must say so at elicitation time rather than let the derivation discover it.

The failure budget (isolate mode) — per operation: the error budget and the criticality that sets it. A 99.5% target is forty-four hours a year, and a restart strategy fits inside it; a 99.99% target is fifty-three minutes, and whole classes of design stop fitting. Availability starts biting past roughly three nines on a named path, and what it bites is blast radius: the axis moves it forces are isolations.

Load (split mode) — magnitude at steady state and peak, shape per path, concentration, and window. This is the merged question, and the merge taught a discipline: load answers are only meaningful *per path*, because "the system is read-heavy" is routinely true at one tier and false at another: Part II contains a system that is read-dominated at the HTTP tier and write-majority at the store, simultaneously, and any sheet that recorded one number for it would be wrong twice. Load presses only on *divergence*:

uniform load, however large, presses sizing rather than structure. The shape vocabulary that decides what a load answer can force — volume (sustained magnitude), burst (a peak with a tolerable settling delay), contention (many actors, one record), deadline (finish inside a window) — belongs to the ledger, where each shape meets the mechanism that contains it.

Release structure (bound mode) — cadence divergence between parts of the system, and deploy safety; one of this question's usual bundles is a fake driver so common it needs its own paragraph below.

The cost and capacity envelope (bound mode) — money, and the question that sounds too small to matter and decides more architectures than latency does: *who operates this?* Four engineers and no operations team is an answer that moves axes. Bound-mode answers rarely press a single move; they set the envelope every other resolution optimizes inside.

The entry gate: answers are purchased, not collected

The whiteboard meeting failed before any question was asked, because it treated answers as things people say. The derivation treats answers as things people *buy*. Five disciplines make up the gate.

Priced. Chapter 1 planted the flag on unpriced SLOs; the gate is where pricing happens. A fourth nine costs a number someone can compute — in redundancy, in on-call, in correlated-failure engineering — and the pricing drill is one question long: *what does the business do differently in the fifty-third minute of annual downtime?* If the honest answer is "nothing," the real target just revealed itself, one nine lower and an order of magnitude cheaper. Most nines dissolve under this question. The ones that survive are real, and the derivation treats them with full respect.

Scoped. Scope lives inside the question — per operation, per data class, per path — never around the system. One system-level number is how bare adjectives sneak back wearing digits. The banned vocabulary is worth naming: "scalability," "high availability," "performance" (words that hide both the scope and the shape of a demand). The sheet forces the honest versions: *which path*, *under what load*, *within what budget*.

Decomposed. Answers arrive as bundles, and bundles press falsely. The canonical fake driver: "the teams need independence." Decomposed, it splits into ownership boundaries (which module boundaries contain at zero deployment cost) and release-cadence independence, which is a genuine topology driver that most teams, asked directly, do not actually want: they want to stop stepping on each other, and one deploy train with owned modules delivers that. The bundled version buys a service fleet; the decomposed version buys a directory structure. The same discipline dismantles "we need audit," which is where the most expensive accident in the storage axis lives, and it splits three ways, because a third demand hides behind the same meeting language. **Audit** asks *who changed what, when*: answerable by an audit log written alongside current state. **Replay** asks *what did the system believe on March 3rd under March 3rd's rules*: answerable only by event-sourced history. And **erasure** — the right to be forgotten, a statutory demand wherever personal data lives — runs *against* both: it requires that specific facts become irrecoverable, which strikes exactly the append-

forever machinery replay buys. Three demands that sound alike in a requirements meeting force three different storage answers, and one pair of them can genuinely contradict: replay-required and erasure-required on the same data class is a statutory collision the derivation must surface, not paper over. The regulated industries' folklore that "compliance requires event sourcing" is this three-way decomposition skipped, at seven figures.

Triaged. Two things masquerade as answers and belong outside the sheet. Business-process deadlines bind the *requester's* clock: a statutory 14-day filing window is a real obligation that no latency mechanism can address and no architecture axis feels: it re-enters later as a calendar constraint on transformations. Only system-clock targets press. And observed failures — last quarter's outage, the incident that started the whole initiative — are evidence about failure domains, never demands: they earn a place in the derivation only if the business converts them into a number it will pay for, which is the gate doing its job on trauma the way it does on wishes.

Surfaced early. When two stakeholders answer the same question differently at the same scope — the CFO wants every report strictly current, the COO won't fund the store that could deliver it — the sheet has caught a business contradiction wearing an engineering costume. Catching it here, where renegotiation costs a meeting, is the cheap version of a discovery that otherwise arrives in production. What the derivation does when a contradiction survives all the way through is Chapter 8's subject, and it is the method at its most useful.

