A FIELD MANUAL

Inside a Large ERP Program

The technical, business, and organizational anatomy, phase by phase

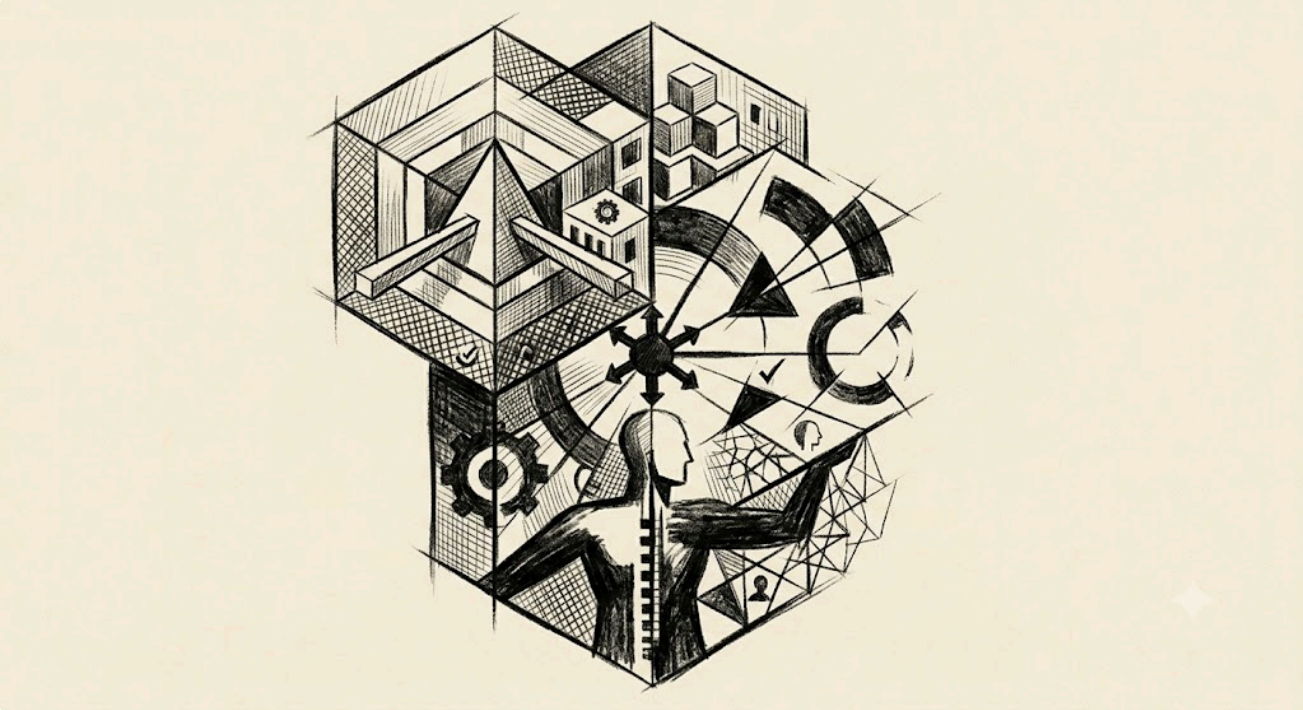
Michel Escoda

INDEPENDENT ARCHITECT & SAP FICO CONSULTANT

CONTENTS

INTRODUCTION

How to read this document



Most of what gets written about ERP programs comes from one of two places. Either a vendor is selling a roadmap, in which case the writing is optimistic by design. Or a consultant is writing from experience, but experience of only one end of the rope: the technical side that never quite grasps why the business keeps changing its mind, or the business side that never quite understands why the system can't just do what was asked. Both produce real insight. Neither produces the whole picture, because the person writing never had to hold both ends at once and answer for what happens when they pull against each other.

This document is written from having held both. Ten years on SAP FICO programs, split between the client side and the integrator side, reading the code as often as running the workshops.

This is not a sales pitch. Nothing here is trying to convince you to buy a platform, a methodology, or a firm. And it's not a configuration manual. You won't find step-by-step instructions for a transaction or a customizing node. What you'll find is an account of how a large ERP program actually runs, phase by phase, from scoping to run, written at the altitude of the person who has to make the program work: the program director, the PMO lead, the stream owner. Not the CIO deciding whether to fund it, and not the consultant configuring a single field. I also needed a place to write down what I've learned, to look back on it, and to share it with whoever might find it useful.

Every phase in this document gets read through the same three lenses, in the same order, every time:

Technical — what's actually being built, and what it costs structurally when it's built wrong.

Business — what the business has to decide, and what it doesn't realize it's deciding by staying silent.

Organizational — who ends up carrying the consequence, and whether that person had any authority over the decision in the first place.

That repetition is deliberate. A single chapter told through one lens would just be another opinion. The pattern that emerges from applying the same three-way cut to twelve different phases — scoping, selection, architecture, design, build, testing, authorizations, data migration, reporting, change, cutover, run — is the actual argument of this book: that ERP programs don't fail in the place anyone is watching. They fail at the seam between the three lenses, where a technical decision nobody flagged as a business decision quietly becomes an organizational cost months later.

Every program starts the same way, with a phase that looks light enough to hand to generalists.

CHAPTER 01

Scoping

Deciding what's in scope before anyone talks about software

Eighteen months into a program, well past the go-live of the core model, the group handed down a decision nobody in the original scoping room had any reason to expect. The in-house timesheet tool, homegrown and working and integrated, was out. A group-wide external solution was in, mandatory, and it had to land during the first rollout, which was already underway.

So we built a second core model. Not for finance, not for logistics. For timesheets. A whole design-build-test loop bolted onto a program that had already scoped itself, costed itself, and signed off on a plan. The plan had no room for it because the plan had never imagined it.

That timesheet decision felt like an external shock, a thing that fell out of the sky onto a well-run program. It wasn't. The specific tool, no, nobody could have called that. But the category of risk, the fact that a group can reach into a running program and reorder its scope, was visible to anyone who had lived through a multi-entity rollout before. The scope wasn't wrong. The problem was who sat in the room when the scope got written.

The declared scope is always the surface

Scoping produces a list of objects. "Financial close." "The target landscape." "Project time recording." Everyone nods, the slide gets approved, the business case quotes a number. What the list almost never contains is the depth of each object, and the depth is where the work actually lives.

Take the close. The scope said "financial close," and the cockpit promised a faster one. What it did not cover was the depth of the reclassification flows underneath that close: the manual postings that move amounts onto the right cost objects, the magnitude of them, and the fact that the automated program meant to replace those manual entries wasn't going to arrive until later in the year. So at go-live the close worked, in the sense that it closed. But it closed on manual reclassification entries nobody had scoped, carried by people doing them by hand, with the real automated flow pushed months downstream. The object was on the list. Its depth was not. And a close that runs on undocumented manual postings is not the close anyone thought they were buying.

The timesheet was the same failure wearing different clothes. "Project time recording" got scoped as a thing that existed and worked. Its depth — the fact that it sat on a homegrown tool the group could overrule, the integration surface it would expose if anyone ever had to swap it out — never came up. You can

list every object in a landscape correctly and still lose the program, because the program gets killed by the depth of the objects, not by the ones you forgot to name.

That's the mechanism. Listing scope and scoping its depth are two different acts, and most scoping does only the first.

Who is in the room

So why does the depth go missing? Look at who writes the scope.

On the business side, the scoping team is almost always light and senior. You get heads of function, transformation leads, people who can speak to strategy and sign a slide. What you don't get, reliably, are the doers — the management accountant who knows exactly which reclassifications run at close and why, the accounts-payable clerk who knows the three exception flows that don't fit the standard. The people who hold the depth in their hands aren't in the room where the depth is supposed to get scoped.

The solution side mirrors it. Scoping is often led by management consultants rather than functional SAP consultants, and that distinction matters more than it looks. A management consultant can structure a workshop, build the business case, run the governance. What they generally cannot do is see the trap, look at a process and know, from having configured it before, where the standard breaks, which flow hides a depth the client hasn't mentioned, which "simple" object is one decision upstream from needing a second core model. The pitfalls worth digging into during scoping are obvious to someone who has lived a close. They're invisible to someone whose deepest contact with the system is a fit-gap template.

Put those two together and you get the thesis of this chapter, stated plainly. The composition of the scoping team is the real scope. Staff the room with senior generalists and you scope the senior, generalist layer, by construction. The blind spots aren't random. They're exactly the things a doer would have caught and a generalist can't.

And the people who scope don't stay

There's an aggravating factor that turns a staffing mismatch into a structural one.

The organizational consultants who run scoping tend to leave early. They cover the program, hand over the deck, and roll off before build starts. Which means the people who decided the scope aren't the people who'll live inside it. No feedback loop. No accountability of the scoper to what they scoped. By the time the depth surfaces, eighteen months on, and the plan has no room for it, nobody who made the surface-level call is still around to answer for it.

Compare that to how it should work and rarely does. The person who scopes a flow ought to be close enough to its delivery to feel the consequence of getting it wrong. When scoping is a phase that specialists pass through on the way to the next engagement, that loop is cut at the design stage. The scope gets written by people optimizing to finish scoping, not to deliver the thing they scoped.

What scoping decides technically without knowing it

At scoping you build nothing. No system yet, no transport, not a line of config. Which makes it tempting to treat scoping as a pre-technical phase — strategy and business cases now, the real work later. That's the trap.

Scoping makes technical decisions constantly. It just makes them by default, through omission, by people who don't recognize them as technical. Every peripheral application that doesn't make the scope is a technical decision: you've decided, silently, that there's no integration to build there. Until there is. Every interface left unmapped, every local-specific flow nobody surfaced, is a technical commitment made by not making it. The timesheet swap was a technical decision the scoping room made without knowing it, the day it assumed project time recording was settled and would stay settled.

These aren't risks that appear during build. They're decided at scoping and merely discovered during build. The difference matters, because you can't fix at build a decision that was foreclosed at scoping by people who didn't know they were deciding anything.