The second row source

One kind of input deliberately lives outside the nine questions, because no stakeholder answers it — the domain itself does, through the analysis work this book's companion methodology owns. Two families of fact arrive this way.

Domain-shape facts, for every operation that changes the world: does a *defined inverse* exist (a reservation releases, an authorization voids — but an email does not unsend)? does the value *decay* (a seat hold that expires is a fact with a half-life)? can the operation be *reshaped* so a whole class of failure cannot occur (made idempotent, commutative, append-only)? These drive exactly one axis — recovery.

Change-driver facts, for the system's rules and data classes: how often, and under whose control, does the logic governing this data class change? A benefits platform whose rules move with political weather, a pricing engine whose rules move with the market, a tax calculation whose rules move annually by statute — these are facts about the domain's *volatility structure*, and they come from the same change-driver analysis that PFD builds its entire design method on. The analysis method is PFD's; the sheet does not require the book — the question as stated, asked of every rule set and data class and answered by whoever owns the roadmap, produces the row this book needs. They rarely press an axis alone; their power is in combination — volatility meeting a continuity answer is what forces isolation between a paying path and its changing rules, as Part III's largest case shows at national scale. This is the precise seam where PFD's central concept feeds this book's sheet: the design method finds the change drivers; the architecture method prices what they press on.

Both families are analysis output, not elicitation answers, and pretending they are a tenth and eleventh question would misstate where they come from. The sheet has nine questions and a second row source with two kinds of row.

What a filled sheet looks like

The venue's sheet in Chapter 1 is the miniature. A real one runs a few pages: every answer a number with a scope and a citation to whoever committed to it, UNKNOWNs recorded as UNKNOWNs — an honest UNKNOWN derives a null position and a note, which beats a confident guess deriving machinery — and the domain-shape facts listed per effectful operation. It reads less like a requirements document and more like a term sheet, which is what it is: the business's commitments, in the only form a derivation can consume.

The sheet also defines what it means to disagree with this book productively. Every dispute about an architecture derived from a sheet is a dispute about a row — a price someone didn't accept, a scope drawn wrong, a decomposition missed. Arguments about rows terminate. That was the promise of Chapter 1, delivered by structure.

Answers press; something has to receive the pressure. Each axis holds a small set of values, each value provides specific capabilities through a specific mechanism at a specific ongoing cost — and "cheapest value that contains the demand" stays a slogan until those costs are on the table. The next chapter builds the table.

Instruments introduced: the nine-question sheet · driver modes (prune / select / isolate / split / bound) · the membership criterion · the entry gate (priced, scoped, decomposed, triaged, surfaced) · the audit / replay / erasure decomposition · the second row source (domain-shape facts, change-driver facts).

The Six Axes and the Ledger

A chat platform's busiest channel goes hot: a hundred thousand people watching one conversation, every one of them pulling the same recent messages from the same database partition. Concurrency climbs, latency cascades, the partition melts. The team's first instinct is the industry's first instinct — shard harder — and it cannot work, structurally: those reads want *one channel's* rows, and one channel has one home no matter how many shards exist. What contains the melt is a thin layer in front of the store that notices a hundred thousand identical in-flight reads and sends the query *once*. The fix costs a fraction of the resharding project it replaced, and no catalog of architecture styles contains it, because it lives below the altitude catalogs describe: it is a fact about what mechanisms can contain.

This chapter is a book of such facts. The method's geometry has three spaces: the **demand space** Chapter 2 produced, where commitments live as numbers with scopes; the **position space** Chapter 4 will walk, where every architecture is a vector on the six axes; and between them the **selection space** this chapter maps — what each position on each axis *provides*, through which *mechanism*, at what *always-on cost*. Classic architecture literature works in this middle space with quality-attribute vocabulary and drowns, because "-ilities" are not decidable. The catalogs skip the space entirely, jumping from demands to named styles by narrated judgment. The derivation becomes checkable at exactly the moment this middle space becomes explicit: **selection is the cheapest vector whose capability envelope contains the demands** — a sentence that means nothing until capabilities and costs are written down, and everything after.

The written-down version is the **ledger**. Its discipline, before its contents:

- Entries are **mechanism physics** — what a projection costs, what a queue absorbs, what quorum commit cannot avoid (the round trips a write pays to reach agreement across a majority of replicas). They are never compatibility rules between styles; interactions surface inside derivations, as contrast.
- Values are **atomic**. "Hybrid" and "mixed" are compositions produced by scope splits, priced as parts plus the boundary between them.
- Values apply **at demand scope**. Event-source one data class; separate one read path; extract one component. A value applied wider than its demanding scope is unforced cost — a definition of technical debt this book will use repeatedly.
- Costs are **always-on**. A mechanism bills for existing — operation, evolution, failure modes — whether or not the capability is exercised this quarter.
- **Containment is a claim about a bound.** A value or mechanism *contains* a demand when it holds the demand's stated number — at its percentile, over its window, under its declared shape — at the demand's scope, at the mechanism's always-on cost. The claim is exactly as probabilistic as the bound it serves: a replica pool contains a read-volume demand at P95 and at a named staleness, not absolutely; an admission gate contains a contention demand up to its configured arrival rate and not beyond. Whatever assumptions the bound carries, the containment claim inherits — which is why every containment survives or dies at the exit gate's arithmetic rather than on the ledger's word.

Demand shapes: what kind of pressure is this?

Before any axis question, a load answer must declare its shape, because shape selects the containing mechanism family — and the industry's standing error is buying volume mechanisms for contention problems. One diagnostic separates them: **would a second copy help?**

Volume says yes. Sustained magnitude spreads across copies, caches, shards; volume is the shape capacity answers.

Contention says no. Many actors, one record — one seat at the on-sale moment, one hot channel, one flash-sale inventory row. The contended record has one home regardless of fleet size, so containment means governing arrivals: admission control and fair queueing on the write side, request coalescing on the read side, or reshaping the operation so the conflict cannot occur. Sharding a contention problem buys hardware and keeps the melt.

Burst is a peak with a tolerable settling delay, and its mechanism is a buffer: the queue absorbs what the deepest dependency never sees. The definition contains its own test — a peak that must be served *synchronously* is volume at the peak number, and calling it a burst just moves the failure two weeks later.

Deadline is "finish inside the window," a batch shape: payroll by the 25th, filings by year-end. Deadlines are answered by windowed throughput and resumable work, and latency mechanisms are irrelevant to them — with the Chapter 2 triage in force, because most deadlines in a sheet bind the requester's clock and press nothing at all.

The shape taxonomy is deliberately open at the front. New workload families arrive wearing new clothes and old shapes: a GPU serving one large model is *contention* (one resource, many claimants — admission and batching, never naive replication of what cannot be copied cheaply); token streaming is the *streaming* substrate's shape at session scope; a training run against a cluster deadline is *deadline*-shaped with checkpointing as its resumable-batch mechanism. The diagnostic questions do not change; the examples will.

Containment rungs: the moves that are free of architecture

Some mechanisms contain demands without moving any axis, and pricing them first is what keeps derivations minimal. The rule that governs them: a tier that owns **no business logic and no data of record** — a load balancer, a cache, a coalescer, an admission gate — is a containment mechanism, invisible to the vector. Systems accrete many such tiers; their architecture does not change thereby, and a vector that counted them would overstate every real system it described.

Rung zero is hardware. Sizing the cheapest position — more memory, more cores, the whole working set in RAM — contains volume and latency pressure with zero new mechanisms, which by this chapter's selection logic makes it the first bid on every containment, and one of the great engineering cultures of the web ran on that rule as an explicit company value. The rung has a ceiling, and the ceiling is economic rather

than technical: it stops when the working set or the write rate outgrows what one box's price curve buys, and past that point every further dollar purchases less than the mechanism it was avoiding. The price curve itself is substrate-dependent, and the rung must be priced on *yours*: on owned metal the curve is gentle and the rung famously long; on cloud pricing it steps at instance-size cliffs, bills egress separately, and caps out earlier: the same rule, run against a different ledger row, sometimes returning a different answer. What does not change is the rule's position: first bid, priced before any axis moves.

Read pressure climbs a fixed chain: cache → coalescing → replicas → projections. Each rung contains a different shape: repeated reads of hot values; concurrent *identical in-flight* reads; same-shape read volume past one node; and only at the top, reads whose *shape itself* diverges from the write model. The top rung is the axis move. Everything below it is not, and the chain's discipline is that you climb it rung by rung, stopping where your citations stop. Part II grades this chain against three industrial systems that stopped at three different rungs — each stop forced by that system's answers — and against the fashion that begins at the top rung by default and calls it best practice.