What it costs when the off-scope comes back

When the depth surfaces, and it always surfaces, the cost doesn't land where the business case can see it.

It lands as overtime first. The team absorbs the unscoped work by working longer, because the plan has no slack. Then it lands as a degraded go-live: things ship before they're ready, because the unscoped depth ate the time that would have made them ready. And then it lands in the place no scoping slide ever models — the people. Sustained overtime on a program that keeps discovering work it should have known about produces stress, and stress on a delivery team produces turnover. You lose the people who hold the institutional knowledge, mid-program, at the moment you can least afford to.

That last cost is the one that compounds. A missed interface is a line item; you can quantify it and recover. An eroded team isn't a line item. The judgment walks out the door inside the person, and the next hire scopes from an even shallower base than the one that caused the problem.

The phase that decides everything, staffed like it decides nothing

The instinct is to treat scoping as the light phase, the strategic warm-up you hand to generalists before the serious people arrive for build. That instinct is backwards. Scoping is where the real scope gets decided, which makes it the phase that most needs the people who can see the depth: the doers from the business who know what runs underneath each named object, and at least one functional consultant who has configured the thing before and knows where it breaks. Not after the scope is written. In the room while it's being written, and still accountable through delivery instead of gone by build.

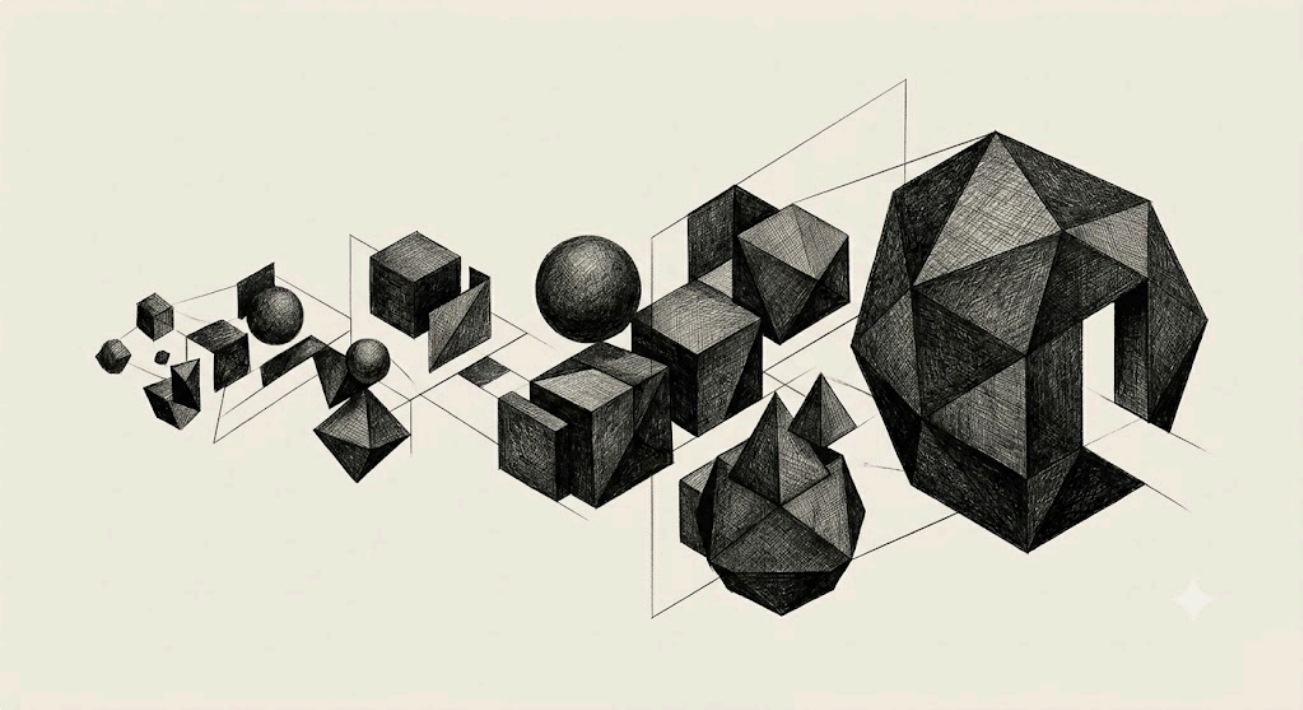
The scope you can read on a slide describes the surface. The real scope is the depth nobody wrote down, and the only defense against it is to put the people who hold that depth in the room before the slide gets approved. Get the room right and the periphery — the unmapped interface, the reclassification depth that runs on manual postings, the group-level swap that's always possible — all show up as questions during scoping instead of as shocks a year and a half later.

You decide the fate of the program here, before anyone has opened the software. Most programs decide it by accident, through whoever they happened to put in the room.

Once the real scope is on the table, depth and all, the next decision is who you choose to deliver it with, and how you write the contract that binds them.

CHAPTER 02

Selection



A selection process produces a scoring grid. The grid lines up the candidates against weighted criteria and promises to tell you, objectively, which integrator is best. It's the artifact everyone trusts, because it looks like rigor.

Here's the problem. When two finalists are close, and at the end of a serious RFP they always are, the grid stops deciding anything. What decides is somewhere else, in the part the grid was never built to see. The client who believes the grid protected them has already made their first mistake of the program.

What the grid actually measures

I've sat on the integrator side of an RFP. Putting the response together is an exercise in staging. You don't show the team that will run the project; you show the best consultants you have. You show the best reference projects, the ones as close as possible to this one. You defend the shortest timeline you can stand behind, build the most convincing deck, and if you can, you put a small proof-of-concept on the table to prove you already master one piece of the problem.

Every serious competitor does exactly this. That's the point most clients miss. The grid measures the quality of that staging, not the capacity to deliver, and all the credible finalists know how to make it excellent. So the grid does something the client never intended: it equalizes the strong candidates instead of separating them. It's good at one job, eliminating the obviously unfit, and useless at the one the client actually cares about, choosing between two good firms.

So what breaks the tie? Two things, and neither is on the grid. The first is price, because there's usually one player who comes in aggressively cheap. The second is the firm's flexibility, its willingness to bend to the client's calendar, to absorb a schedule that suits the client more than itself. That's what the close calls turn on. Not the competence the grid spent forty pages scoring.

What the demo doesn't show

The same mechanism runs through every demo you'll watch, and it's worth naming the general rule before the example: what's demonstrable in pre-sales is precisely what was prepared to be demonstrated. The happy path is the product. Everything off that path is invisible until you're the one walking it.

The cleanest illustration I have isn't an integrator demo, it's a vendor one, and the distinction matters. When SAP Fiori was new, it was presented as a mature, finished experience, in the press and in front of clients. It looked ready. Then we implemented it. Every new Fiori app we put on the launchpad came with its own problems: setup, authorizations, integration with the backend, basic functions that should have worked and didn't. Buttons that did nothing. A good share of the apps were buggy, some mildly, some seriously enough that we fell back to the old technology to get the work done.

Two things about that example. First, it's a product-maturity gap, the vendor overselling its own roadmap, not an integrator trap. They're different failures. The vendor oversells how finished the product is; the integrator oversells how completely it has mastered it. Both leave you with the same surprise on the project, but you defend against them differently. Second, Fiori today is not the Fiori of that story. It's mature now, and strategically unavoidable — S/4HANA Cloud ships with no SAP GUI at all. But the ground is still mixed: as of late 2025, surveys put a majority of organizations running mixed UI environments, with only a small fraction fully transitioned.

The lever for the client is the same in both cases. Don't accept the showcase. Ask to see the thing that doesn't work cleanly yet, the migration of your actual legacy data, the edge case, the unhappy path. A demo that only shows the prepared scenario has told you nothing you can plan against.

The contract decides the behavior

Now the part most clients get backwards. The contract isn't insurance. It's an incentive structure, and the structure you choose will shape how the integrator behaves for the next two years.

The common belief runs like this: a fixed price (forfait) protects the client and puts the result risk on the integrator, while time-and-materials (régie) is the dangerous one, where the firm just bills hours and has no reason to be efficient. It's a reasonable intuition. It's also mostly wrong.

Start with the fixed price. It pushes the integrator to go fast, to stay inside the envelope and protect the margin. Fast isn't the same as careful. When every extra day eats the firm's profit, the pressure is to close items and move, not to do the quiet, unbilled work that makes a configuration durable. The incentive is speed, and speed has a quality cost you'll pay later.

Now time-and-materials, which is more aligned than its reputation. The more efficient and genuinely useful the firm is, the more the client values them, and the longer and healthier the engagement. Efficiency serves the integrator's own commercial interest directly, because a satisfied client renews. That's a cleaner alignment than people credit it with.

There's a second, more structural reason, and it's the one almost nobody contracts for. The integrator's people are not "fixed-price." A consultant can leave. A freelancer can walk off the engagement. So a fixed-price contract locks the firm into a commitment while its actual workforce stays mobile underneath it. That mismatch, rigid contract over fluid team, is a permanent source of tension on milestones and deliverables: the team shifts, but the commitment can't shift with it, and the gap lands on the schedule. Read the contract for what it really is. Fixed price doesn't buy you peace of mind. It buys you a particular speed and a particular rigidity.

Milestones and penalties: a lever, not a switch

This is where clients most often comfort themselves with a clause that won't do what they think.

Penalties don't fire like a switch. In any real conflict between an integrator and a client, the penalty clause opens a long negotiation over the fine print, not an automatic deduction. The reason is that the operational reality of a complaint is almost impossible to prove contractually.

Here's the mechanism. You write into the contract that ten deliverables are due by a certain date. The ten arrive on time. Contract respected. But the quality? If you judge them thin, hollow, not at the level you needed, how do you prove it? "Delivered" is binary and checkable. "Delivered well" is a judgment, and a judgment doesn't trigger a penalty, it triggers an argument. So you debate. That's the real function of milestones and penalties: they're a lever in that negotiation, a way to bring the firm back to the table and apply pressure. They are not an executable guarantee. The client who signs believing the clause will protect them automatically has misunderstood the tool in their hand.