The axes

Six axes, each with atomic values, each value a **provides / mechanism / costs** entry — and the entry's three fields are exactly the joints the procedure will articulate: prune strikes a value when *provides* cannot meet a correctness demand; press tests demands against what *provides* holds at scope; resolve counts *mechanisms*; and *costs* is the always-on bill the whole selection minimizes. A richer schema — requires columns, excludes columns — would add nothing the three fields do not already encode: exclusion is the absence of a provides, and a prerequisite is a mechanism, priced like any other. What follows renders the ledger at chapter altitude — the full entries live in the reference cards.

Deployment topology — *single deployable / multiple deployables / unified runtime / serverless.* This axis prices two countables, kept apart because its costs split between them: the **artifact**, versioned and released as one — release coupling and the delivery plumbing bill here — and the **runtime unit**, an instance group shaped and scaled as one — scaling shape and runtime blast radius bill here; the network inside a crossing bills to neither, only to the boundaries the composition substrate actually crosses, and is largely pre-paid where the substrate is already event-based. The single deployable — one artifact, identical instances — provides in-process calls on every internal path, one operational surface, and one composition in production — the composition you test is the composition that runs — at the cost of whole-system blast radius and whole-unit scaling; its modular form adds ownership boundaries at zero deployment cost, which is the fact that dissolves most "we need services" pressure once Chapter 2's decomposition has run. Its **role-selective form** — the same artifact, handlers or process classes enabled per instance group, a form with decades of record as web-and-worker processes from one codebase and database node roles from one binary — buys back per-role scaling envelopes and inter-pool blast isolation with no second artifact, at the price of the activation machinery becoming a mechanism of its own: role configura-

tion, per-role capacity, knowing which instance runs what. What no single artifact buys is release independence: one artifact is one train, disabling a handler at runtime is a kill-switch — recovery machinery, not a cadence — and waking a dormant handler on a running instance takes the isolation back, one process envelope now shared. The train itself is a bill that grows with the codebase — one build, one test gate every team's merge waits behind. Module-aware builds and affected-only test selection contain it a long way, a containment rung of its own; what remains when they cap out is a cadence demand, pressing the axis through the answer sheet rather than around it. Multiple deployables provide what only artifact multiplicity can: independent release cadence, independent toolchains and dependency evolution, blast isolation that holds at deploy time too. Their cost is a network inside every crossing — a latency floor, partial failure as an internal concern, consistency across the boundary decaying from transaction to protocol — paid in full where calls are direct, and mostly pre-paid where the substrate move already put a broker between the parts. Their second cost is the delivery plumbing, multiplied: each artifact is its own pipeline, its own release process, its own dependency stream to keep patched — and its own row in a version matrix, because independent cadences mean production runs a mixture of versions through every rollout window. The composition that was tested is no longer the only composition that runs; cross-artifact compatibility stops being an event and becomes a standing bill — contract tests, deprecation windows, the N-by-M pairs. The unified runtime holds packaging open — one-or-many decided at deploy time without rewiring — at the cost of a platform dependency; the role-selective form is its hand-rolled ancestor, and what the product adds is the same deploy-time freedom for *direct-call* composition. The platform class is young; the pattern is not. This book's author builds one such runtime, disclosed here precisely so the derivations can be checked for favor: none requires it. Serverless provides per-invocation scaling to zero at the cost of cold-start tails and per-invocation pricing that crosses over against sustained load.

Composition substrate — *direct calls / event-based / streaming.* Direct provides the lowest latency and read-your-effects immediacy at the cost of temporal coupling: the callee must be up *now*, availability multiplies down the chain, bursts arrive unbuffered at the deepest dependency. Event-based buys temporal decoupling, burst absorption, and fan-out, and its price deserves stating in full because it is so often paid unknowingly: propagation lag on every consumer view, delivery semantics that force idempotent consumers, ordering only per key — and the between-steps state becomes durable, named, and *operated*. Streaming provides ordered, replayable, consumer-paced consumption for the one data class whose volume earns a partitioned log, at the cost of partition-key design becoming load-bearing.

Read/write model — *unified / separated.* Unified provides read-your-writes for free and zero projection machinery; the entire containment chain above lives inside this value, which is why it survives far more load than its reputation suggests. Separated provides independent scaling and shape for one read path, through projections, at the cost of a staleness window, machinery to build and backfill, and dual schema evolution — a price worth paying exactly when a read path carries its own contractual target *and* its own shape, and worth refusing otherwise.

State storage — *current-state / event-sourced.* Current-state provides the read as the state; paired with an audit log written in the same transaction it answers every

who-what-when question, which is why Chapter 2's three-way decomposition guards this axis. Event-sourced provides genuine replay — state at any past moment, *why* under that moment's rules — at a cost that compounds forever: every read a projection, event schemas versioned for life, unbounded growth, and a model the team must learn to think in. The value is real and the demand that earns it is rare; the ledger's job is keeping the two matched. And one statutory demand runs *against* this axis's expensive value: erasure. Where the law requires that personal data become irrecoverable, append-forever machinery meets its counter-demand, and the containment mechanisms are their own small ledger row: scope exclusion first (keep personal data out of the immutable stream, referenced from a mutable store), crypto-shredding (encrypt per subject, erase by destroying the key), tombstoning where the store supports honest deletion. Replay-required and erasure-required *on the same data class* is a genuine contradiction with a statutory source: the derivation surfaces it and hands back the menu, exactly as Chapter 8 shows.

Persistence configuration — *single shared / distributed shared / sharded / per-component / polyglot.* The single shared store provides cross-component transactions for free: strict consistency, delivered by fifty years of engineering, at the cost of a shared ceiling and shared blast radius. The distributed shared store holds a unique position: it is the *only* value in this ledger that provides strict transactions across regions with zero data loss on regional failure, and its price is physics, a write floor of cross-region round trips times quorum, which no vendor tunes away. Sharded provides write scaling past one node when the demand set carries a natural partition key; its cost is that cross-shard transactions effectively cease and the key becomes load-bearing. One escalation deserves its own entry: when volume sharding compounds with blast-radius isolation — one tenant's disaster must not touch another — the shard boundary widens to the **full stack**, a complete isolated instance of the system per shard: cells. The cost is the whole stack duplicated per cell and cross-cell features structurally forbidden, and the prohibition is the value working, as Part II's hundred-plus-cell example shows. Per-component and polyglot provide independent evolution and stores shaped to their data, at the cost of transactions across components decaying to protocol and one operational competency per store technology.

Recovery — *compensate-by-inverse / degrade-and-continue / design-out* — the axis with no null, because every effectful operation must answer it, and the axis whose inputs are Chapter 2's domain-shape facts rather than anyone's preferences. It is also the axis a careful reader flags as the odd one out: the other five position *structure*; recovery prescribes *behavior under failure*. The asymmetry is real, and the seat is earned under the same criterion as the rest — systems with different recovery answers differ structurally before any technology is chosen: compensation paths are real code with real tests, design-out reshapes the domain model itself, degradation demands bounded windows with visible state. The exit gate's failure column consumes exactly this axis's choices, per operation. An axis is a dimension along which systems must differ; nothing requires the dimensions to be the same kind of thing. Readers of this book's companions know these three as **the recovery triple**: BER, FER, and design-out; the classes are identical, and the short names remain the series' compact notation (backward recovery compensates, forward recovery degrades and continues)

while the long names carry the prose, because a name that says what you do beats one you have to look up. Compensation provides restored consistency where the domain offers an inverse, at the cost of compensation paths that are real code with real tests, and where the inverse must be *built*, it is a use case of its own with its own targets. Degrade-and-continue provides forward progress where staleness or decay is tolerable (defaults, queues-for-later, values that expire) at the cost of degraded windows that must be bounded and visible. Design-out provides the strongest guarantee available anywhere in this book: the failure class stops existing — idempotent writes, commutative operations, append-only records, a structural constraint that makes the double-booking unrepresentable — at a price paid once, in the shape of the model. Where design-out is available it embarrasses the alternatives, and checking for it first is a standing rule.

The boundary has a price

Compositions appear throughout the values above — separated *at one path*, sharded *at one scope* — and every composition contains a boundary, which the ledger prices like any mechanism. A scope split pays: a contract, versioned and negotiated evermore; consistency across the seam decaying from transaction to protocol; a translation layer answering to both sides' changes; an operational seam where deploys, monitoring, and incidents must be correlated, and for persistence splits, one durability story per store. This entry cuts both ways. It is why splits must be forced rather than fashionable, and it is why, when Chapter 4's conflict rule finds pressures at genuinely different scopes, splitting *at that boundary* is honest engineering: the price buys separation the demands actually require.

The selection rule

The ledger closes with the rule that consumes it, and the rule is short:

1. Start at the **null vector** — the cheapest value on every axis: single deployable, direct calls, unified reads, current-state, single shared store.
2. Move only when a demand is **uncontained** — and check the rungs before conceding that it is.
3. For prune-mode demands, test **scope exclusion** first: narrowing where the demand applies can beat hardening any axis — Part II contains a payment-card mandate contained by routing card data around the entire system, which thereby never entered the mandate's scope at all.
4. Among containing values, prefer the one introducing the **fewest new mechanisms** — each is an ongoing bill, and mechanism-counting is how "cheapest" stays honest when prices resist arithmetic. This is the precise sense of Chapter 1's "arithmetic": not that costs add like numbers — they do not — but that the comparison is mechanical, counting mechanisms rather than weighing taste.
5. Apply the value at the **narrowest scope that contains the demand**.