The team sold is not the team that arrives, and the lock cuts both ways

Of everything in this chapter, this is the one to internalize, because the lever here is mutual and clients usually grab only half of it.

The team that was sold almost never arrives intact. It's rare, genuinely rare, and an informed client doesn't expect otherwise. The strong profiles in the bid were there to win the bid. Delivery staffs from who's free when the project actually starts.

That gives the client a lever it underuses. If the seniority of the profiles doesn't match what was sold in the bid, you can, and should, require the integrator to realign the consultants on the project to what the appel d'offres committed to. That's a contractual right, and most clients let it sit unused.

But the lever runs the other way too, and this is the part that surprises people. The client does not own the consultants. In a tight digital market, good consultants are not obliged to stay. If a client treats them as interchangeable subcontractors, refuses to work in close collaboration, lets the working conditions degrade, they leave. They have somewhere else to go. So the lock is mutual: you can demand the right profiles, and you also have to be worth staying for. Team composition isn't a contractual asset you secured at signature. It's a relationship you maintain, in both directions, for the life of the program.

What you actually chose

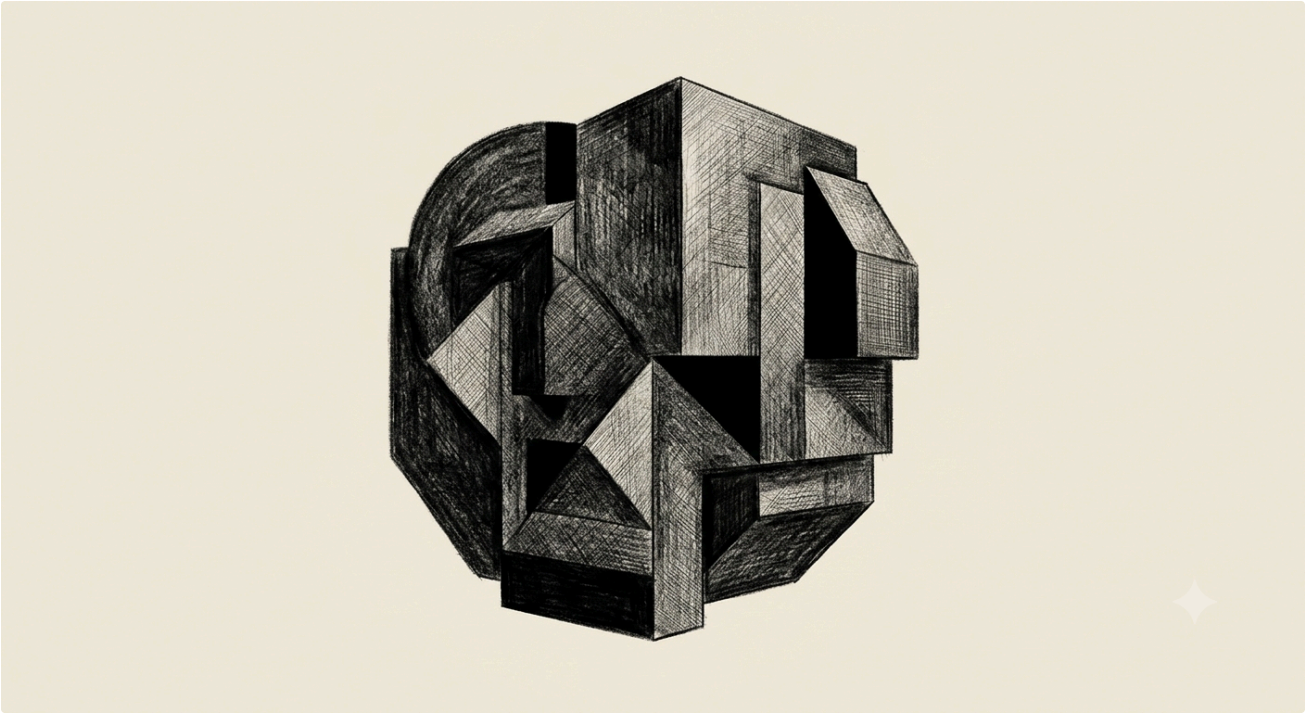
Selection isn't won on the day you sign. That day, you didn't buy a guaranteed outcome. You chose who you'll be steering alongside, through all the blind spots your grid never scored: the price pressure, the demo gap, the incentive baked into the contract, the team that quietly changed. The contract won't run any of that on its own. It sets up incentives you'll have to read and pull on for the whole program.

And once the signature is dry, the work moves to what gets built.

CHAPTER 03

Program architecture

Designing the landscape, the integrations, and the extensions



Architecture is the only part of an ERP program whose mistakes are invisible at the moment you make them. A bad scope decision shows up in the next steering committee, a bad integrator in the first delivery. A bad architecture decision shows up two years later — in a performance incident at month-end, or in an upgrade that runs three times longer than the vendor promised, long after everyone has forgotten who decided what, and why.

This chapter walks through the three places where that happens: the landscape, the integration layer, and the extensions. Three different technical subjects, one identical failure mechanism.

The landscape: complexity is not a virtue

I have worked on landscapes in every shape you can imagine, and the pattern is consistent: the more environments and the more transport tracks you add, the more conflict you create. Complexity in a landscape does not buy you safety. It buys you surface area for things to collide.

The textbook setup is a three-tier line — development, quality, production — with a sandbox in front. The version I have seen cause the most pain is the doubled landscape: a full project line and a separate maintenance line, the maintenance track rejoining the main track in quality. On paper it looks responsible. You keep your run-the-business fixes away from your change-the-business project. In practice, every retrofit becomes a negotiation, every release becomes a sequencing problem, and the moment you layer in a migration or a version upgrade, you are managing collisions between objects that two different teams touched for two different reasons.

Then there is the dedicated integration environment — the one inserted between development and the UAT environment specifically to test interfaces. It sounds rigorous. It rarely works as intended. You cannot meaningfully test an interface without end-to-end data flowing through it, and that data lives in UAT, not in the integration box. So one of two things happens. Either you test the interfaces twice — once in integration with synthetic data that proves nothing, then again in UAT with real data — or the consultants quietly test in UAT and never mention to their manager that the integration environment went unused. I have seen both. Neither is the clean story that was sold when the environment was provisioned.

The setup I would defend is leaner: sandbox, development, UAT, quality, production. Drop the dedicated integration line. Separate project work from maintenance not with a parallel track but with project codes inside the transport

requests, so you can filter, sequence and retrofit by code rather than by maintaining a second physical line. You lose nothing real and you remove an entire category of collision.

That is an opinion, and I will own it as one. There are landscapes complex enough — multiple concurrent projects, heavy regulatory separation — where more tracks genuinely earn their cost. But the default should be the lean line, and the burden of proof should sit with whoever wants to add the next environment, not with whoever wants to keep things simple.

The integration layer: the interfaces we built three times

On one program, for a large consulting firm, we built the interfaces three times.

First as point-to-point. Then we moved to SAP BTP. Then the group decided on Google Cloud Platform. Three architectures, three rounds of rebuilding, on the same set of flows. The delay was real, the team was overloaded, and by the end nobody could say cleanly why the second choice had been wrong, only that a third one had now replaced it.

Here is the thing worth sitting with: none of those three choices was stupid on its own. Point-to-point is what you fall into when there is no platform decision at all. BTP is a coherent choice — it is SAP's own integration platform, it sits natively against S/4, and if the group is already on RISE there are credits that may already cover it. GCP is also a coherent choice, made for a different reason: it is more open, less tied to the SAP stack, and for a group with a large estate of non-SAP applications that openness is more attractive over the long run. That is my read,

and a BTP advocate would push back — native integration and a single SAP-governed monitoring view are real arguments on the other side. Reasonable people land differently.

The point is not which platform wins. The point is that the criterion that decides between them — native integration versus openness — is knowable at design time. It could have been argued once, written down once, and settled once. Instead it was settled three times, each time by conviction in a meeting, each time at the cost of a full rebuild. The waste lived in the missing decision, not in any one of the choices.

On the substance, the direction of travel is not really in dispute anymore, and you do not have to take my word for it. SAP has set end of mainstream maintenance for on-premise Process Integration and Process Orchestration — the classic middleware backbone — at the end of 2027, with extended maintenance available until 2030, aligned with the NetWeaver 7.5 timeline. Whatever you think of the replacement options, the old on-premise middleware has a published expiry date. Point-to-point as a default, and aging on-premise middleware as a home for it, is a position with a clock on it.

What a platform buys you, when you do pick one, is worth stating plainly because it is the actual engineering argument under the marketing. You mutualize transformation logic instead of re-implementing the same mapping rule in five different interfaces. And you push computation into a back-end platform built for it, rather than loading it onto user-facing systems that were never meant to carry that weight. That is the mechanism. It is also why the trend is real and not just vendor noise — though when an integrator tells you their platform "replaces legacy middleware," remember they sell the migration, and read the claim accordingly.

Extensions and clean core: a strategy against deep habits

"Clean core" is one of the most repeated phrases in SAP conversations and one of the least precisely understood. Ask five people on a program what it means and you get five answers — an architecture principle, an upgrade strategy, shorthand for "less custom code," or a nod from someone not quite sure what they just agreed to.

The official definition is more reasonable than the slogan. SAP describes a clean core as keeping the ERP core as close as possible to the standard, vendor-delivered state, and handling customization through decoupled, upgrade-safe extensions — on BTP, side by side, rather than as modifications inside the core. Notably, SAP itself is explicit that a clean core does not mean zero custom code; it means custom code that is decoupled from the core so upgrades stay stable. As a design philosophy, that is defensible. I am not going to argue against the principle.

What I will argue against is the idea that the principle survives contact with a real program. Clean core is a marketing argument that does not hold up against the program committee, the finance department, and twenty years of deep habit. It can work for a client that has never run SAP — a greenfield estate with no legacy processes to defend. But for a company that has run SAP for two decades, where the business has built its identity around the way the system behaves, you can push to stay close to the core all you like; you will rarely convince the business directors to change how they work. The gravitational pull of "this is how we have always done it" beats an architecture slogan every time it comes to a vote.