Why six axes and not sixteen? The same criterion that governed the questions, mirrored: an axis earns its seat only if systems with different answers must differ *structurally* along it before any technology is chosen. Candidates that fail collapse into technology choices — products, languages, clouds — which matter enormously and are not

architecture. The strongest rejected candidate deserves its rejection shown, because it is the one every reviewer proposes first: **security topology** (trust boundaries, tenancy isolation, authentication placement). Run the criterion: the demands it carries route through axes that already exist: tenancy isolation is blast-radius isolation and arrives at cells; data-protection mandates are prune-mode external constraints and arrive at scope exclusion or erasure mechanisms; what remains after those routings (which gateway, which identity product, mTLS or tokens) is mechanism and technology choice, real work that no *structural* position decides. Two systems with different security answers do not differ along a seventh axis; they differ along the six, plus their technology sheet. The other candidate every reviewer proposes is **team topology** — the org chart, Conway's Law offered as a seventh axis. Run the criterion: team ownership is a real pressure, and the deployment entry above already contains it — module boundaries at zero deployment cost, with a genuine split forced only when release cadences diverge, a demand already on the sheet, pressing an axis that already exists. A team structure that splits a system no answer divides is not a missing dimension; it is an unforced position, Chapter 9's subject. Two systems with different org charts and identical answers derive the same vector; the org chart that overrode the derivation left debt behind, not a new kind of structure. The count is an output here exactly as it was in Chapter 2: the criterion is the theory, six is its current result, and the result is empirical — earned against the derivation record, not proven from first principles. A completeness proof for a design space is not on offer, here or anywhere. If a future derivation finds a security or organizational demand that presses nothing existing and forces structure of its own kind, the criterion outranks the count here too. That is what it is for. The six above have survived every derivation this book runs, one of them by surviving an audit this book nearly retired it with; Part II tells that story against a live system, because an instrument's near-death is evidence about the instrument.

Demands from Chapter 2, capabilities and costs from this one. What remains is the procedure that walks them against each other — including the moment two demands press one axis in opposite directions, which is where most methodologies wave their hands and this one runs a test.

Instruments introduced: the three-spaces model · the ledger and its discipline · demand shapes and the second-copy diagnostic · containment rungs and the read chain · the six axes' values · the boundary cost · the selection rule · the axis-membership criterion.

The Derivation

An order platform's design review has been stuck for a month. The checkout lead insists on strict consistency and a transactional store — money moves, carts commit, nothing may be half-true. The analytics lead insists on eventual consistency and cheap horizontal reads — dashboards scan millions of rows and nobody cares about the last ninety seconds. Both cite the consistency axis. Both are right. The review treats this as a conflict to be won, schedules another meeting, and escalates to the CTO, who will decide it by seniority, which is to say by taste.

The month was wasted on a question that has a mechanical answer, because the two demands never actually met: one attaches to the checkout path, the other to the reporting path. They press the same axis *at different scopes*, and pressures at different scopes are an instruction, not a conflict — the system splits at the boundary between them, and each side gets the value its own demands force. The meeting was arguing about which half of the truth would govern the whole system. The derivation's answer is that nothing has to govern the whole system; that is what scopes are for.

This chapter assembles the full procedure — the two instruments from Chapters 2 and 3, plus the resolution logic that handles everything the miniature in Chapter 1 was too small to need. The procedure has a name, `next_step`, chosen for what it computes: given where a system stands and what the answers currently are, the next correct position. Greenfield is the special case where the system stands nowhere.

The procedure

Step 0 — the null vector. Every derivation starts from the null vector — the cheapest value on every axis, exactly as Chapter 3's selection rule defined it. Recovery has no null; it is decided per effectful operation in step 4. For greenfield, the starting position *is* the null vector; for a living system, the starting position is wherever the system stands, and everything else proceeds identically — a symmetry Part III spends three chapters on.

The null vector earns a defense, because starting cheap sounds like an aesthetic. It is a measurement precondition. Pressure is only detectable as *escape from containment*, so containment must start somewhere defined, and it must start at the bottom: start anywhere higher and the unused capability is invisible — nothing presses against a wall that was already built, and you cannot distinguish the wall that earns its cost from the wall that is simply there. Every axis the null vector holds is a claim the answers are free to refute. That is the correct relationship between a method and its inputs.

Step 1 — normalize. The entry gate from Chapter 2, run to completion: every answer priced, scoped, shaped, decomposed to mechanism level, triaged. Nothing new here except the insistence that it happens *before* any axis is discussed — a design meeting that starts at "should we use services" has skipped the step where most of its agenda dissolves.

Step 2 — prune. Correctness answers strike values. The consistency contract, the loss budget, residency pins, and mandates that survive scope exclusion each remove from the board every value whose ledger entry cannot provide them. Pruning is binary and cheap, and it goes first because negotiability orders the work: correctness cannot be bought back, physics can only be respected, economics optimizes inside what remains.

Step 3 — press. For each remaining demand, check containment against the current vector's capability envelope. Contained demands are **inert** — recorded nowhere, pressing nothing. This is the step where the regime insight operates: a driver presses only when it crosses what the current position can hold, which is why most answers on most sheets press nothing, and why that outcome is a result rather than an anticlimax. An uncontained demand records a pressure: which axis, toward which value, at which scope, through which mechanism.

One check inside this step earns emphasis because a live derivation in Part II turned on it: pressures must be tested **in combination**, not only row by row. Two answers that are individually contained can jointly clear a bar neither reaches alone — a read volume that replicas would contain, meeting a read *shape* that replicas cannot serve, forces the projection that neither answer forced by itself. The pressure pass reads the sheet as a system, not as a list.

Step 4 — resolve. Pressed axes move by the selection rule: cheapest containing value, fewest new mechanisms, narrowest containing scope, rungs before moves, scope exclusion before hardening. Then the recovery axis is decided per effectful operation from the domain-shape facts: inverse exists and consistency must be restored — compensate; degradation tolerable and boundable — degrade and continue; the operation reshapes so the failure cannot occur: design it out, and check this option first.

A word on order, because the walk visits axes one after another and a reader is right to ask whether a different order lands elsewhere. It does not, and the reason is Chapter 3's own definition: selection is *the cheapest vector whose capability envelope contains the demands* — a property of the whole vector, not a sequence of local moves. Which axes move, and toward which values, is fixed before resolution begins: pressures are recorded in step 3, from the demands against the starting vector, before any axis resolves — so no axis's resolution creates or removes another's pressure. What remains is choosing values, and *fewest new mechanisms* is counted over the finished vector, not tallied as the walk proceeds — a bus introduced for one axis is free for every other that can ride it, whichever order they were written down. The walk is the efficient route to a minimum that does not depend on the route; with six axes, a genuinely close case is small enough to settle by enumeration.

Step 5 — verify. The derived vector faces the exit gate — the next chapter — where every guarantee must name its mechanism and every budget must survive arithmetic. Verification is part of the procedure, and treating it as a later phase is how wrong vectors reach production with their paperwork in order.

The conflict rule

Step 4 contains the procedure's one genuinely subtle move: what happens when demands press one axis in *opposing* directions. The order platform above is the common case, and the rule handles all cases with one test.

Different scopes → split. When the opposing pressures attach to different paths, data classes, or subsystems, the axis refuses to compromise — a middle value would underserve both demands while paying for neither cleanly. The system splits at the boundary between the pressures, each part derives independently, and the split pays the boundary cost from Chapter 3 in the open. Checkout gets its transactional store; analytics gets its cheap reads; the boundary between them gets a contract and an operational seam, priced and visible. Note what the split is *not*: a decomposition by org chart, by noun, or by fashion. The boundary falls where the pressures diverge, which is the only place it earns its cost.

Same scope → decompose further. When opposing pressures attach to the *same* scope, the conflict is usually a bundle that step 1 under-decomposed. The canonical case from Chapter 2: team ownership pressing toward many deployables, operational cost pressing toward one. Decomposed, ownership turns out to be satisfiable by module boundaries at zero deployment cost, and the conflict evaporates: a modular monolith serves the binding parts of both demands minimally. Same-scope conflicts are sent back through decomposition, and most of them do not come back. The loop terminates, and the floor is defined: decomposition ends at *mechanism level* — the entry gate's own language — because a demand stated as a mechanism requirement has nothing smaller to split into. A mechanism-level pair still opposing at one scope has nowhere left to go, which is the next case.