For an example closer to the system — because this runs from the governance vote down to the code itself — most of what I see is still classic RICEFW, ABAP development much as it was built before. In-app extension exists, but it shows up less often than you would expect, because the standard leaves comparatively little room to maneuver. The clean-core future is real on the slides. On the floor, the ABAP is still being written.

The code nobody wants to take over

There is a specific failure I see because I read the code, and most reviews do not.

When a new team takes over an existing program, the developers almost never make the effort to move to current best practices — CDS-based modeling, modern extension patterns, the things that would actually pay off. The center of excellence gets inherited more or less identically. Sometimes that is a deliberate speed call: rewriting working code to a newer pattern is effort with no visible feature at the end of it. More often it is something quieter and worse — the new team has lost the historical knowledge of what the code actually does, so they dare not touch it. They build around it and leave it alone.

That is technical debt that was inherited, not created, and it is the most dangerous kind, because no one currently on the program feels responsible for it. On a system handling modest volumes you can live with it for years. On a program moving serious data, you cannot. The old patterns that were fine at pilot scale turn into performance problems at production scale, and the incident arrives at exactly the wrong moment — period-end, year-end, the close that cannot slip. By then the people who could have explained the code are gone, and you are debugging archaeology under time pressure.

This is the quiet cost of treating an inherited codebase as a black box. The debt does not announce itself. It waits for volume.

The decision that was never made

Step back from the three subjects and the same shape appears in all of them.

The landscape got an extra track because it felt prudent in the moment. The integration platform got chosen three times because the criterion was never written down. The extensions stayed in classic ABAP because clean core lost the vote it was never really allowed to win. The inherited code stayed untouched because no one owned the history. In every case the architecture decision was made fast, by conviction, in an arbitration meeting, without a decision matrix and without a trace. The result is application spaghetti — reworked partially when there is time, not reworked at all when there is not, and carried forward as debt either way.

The temptation is to call this a technical failure. It is not. The engineering choices, taken individually, were mostly defensible. What failed was governance: the discipline to name the criterion before the meeting, write the trade-off down, and make the decision once so it does not get made three more times at three times the cost. Architecture debt is a decision that was never made, compounding quietly until someone has to pay for it.

The same trap is waiting in the next phase, where the arbitrations are functional rather than technical but the mechanism is identical.

CHAPTER 04

Design

Trading off fit-to-standard against custom requirements

The program directive is never ambiguous. Stay close to the standard. Configure, don't develop. Keep the core clean. Everyone signs up to it at kickoff, and it goes on every steering slide for two years.

Then watch where that directive actually lives. It lives in the mouth of a consultant, in a design workshop, facing a business owner who has done the job one way for fifteen years. And that consultant has no power to enforce it. He can propose the standard. He cannot make anyone accept it. So the gap between the directive and its application is total: the principle is carried by the one person at the table who has no authority to apply it.

That gap is the subject of this chapter. Design isn't where fit-to-standard is won. It's where it's argued by people who can't decide it.

How a design actually gets made

Design isn't one meeting. It's a sequence of workshops.

The first workshops collect. You gather the business need and the historical way of working, the way it has always been done, exception flows and all. Then the consultants go away and think. They come back with a proposed design, built to land as close as possible to what the SAP standard can do, and they put it in front of the global process owner or the key users.

What you hope to see at that moment is flexibility, a business willing to bend its habits to work the standard way. What you usually get is the opposite. Habit blocks. The business owner looks at the standard flow and says it doesn't fit, asks for a specific alternative that matches the way they already work. And now the machine turns: the consultants go back to the developers and the project management, build a costing and a schedule for the requested specific, and that package goes up to the arbitration committee.

Notice what just happened. The workshop didn't decide anything. It surfaced a need, hit a wall of habit, and packaged a request. The actual design decision moved one level up, to a room the key users aren't in. The workshop instructs. It doesn't rule.

The authority is somewhere else

Here is the structural fact most programs never say out loud. Only the arbitration committee can force the business onto the standard. The consultant can't. The program directive says "stay close to the standard," and the person repeating it in the workshop has no lever to make it true.

There's a human cost to that gap. The consultants wear themselves out trying to hold a line they have no authority to hold. They push the standard, argue for it, fight the business owner to respect the program's fit-to-standard directive, all while the balance of power is against them. The sane way to work is to stop fighting battles you can't win at your level: propose the standard, and at the first real resistance, escalate. Raise the point, hand it to the committee, let the authority that exists settle it. The consultants who don't learn that, who keep absorbing the friction personally because they feel responsible for a directive they can't enforce, are the ones who burn out. A lot of them do. I did.

So the committee is where it's settled, and the committee is a confrontation. The business line comes in defending its specific. The program leadership is supposed to block it, to say no, this runs in standard, you don't get the development. That's the design, working as intended: the program director holds the line the consultant couldn't.

But the line only holds if someone at the right level actually defends it, with the right argument. If the program leadership doesn't hold, or doesn't have the technical case to show the committee that the gap really does run in standard, the gap passes. And it passes wearing the authority of a committee decision, which makes it final. And that's where the consultant role really is in the end: presenting at the committee the best arguments, slides, and pros and cons to push it toward the standard.

That's the part to sit with. Fit-to-standard isn't a property of a good design. It's the outcome of a power struggle in a committee room. No committee that holds the line, no standard, no matter what the directive says or how many slides repeat it.

The daily cost transfer that never stabilized

Let me give you the cleanest example I have, and it comes from an intercompany flow.

The business wanted the costs of an internal supplier to land in the internal customer on a daily basis. The reason was partly habit — their previous ERP, an Oracle system, had worked that way, and they didn't want to change. But it wasn't only habit. They wanted to be able to invoice the external customer off the internal costs without waiting for the internal supplier's invoice, which arrived less often. That's a real business need, not a whim, and it's worth saying clearly: the people asking for this had a defensible reason.

The problem is how SAP moves cost between companies. In SAP, an intercompany cost transfer goes through internal invoicing. If you want the cost to move, you raise an internal invoice. The business didn't want to multiply invoices, and didn't want to lower the transfer frequency to match an invoicing rhythm. So the request that went up was a specific: a custom program that would post the costs daily, revenue included, through an FB01 posting. Which meant, under the hood, collecting all the costs, collecting the associated revenue, merging the two, and then determining the analytical object in the internal customer, every day, by code.

The committee approved it, and the build began. After months of development, the program still wasn't reliable, not quickly anyway. There were too many sub-cases, edge conditions nobody had been able to anticipate in advance, and each one needed its own handling. In the end the daily ambition collapsed back to weekly invoicing, the thing the standard would have given them in the first place. Months of custom development to arrive, eventually, near where standard would have started.

The decision to attempt it took one committee. The bill, months of build, then the climbdown, landed on the people who came after.

Who decides is not who pays

That example isn't a one-off. It's the mechanism, and the mechanism is simple enough to state in one line: the person who wins the gap is almost never the person who pays for it.

The business that secured its daily cost transfer had a real reason to want it, and it was never going to concern itself with the technical debt the custom program created. Why would it? It asked for something that served how it needed to operate. The debt — the months of build, the unstable program, the edge cases multiplying in code, the eventual climbdown — lands somewhere else entirely, on the technical team, on the run organization, on whoever inherits the program afterward.

The calendar version of this is "design validated by people who'll never run the system." That's true, but it's not quite the heart of it. The business owner does run the system, every day, with the process they fought for. What they don't carry is the debt. The convenience sits with the decider; the consequence sits

with the people who maintain it in the long run. And note, the business asking for something reasonable doesn't change the outcome — the requester here wasn't being unreasonable, and the debt landed all the same.

Which tells you the question the committee should be asking, and usually isn't. The committee asks: is this gap justified for the business? And the answer, from the business's side of the table, is almost always yes, because the business is sincere, the specific really does suit how they work. That's the wrong question. The right one is: who carries the debt this gap creates? Because let's raise another misalignment while we're here: the program director is usually not the manager of the run team that will clean up his potential mistakes afterward. And a second question the cost-transfer case makes obvious: has anyone actually established that the specific can be built reliably, or are we approving an ambition?

What design really decides

Pull it together and design stops looking like a workshop phase. The workshops collect and instruct, but they don't decide, and the consultants who run them hold a directive they can't enforce. Fit-to-standard is won or lost one level up, in the arbitration committee, on a single question the committee too rarely asks: is the person winning this gap the same person who'll carry its debt?

When the answer is no, and it almost always is, the program buys years of debt with minutes of meeting time, and pays for it long after everyone in the room has moved on. Get that question into the committee, make the cost visible to the people deciding, and most of the bad gaps die where they should, before they reach the code.

Once the design is arbitrated, someone has to build it. And whether it gets built well, or merely gets built, is its own story.

CHAPTER 05

Build

Turning design decisions into configuration and code

The build phase is traditionally approached as a standardized factory floor. Management visualizes a clean pipeline where competent teams line up discrete objects: a piece of standard configuration, a block of custom ABAP code, a set of test scripts. This theoretical framework is comforting to PMOs, but it hides the exact mechanism of project failure.

The build is not a monolithic manufacturing block. It is a critical, fragile chain of handoffs: from the business to the functional consultant, from the functional consultant to the developer, from the developer to the tester, and eventually back to the business. It breaks at the seams between these links far more often than inside any single box.

When a program transitions to a distributed delivery model (offshore or nearshore factories), the problem does not change its nature — it changes its scale. Every seam is suddenly stretched across a time zone, a cultural boundary, and a Jira ticket. An enterprise ERP program is never judged by the isolated quality of its links; it survives or dies based on the structural solidity of its seams.

The technical seam: the fiction of the standard Agile user story

The first structural seam opens between the functional design and technical execution. In modern ERP rollouts, integrators frequently push for a pure "Agile factory" approach, relying on lightweight user stories to drive offshore development. For an enterprise core system like SAP S/4HANA or Oracle Cloud, this method is highly dangerous.

A user story stating "As a credit manager, I want to block high-risk invoices" is completely useless to a developer sitting five thousand miles away. It lacks the architectural context required for complex integrated systems. When left with vague parameters, an offshore factory will inevitably build a decoupled, custom silo that ignores database locks, standard memory blocks, or core table joins.

To bridge this technical seam, you must reject the shortcut of conversational specs and enforce a rigid, structured functional appendix. Every technical specification must explicitly detail the precise database tables and standard field structures involved (BSEG, ACDOCA and the like), the exact standard SAP BAPIs, BADIs, or extension points to be used, protecting the "clean core" principle, and the transaction volume expectations and specific database indexing needed to prevent catastrophic locks during month-end closes.

Landscape governance and the transport matrix

The second technical failure point during build is the anarchy of uncoordinated system transports. In multi-stream programs (finance, supply chain, logistics), consultants configure changes simultaneously in a shared development environment. Without ironclad governance, one stream's transport package will silently overwrite another stream's settings, resulting in constant regression loops during unit testing.

To secure this seam, direct configuration in shared test clients must be frozen. You must establish a weekly cross-stream transport synchronization board. Before any configuration package or custom workbench object is promoted to the QA environment, it must be cross-checked against an automated dependency matrix. No transport affecting global organizational elements — company codes, charts of accounts, global plant matrices — can be moved without multi-stream sign-off.

The business seam: the latency of the unanswered query

The build phase cannot operate in a vacuum; it requires continuous business arbitration. As developers write code and consultants configure processes, they inevitably encounter gaps that the initial design phase missed. This is where the business seam is tested.

When a developer in an offshore factory hits a logical contradiction in the spec, they open a ticket. If the local functional team takes three days to review it, and the business process owner takes another week to provide a decision, that single ticket introduces ten days of latency. Multiply this by three hundred active development objects, and your build schedule is mathematically destroyed.

The offshore factory does not wait idly. To meet their contractual execution metrics, they will either guess the business logic — introducing deep functional debt — or pause the development and reassign the resource. When the answer finally arrives, the developer has moved on, and re-boarding them costs another three days.

Securing the business seam requires a dedicated, daily arbitration ritual. The functional stream leads and business counterparts must commit to an ironclad twenty-four-hour SLA: any technical query raised by the factory must be resolved or formally arbitrated within a day. And if a business decision cannot be made instantly, the functional lead must enforce a temporary "standard path" configuration rather than allowing the development block to stall — no placeholders.

The organizational seam: the junior trap and stream authority

The final, and most sensitive, seam is organizational. It defines how human beings communicate across boundaries of geography, seniority, and contractual incentives.

The typical integrator model relies heavily on pyramid leverage: a highly experienced, articulate partner sells the project, a couple of senior architects design it, and an army of junior offshore developers actually builds it. This creates an immediate organizational seam. If a junior developer is left to code complex financial reclassification logic without direct senior oversight, they will deliver an object that passes its basic "happy path" unit test but crumbles under volume or real-world integration data.

You cannot prevent the use of junior resources — it is the economic reality of large system integrators. However, you must actively police the seam. Enforce mandatory, peer-reviewed code checks and buddy-programming systems where a senior architect signs off on the technical architecture before a single line of custom code is finalized.

The organizational seam also breaks at the top. When a program timeline tightens, the program director is tempted to short-circuit the hierarchy. Seeing a delay in the finance stream, the director may bypass the finance stream lead and speak directly to the technical factory manager to demand acceleration.

This is a structural mistake. By short-circuiting the stream lead, the director destroys their authority and breaks the transmission of functional command. The factory manager accelerates execution by cutting corners on documentation and unit testing, and the stream lead stops taking accountability for the final quality. The program director's job is not to manage the technical factory; it is to protect the stream lead's authority, ensuring they have the space and the senior resources to enforce quality control at the seam.

What the build really is

When you step back from the spreadsheets and tracking tools, the build phase stops looking like a technical execution stage where teams simply produce software objects. It is revealed for what it truly is: a continuous chain of human and system handoffs. The configuration and the custom code are rarely where the project fails; the seams between them are.

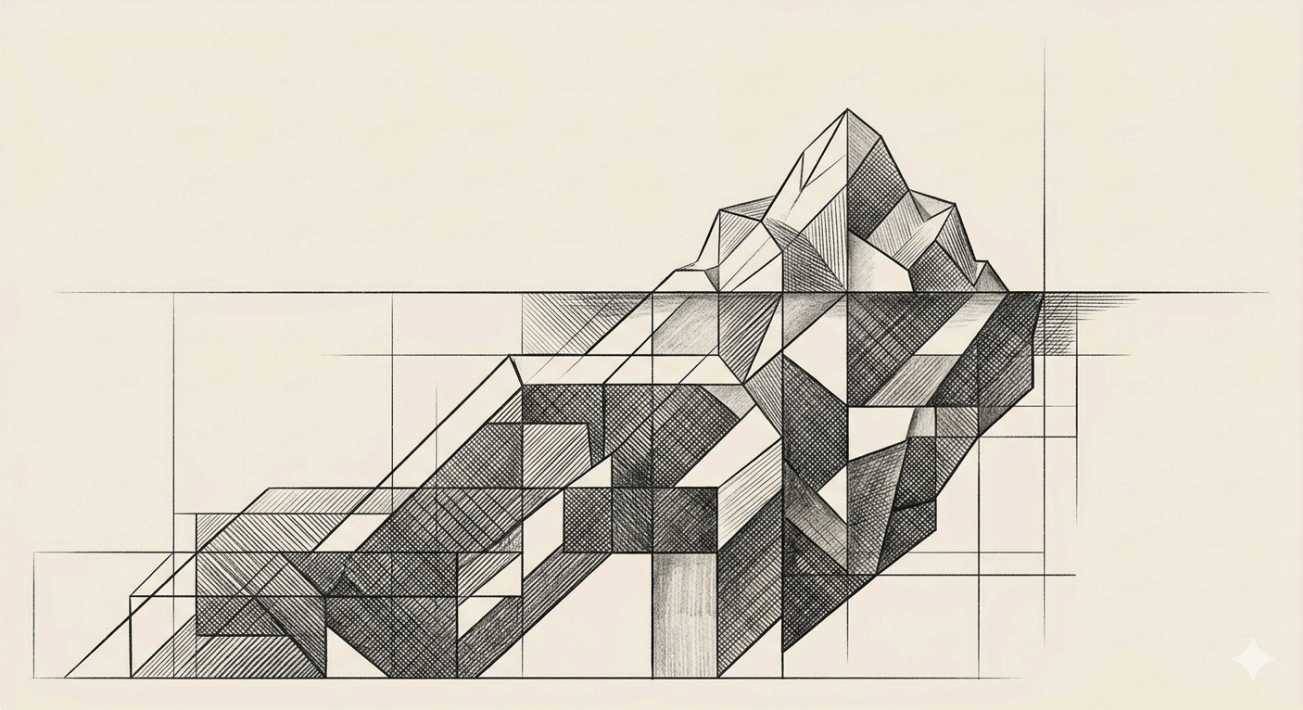
Distributed delivery does not invent these structural flaws; it merely exposes them by injecting geographical distance, cultural friction, and ticketing latency into the most fragile parts of the architecture. Running a successful build is not about fighting for flawless human perfection in every box. It is about protecting the arrows between the boxes: a functional specification that leaves no room for guessing, an arbitration process that answers in hours rather than weeks, a landscape governance that stops transport conflicts before they happen, and a senior engineering oversight that never lets a junior code alone.

Whatever is built during this phase, whether sound or fragile, must now be rigorously proven before the enterprise can risk its operations.

CHAPTER 06

Testing

Proving the system works before you turn it on



Testing presents itself as the objective part of a program. After all the judgment calls of scoping and design, here at last is a measure: the system either passes or it doesn't. A green UAT, and you're ready.

It doesn't work like that. A green UAT does not say "the system works." It says "what we decided to test worked." And everything that matters is hidden inside that decision — what got tested, who was responsible for it, which defects were allowed to wait — because that decision is made earlier than people think, more quietly than they'd like, and often for worse reasons than they'd admit. Testing

isn't a measurement. It's the only insurance a program buys itself, and like any insurance, its worth is decided when the coverage is written, not when the claim comes in.

The cascade and the level that gets sacrificed

The textbook cascade is well established, and worth stating once so we can talk about where it bends. Unit testing checks each piece of code or configuration on its own. System integration testing, SIT, checks that the modules actually talk to each other. UAT puts the business in front of real scenarios on production-like data. Regression confirms that what worked before still works after a change. Four levels, each catching what the one below it couldn't.

On paper. In practice, when the schedule tightens, and it always tightens, one level gets sacrificed before the others, and it's almost always SIT. The reason is visibility. UAT has a business audience watching, so cutting it is politically loud. Unit testing is the developers' own floor, cut it and the build visibly falls apart. Regression has the weight of "don't break production" behind it. SIT has none of that. No business audience, no immediate go-live pressure, just consultants checking integration in a room nobody important is watching. So it's the first thing pared back when days go missing.

Which is the worst possible choice, because SIT is the level that tests the seams between modules, and tightly-coupled SAP breaks at the seams more than anywhere else. Cut SIT to save a week and you've reduced your coverage in exactly the place the risk is highest and the visibility is lowest. The insurance gets thinner precisely where the claims will come from, and nobody in the room feels it happen, because the thing you stopped testing is the thing whose failure shows up two months later.

Two things called UAT

Here's a fight I've watched up close, inside a large consulting group, between managers who could not agree on what a UAT even is.

One camp held that UAT is essentially a copy of the SIT. The business sits down and replays the scenarios the functional consultants just ran, confirming the same flows give the same results. The other camp held that SIT only tests what the consultants understand of the flow, and that the business should build its own UATs from its own view of how the work really goes. The two definitions sound close. They are not. They certify completely different things.