Still opposed after decomposition → contradiction. When demands that genuinely share a scope genuinely oppose after honest decomposition, the derivation halts and says so — and this is the procedure working, not failing. The answers contain a conflict the business has not resolved, and no arrangement of software resolves a business contradiction; it can only hide one, expensively. What a useful method produces at this point is Chapter 8's subject: the contradiction, detected mechanically, priced as a menu of which answer could bend and at what cost.

The pressure matrix

The derivation's working artifact is a table: one row per demand (and per domain-shape fact), columns for scope, mode, shape, the axis pressed, the value demanded, and the mechanism that justifies the pressure — or the single word *inert*. Part II fills several of these tables in earnest; the appendix worksheet carries the blank.

The matrix does two jobs. During derivation it is the workspace. Afterward it is the *traceability record*: every position in the final vector cites the rows that forced it, which means the architecture can answer questions about itself — why this axis moved, what would have to change for it to move back. Chapter 9 builds the audit on exactly this property, and the decision records that survive into the repository are these rows in prose.

One thing the matrix must never be: pre-filled. A matrix shipped with answers already pressing axes is a catalog with extra steps — this book's own warning label. The instrument is the *empty* table plus the discipline for filling it; every filled version belongs to one system on one date.

What the procedure refuses to decide

Honesty about the mechanics requires naming their edge, and the edge is principled rather than apologetic. The procedure decides nothing about the answers themselves: targets are the business's to set; the derivation owns consequences. It does not choose within genuine recovery ties: where the domain offers both an inverse and tolerable degradation, the ledger prices both and a human weighs restored consistency against liveness, a business preference wearing a technical costume. It does not resolve contradictions — it detects them and hands back a priced menu. And it does not pick products: the vector says *distributed shared store*, and whether that is one vendor or another is a later, different decision — one the vector finally makes tractable, since candidates now face a specification rather than a beauty contest.

Everything else — every "it depends" that used to end meetings, every style debate, every seniority tiebreak — is inside the mechanics. That was the design goal: mechanize the steps that never deserved judgment, so that judgment concentrates where it is genuinely owed.

What the procedure is, mathematically

Stated in another discipline's vocabulary, `next_step` is **constrained optimization over a finite design space**. The answer sheet supplies the constraints. The ledger defines the space — six axes, a handful of atomic values each, composable by scope splits. The objective is cost; the output is the cheapest vector whose capability envelope contains the demands. A reader trained in operations research will recognize the shape at a glance, and the recognition is correct — nothing in the word *derivation* denies it. Naming it buys three boundary statements that would otherwise stay implicit.

The space is chosen, not complete. The procedure selects among mechanisms the ledger prices; it cannot invent one. When the field produces a genuinely new mechanism — a storage class, a substrate, a containment trick like the coalescer that opened Chapter 3 — it enters as a ledger amendment: a new value or rung with its own provides/mechanism/costs entry, available to every derivation after that day and invisible to every one before. What the procedure guarantees is minimality *within the ledger it ran against* — a checkable claim — not minimality over all possible engineering, which no method can promise honestly.

The objective is not a scalar. Costs arrive in currencies that do not convert — money, operating attention, latency floors, blast radius — and the selection rule already refuses the conversion: comparison is by mechanism count precisely because prices resist arithmetic. Where counting fails to settle it — two vectors, each cheaper in a currency the business must weigh — the procedure has found a genuine tie, and ties sit on the refusals list above by design. Weighing one currency against another is

business preference; a formula that hid the weighing would be the fake precision this book exists to refuse.

The minimum is global over that space. The order argument earlier in this chapter carries the weight — pressures fixed before any axis resolves, mechanisms counted over the finished vector, enumeration as the tiebreak that six axes keep cheap. A greedy walk trapped in local minima is what the procedure was built not to be.

So the architecture does not follow from reality itself. It follows from the answers, over the ledger, up to the ties that come back priced — and each qualifier is visible on the page: the answers sit on the sheet with owners, the ledger sits in the appendix with costs, the ties return as menus. *Derivation* names what the optimization vocabulary leaves out: where the constraints come from. They are elicited facts rather than preferences, which is why two teams holding the same sheet and the same ledger reach the same vector, up to the ties the procedure refuses. Constrained optimization is what runs; derivation is why the result binds.

What the procedure hands over is not yet a verdict. Every position in the vector is a claim, and claims get checked — the last instrument in the toolkit exists to do exactly that, before anything gets built.

Instruments introduced: `next_step` (null vector → normalize → prune → press → resolve → verify) · the regime insight · combination pressure · the conflict rule and scope test · the contradiction halt · the pressure matrix · the procedure's refusals · the optimization naming and its three boundaries.