My own position is the second one, firmly, and the reason is about responsibility. A UAT that just replays the SIT lets the business off the hook entirely. It never forces them to go looking for what the system will actually be like once they're living in it after go-live. They confirm the consultant's flow and sign. Whereas a UAT the business builds itself forces them to imagine their own real day, the awkward customer, the split delivery, the credit note against an invoice that was already partly paid, and to test whether the system survives it. The first kind proves the consultant can re-run their own demo. The second kind proves the system holds up against reality. Same three letters, two utterly different insurance policies, and most programs never notice they bought one when they thought they were buying the other.

This is the heart of why a green UAT can mean so little. If "UAT passed" was generated by the copy-the-SIT model, the certificate attests to almost nothing about the post-go-live world. The coverage was written narrow, and no one read the policy.

The happy path certifies only good weather

The narrow policy has a signature shape, and the testing literature even has a name for the trap: testing only the happy paths creates false confidence. Feed the system the clean case, the one everything was designed around, and of course it passes. The cases that break a system are the ugly ones — the null value, the out-of-sequence event, the quantity that doesn't divide evenly — and those are exactly the ones a happy-path script never contains.

This connects straight back to the two UATs. The copy-the-SIT model is a happy-path machine by construction, because the consultant's SIT scenario is the designed case, the one built to work. Hand that to the business as their UAT script and they walk the golden path, tick every box, and certify a system that has never once been shown a problem. A green UAT built only on happy paths is an insurance policy that pays out only when the weather is good, which is to say when you didn't need it. The whole value of a test is in the cases you were afraid of, and a script handed down from the people who built the thing contains none of them, because they built it not to have any.

Triage: deciding what the policy won't cover

At some point every program decides, explicitly or not, which defects it will fix before go-live and which it will carry past it. This is the moment the coverage gets its exclusions written, and it deserves to be done well.

Done well, it has a real criterion, and mine is concrete. A defect can wait past go-live when three things hold: the close is only weakly impacted, invoicing remains possible, and the manual workaround is manageable. Those three — close,

invoicing, workaround — are the load-bearing functions of a running business. If a defect leaves all three standing, the business can live with it for a while, and deferring it is a legitimate call against real risk.

Done badly, the same triage runs on the wrong fuel. A defect gets stamped "minor" because the schedule can't absorb a fix, or because the stream that owns it has more political weight than the stream that reported it, not because the risk is actually low. And here's the mechanism that makes deferred defects come back to bite: the manual workaround that justified the deferral as "temporary" almost never is. It quietly becomes permanent, and the cost of it lands on the run team, every close, every month, as a tax nobody priced when they waved the defect through. Deciding what your insurance won't cover is a normal, necessary act. Doing it for schedule or politics instead of risk is how a program insures itself against the wrong things.

The risk nobody remembers to insure: legacy regression

There's a category of risk that sits outside the visible project, which is exactly why it goes uncovered. The new system doesn't land in a vacuum. The legacy doesn't disappear, as the architecture chapter argued, and the moment you plug the new system into the existing landscape, you can break something in a peripheral system that nobody thought to retest, because it wasn't on the project's map.

The testing literature is blunt about why: because SAP modules are so tightly integrated, even a small change — a pricing condition, a tax setting — can ripple into processes far from where it was made. Now extend that beyond SAP's own

modules to every interfaced system around it, and you have a regression surface far larger than the project's test scope, which was drawn around what the project built, not around everything the project touches.

There's a subtlety worth knowing, too. In the SAP world, full regression testing is often positioned for the run phase after go-live, aimed at catching what version upgrades disturb. Which means a project frequently doesn't run complete regression against the connected legacy before go-live at all — it discovers those regressions in production, once real transactions start crossing the interfaces. The coverage gap isn't an oversight by careless people. It's structural: the risk lives in the space between systems, and the test scope was drawn around a single system. You insure what you can see, and the connected legacy is the part you've half-forgotten is still load-bearing.

Who decides the coverage

Step back to who is actually making these calls day to day, because it isn't the program director. It's the test manager and the stream leads, in the unglamorous flow of triaging defects, signing off levels, and deciding what "tested enough" means this week.

That's the point worth ending on. Every one of those daily decisions is a coverage decision on the program's insurance, and most of them get made implicitly, absorbed into the schedule rather than named. The test manager who concedes the SIT to hold a date has made a decision about coverage without calling it one. The lead who accepts a copy-the-SIT UAT because the business is busy has narrowed the policy without telling anyone the policy got narrower. The good ones make these calls explicit and at the right level, on a criterion of risk:

this is what we're choosing not to test, here's why, here's who carries the consequence if we're wrong. The weak ones let the calendar make the calls and call the result a test plan.

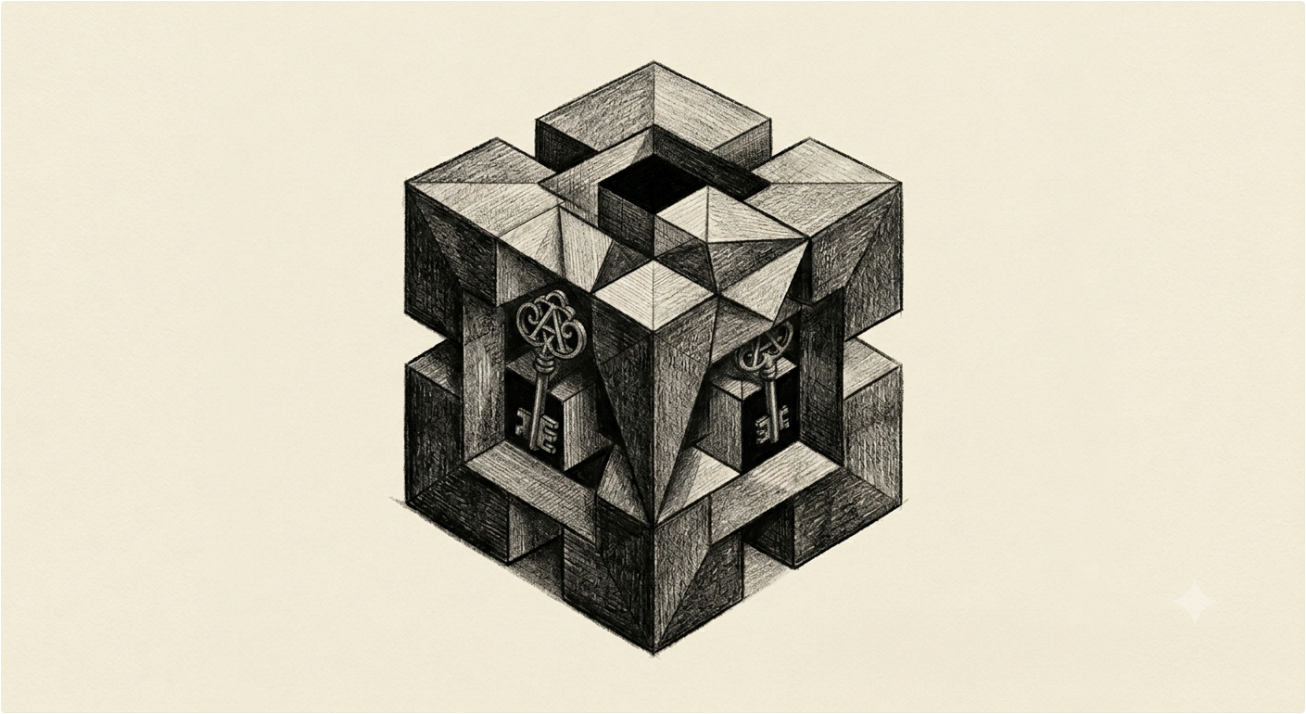
A program isn't judged by whether its UAT went green. It's judged by what that green was built to cover, and whether the people who narrowed the coverage did it on purpose, for reasons they could defend, or let it happen because a date was coming.

Once the system is tested for what you decided it should prove, the next gap opens around the people who'll use it, and the rights they need to do their jobs, rights that can only be designed once the processes behind them stop moving.

CHAPTER 07

Authorizations

Assigning roles too early, and rebuilding them later



The authorization stream almost always starts clean. It's set up like any other workstream, its design kicks off at the same time as the functional modules, and for most of the program nobody worries about it. Then it blocks at testing, a few weeks before go-live, when there's no slack left to absorb it. A stream that did nothing wrong at the start becomes the tension point at the end. That gap, between a healthy beginning and a painful finish, is the whole subject of this chapter.

This is a greenfield story. On a brownfield program the roles already exist, you carry them forward and adjust, and the mechanism below doesn't bite the same way. On a greenfield build, where every role is created from nothing against an

organization being designed in parallel, it bites hard.

The false culprit: starting too late

The obvious explanation is that the stream started too late, and it's wrong. Authorizations are treated as a stream among others, design begins in step with the modules, and that's fine. There's no problem starting the design at the same time as everyone else. Starting on time is the right call.

So if the timing of the start isn't the problem, something else is moving underneath.

What actually moves: the transactions, not the organization

Here's the distinction that the whole chapter turns on. The business organization can be perfectly stable at design time. A company knows its legal entities, its plants, its profit centers, its reporting lines. What is not stable is the set of transactions attached to each role. Those keep moving through the entire build, up to the first go-live and, honestly, past it.

During design, the business maps its organization, and that organization is often complex. You come out of it with a wide variety of roles. But each of those roles is bound to transactions that won't be settled until go-live. So all through build, the authorization team is working against roles whose content keeps shifting. Every design arbitration that moves a process, every transaction that gets added or swapped in a module, lands on the role mapping and forces a rework.

That's the technical reading of the trap. The team isn't slow. It's building on a layer that hasn't stopped moving, and each movement costs more than it should because of what comes next.

The combinatorics that kill testing

The rework is survivable during build. It becomes critical at testing, and the reason is combinatorial.

Take one position: a management controller. Now cross it with the organizational matrix. Factory controller or head-office controller. Then company code, profit center, plant. Then the fact that some of these people also touch accounts payable, others accounts receivable. By the time you've crossed the function with the full org matrix, a single position has produced something like a dozen roles to test. For one job.

Each of those has to go through unit testing, then integration testing. Multiply a dozen variants by every function in the scope and the test workload stops being a line item and becomes the thing that threatens the go-live date. This is where the early-mapping decision finally presents its bill. You designed the roles to match the organization in all its fineness, and now you have to prove every one of those fine combinations works, at the precise moment the program has no time left.

And the surface you have to prove is now larger than it used to be, because of the Fiori front-page layer.

The S/4HANA layer: front end and back end

On S/4HANA, a role isn't just a set of back-end transactions anymore. Access to a Fiori app rests on the launchpad layer (catalogs and, since the 2021 release, spaces and pages rather than the now-deprecated groups) sitting on top of the OData services that actually carry the business data.

How that splits across roles depends on the deployment. In a hub deployment, where the front-end server is separate from the back-end, you maintain two PFCG roles for the same access: a front-end role carrying the catalog, the space, and the OData start authorization, and a back-end role carrying the execution and data-access authorization. In an embedded deployment, the two can collapse into a single role. Either way, the point for testing is the same: an app can break at more layers than before. The tile may not show, the OData service may not start, or the back-end business authorization may be missing, and each is a different failure. On the hub projects I've worked, that meant validating the front and the back as two coordinated things, on top of a role count that was already exploding.

So the combinatorics from the previous section don't just multiply across the org matrix. They multiply across the interface layers too. Which is why, in practice, the program ends up doing the only sane thing left.

Nobody owns the pain, which is why nobody prevents it

Before the way out, it's worth being precise about who carries this, because that's why it's never anticipated.

There is no single owner. The SAP authorization team absorbs the technical rework. The functional teams on each module watch their transactions move, which moves the roles. And the business sees its own roles change often, with a real risk of regression each time a role is rebuilt. The pain runs through all three readings at once — technical, functional, and organizational — and that's exactly why no one stream sees it coming or holds it. A problem distributed across three owners is a problem owned by none of them. It surfaces only when it converges, at testing, as one undeniable workload.

The way out: cut wide for go-live

The simplest move, and the right one, is to cut directly for the first go-live. One controller role, restricted by company code or profit center. One wide role per domain, scoped by perimeter, rather than the fine matrix of variants the design produced.

Then you run a parallel targeting project on authorizations across the rollouts, refining the roles over time, at the pace of the business teams who can actually tell you where the fine lines should fall. The principle underneath is the one thing to take from this chapter: wide first, fine later. You defer the precision to the moment the perimeter is known, instead of designing it into the dark while the transactions are still moving.

This isn't a retreat from control. It's a reordering to secure the go-live. This leads some to object that the segregation of duties will not be respected. Let's address this point.

Segregation of duties: precision deferred, not control lost

Cutting wide does not mean SoD goes out the window. You keep a basic segregation of duties with a perimeter restriction. What you drop is the fine composite logic and the over-tight scoping.

Concretely: if a management controller also needs to do accounts payable, they get both wide roles. That's more roles assigned to that user than if you'd built a single tailored role — controller with an accounts-payable attribute and a few key transactions — but the wide roles are stable and the tailored one would not have been. Because every assignment is traced by the project, real SoD conflicts are rare. When one does appear, you grant the specific problem transaction on request, rather than pre-designing an exhaustive composite matrix to cover a case that may never occur. Same logic as the mapping itself: the precision is deferred, not abandoned.

What to take from this

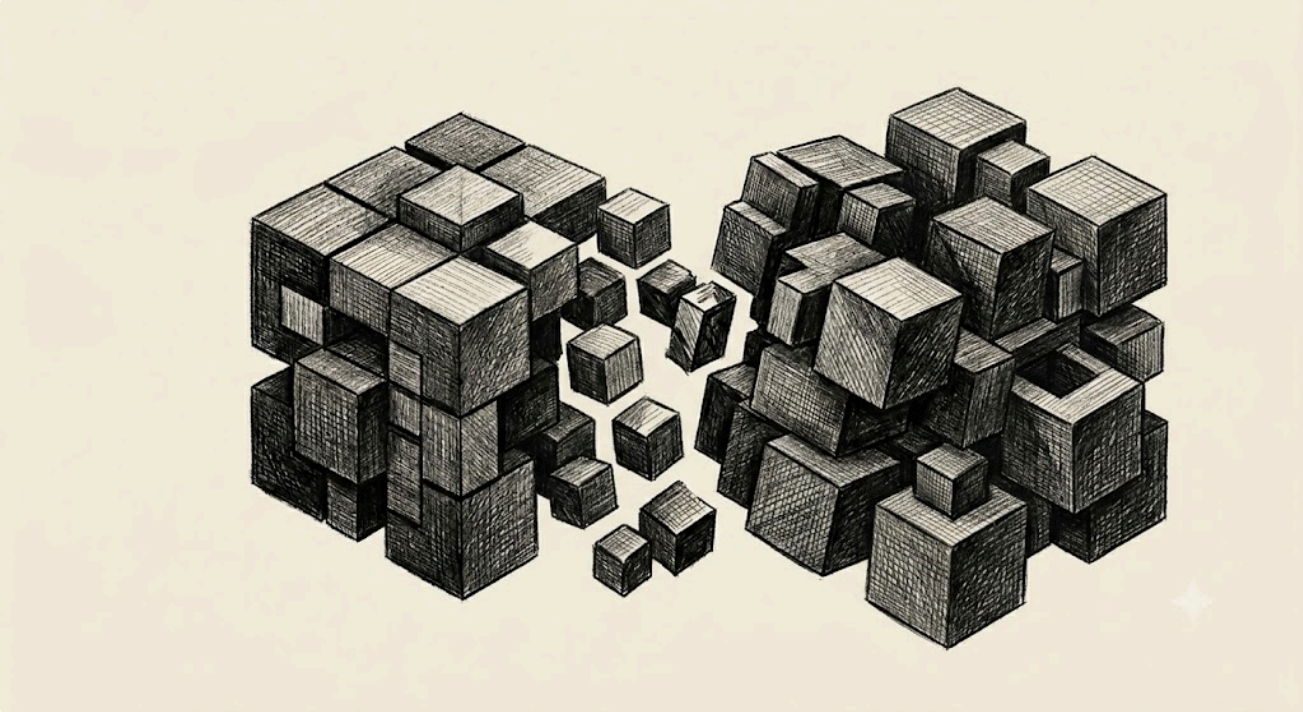
Authorizations are not a design problem to be solved finely up front. They're a layer to be frozen as late as possible. The program that maps its organization into roles at design time pays for it at testing, in combinatorial workload, at the moment it can least afford to. The program that cuts wide for go-live and refines through the rollouts keeps control of its schedule and gives the business the precision later, when the perimeter has stopped moving.

It's the same shape of failure as the next stream in this series: a workstream everyone underestimates because its real complexity doesn't show until late, downstream, when the cost of having underestimated it is highest.

CHAPTER 08

Data migration

Moving and cleaning the data the whole program depends on



A few weeks before go-live, the third mock load fails. Not on something exotic — on the data that runs the business: article master data, open items on the customer and vendor accounts. The kind of data you can't operate without, because without it there's no clearing, no payment run, no collection. The business impact is real and immediate, and for the first time the question on the table isn't "how do we fix this" but "do we still go live." When data migration derails, it doesn't push a task back by a few days. It pushes the go-live itself.

That's why it's the riskiest and most important stream on the program. And the interesting part is that almost nobody underestimates the migration itself. What gets underestimated is what makes it fail.

What isn't the risk: moving the data

Start by clearing away the obvious suspect. Migrating data, in the narrow sense of extracting it from one system and loading it into another, is a tooled and well-understood problem. The mechanics are not where programs get hurt. Everyone knows the migration is critical, it gets staffed and planned as critical, and the transport itself rarely fails on its own terms. The failure pattern practitioners keep describing isn't the transport, it's the scheduling: migration gets treated as the last ten percent of the project when the profiling, cleansing and reconciliation belong in the first thirty to forty, and once that work is deferred to the final weeks every defect becomes a go-live blocker with no time left to fix it. The point is consistent across post-mortems of failed implementations, where the root cause traces back to the data layer rather than the software.

So if the part everyone watches isn't the part that breaks, the risk is sitting somewhere people aren't looking, underneath the transport itself.

The real risk factor: legacy data quality

What gets underestimated is the state of the data in the legacy system. The cleaning is always longer and more expensive than planned, because the assumption going in is that the data is cleaner than it actually turns out to be. Years of workarounds, half-retired records, fields quietly repurposed for something other than their original meaning, duplicates nobody ever reconciled: all of it surfaces the moment you try to move it into a system that enforces rules the old one let slide.

Cleaning that data happens in two stages, with two different owners. First, the business teams clean at the source, inside the legacy system, wherever the data can be corrected there. They know what a record is supposed to mean, so they're the ones who can decide whether it's right. Then IT takes over for the mass cleansing, applying rules that can't be enforced in the legacy system at all, transforming the data inside the ETL tool. The two stages aren't interchangeable. Skip the business-led cleaning at the source and IT is left mass-transforming data it doesn't fully understand the meaning of.

That's the risk on a brownfield conversion. On a greenfield build, there's a second layer on top of it.

Greenfield: you don't transport, you transform

On a greenfield program you're not moving data into the same shape it had before. The target is a redesign, and that changes what migration even means.

It's not only a new chart of accounts or a new organizational model, though it's often those too. It's the transactional business rules themselves. If the project moves the company onto IFRS, the data has to be restated to fit. If it decides that flows which used to run through an external tool now run inside the ERP, that data has to be brought in and reshaped to rules it was never captured under. Each of those decisions turns migration from a transport problem into a transformation problem. You're not carrying the data across, you're rebuilding it to fit a structure and a set of rules it never lived in. That's the greenfield surcharge, and it sits on top of the cleaning.

Cleaning and transformation would already make migration the heaviest stream. But what makes it the most dangerous is where it sits relative to everything else.

The vise: caught between quality below and rules above

Here is what makes migration unique on the program. It's exposed on two sides at once, and it controls neither.

On one side, it's the foundation downstream. The migrated data is the base of the transactional flows every other stream runs on, so a defect in migration surfaces as someone else's problem — a test postponed or failed, in build, in the functional streams, anywhere that needs real data to prove a process works. The migration doesn't get to fail quietly in its own corner. When it's wrong, it blocks everyone.

On the other side, it's the receptacle upstream. The migration is where the business rules land. So when a rule changes during testing — because the business refines a point, decides a mapping was wrong, adjusts a transformation — the migration is hit in return. And this happens often. It isn't a rare event you can treat as an exception. The consequence is concrete and brutal for the migration team: they rework the same objects far more times than there are mock loads. The "three mocks" is a floor, a planning convenience. The real count of how many times an object gets rebuilt is set by how many times the rules above it move.

So the migration sits in a vise. The quality below it is worse than assumed, and the rules above it won't hold still. Neither pressure originates in the migration stream, and both are paid for there.

What the mock loads reveal, in order

The mock loads are where that vise becomes visible, one turn at a time. Three are usually enough, and they aren't three repetitions of the same exercise. They're three different things.

The first mock is technical. Its job is to prepare and shake out the tooling and the transformation rules, to prove the pipes work, not to land business-perfect data. The two that follow are business mocks, and each one aims to land as close to the go-live target as it can, on cleaned data and real volumes this time. Each pass uncovers a layer of reality the previous one hid: the first business mock exposes what the technical mock's clean assumptions glossed over, and the last one is meant to be the dress rehearsal.

Which is exactly why the third mock failing is the moment the whole chapter opened on. By then there's no margin left to absorb it, the data that fails is the data the business can't operate without, and the rules may have moved again since the second mock. The vise closes at the worst possible time, on the data that matters most.

The way out: make rule changes visible and costly

The mistake is to treat this as a technical problem and ask the migration team to be faster or more careful. The pressure on them isn't technical in the first place, so a technical answer won't relieve it. What relieves it is governance.

Every mid-project change to a business rule, for the migration and for every other stream, has to go through a committee, typically a Design Authority Board that arbitrates gaps. The lever that protects the program is how you classify those changes. You push them through as project gaps, a change of scope,

rather than as business gaps, an expected new feature the business is simply entitled to. That distinction does two things at once. It protects the IT teams from unbounded rework, because a change now has a visible cost and an owner. And it forces the business to think hard about whether a rule change is genuinely worth it, because changing a rule is no longer free, it's a decision someone has to defend in committee.

What to take from this

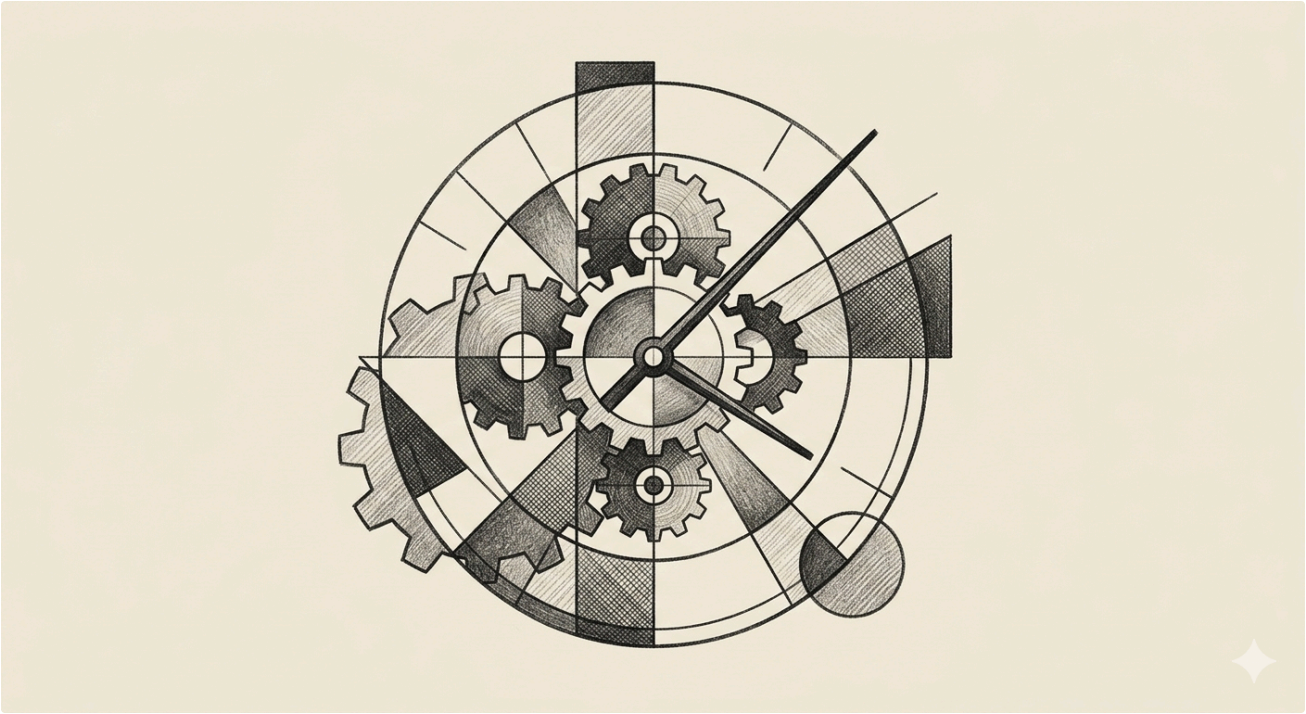
Data migration is not a technical transport problem you execute at the end of the project. It's the most exposed stream on the program, caught between legacy data quality that's worse than anyone assumed and business rules that won't stop moving through testing. You don't secure it by asking the team to migrate better. You secure it by governing what destabilizes it: cleaning the legacy data early and at the right level, and arbitrating every rule change so the migration doesn't silently absorb the cost of a business decision nobody had to defend.

A program that lets the rules drift without arbitration pays for it in migration rework, and eventually pays for it in the date. And whatever comes through this stream, in whatever state, is the same data the next stream has to display.

CHAPTER 09

Reporting

Building the same reports more than once



The more methodology you put in up front, the less you rebuild. That's true everywhere on a program, but reporting is where it shows most brutally. It's the workstream where missing groundwork doesn't fail quietly. It gets paid back in reconstructions — the same reports built a second time, a third time, because the first versions rested on something that hadn't stopped moving. Reporting is the workstream of rework, and most of that rework was avoidable.

This chapter is about the three reasons the same reports get built twice, at least, and what stops each one.

Greenfield and brownfield: two different traps

The shape of the problem depends on the kind of program. On a brownfield conversion, you carry the old reporting technology across. That means little rework and little risk of business regression, because the reports the business knows keep working. The trap there is performance: moving to S/4HANA, some of what used to be physical tables now sits behind views, and a report that ran fast on the old system can run slow on the new one. It's a performance watch, not a rebuild problem.

Greenfield is where the rebuild problem lives. You're not carrying anything across. You're changing the reporting technology, depending on data that's still being migrated, and absorbing a functional redesign whose rules are still moving, all at once. Three sources of instability stacked under one workstream. That's the program this chapter is about, and the rest of it follows the three reasons the reports get built more than once.

Reason one: a technology chosen too new

Reporting is the workstream where the technology moves fastest. Every year SAP and the surrounding vendors ship something new, and not all of it is mature. Pick the newest option for a greenfield build and you take on two risks at once: you become the one who works out the bugs on a solution that isn't reliable yet, or you watch that technology fade for lack of adoption and find yourself migrating a brand-new solution again within two or three years. The safer bet is a technology launched three to five years ago that has been widely adopted. Mature enough that the bugs are known, adopted enough that it isn't going to disappear under you.

Choosing between the options is easier with a clear decision matrix than with a vendor pitch. The dimensions that actually drive the choice are a handful. Is the reporting operational, close to the live transaction, or strategic, looking across history? Does it draw on one system or several? Does it need real-time data or a consolidated snapshot? And do you have the skills in-house to run the tool you pick? Mapping the main SAP options against those dimensions, as of mid-2026 and with the caveat that this landscape shifts fast:

TECHNOLOGY	BEST FOR	DATA SCOPE	LATENCY	SKILL BARRIER
Embedded analytics (CDS views)	Operational reporting, close to the process	Single system, live S/4HANA	Real-time	Low — accessible to business teams
SAP Analytics Cloud (SAC)	Dashboards, planning, visualization on top	Single or blended, via CDS/InA	Real-time (live connection)	Low to medium
BW/4HANA or Datasphere	Strategic reporting, history, consolidation	Multi-system, plus external data	Snapshot / consolidated	High — ABAP and data-warehouse modeling

A few things the table can't carry on its own. According to SAP's own integration guidance, the recommended data source precedence is ABAP CDS views first, then ODP extractors, then raw tables, and released CDS views are the recommendation because they stay stable across upgrades. SAC's live connections are only as good as the CDS views underneath them: SAP-aligned analysis is blunt that when the data foundation is messy, SAC surfaces the mess

to users, so investing in CDS quality before building dashboards pays off downstream. And embedded analytics is included in the standard S/4HANA license, where BW/4HANA and Datasphere are separate builds with their own cost and skill demands.

To illustrate how continually this technology moves, one date worth holding in view when you weigh whether a tool is near end of life: the mainstream maintenance for the older BW 7.5 is reported to end in December 2027, which is exactly the kind of horizon that turns a chosen tool into a forced migration.

Once you know this matrix and use it, it isn't the hard part. The hard part is that even the right technology gets rebuilt if you report the wrong data too early.

Reason two: a build started on moving data

This is the same planning clash that runs through the authorization stream. You cannot test whether a report is reliable when the transactional flows and the business rules feeding it are still moving, and you certainly cannot when you aren't even sure the migrated data you're displaying is correct. A report built on that ground isn't a report yet. It's a draft that looks finished, and it gets redone once the ground settles.

The reports that suffer most are the big aggregates, the P&L-type statements that pull from everything. They're the most fragile precisely because they depend on the entire upstream being stable — every flow, every rule, every migrated balance. Build one of those early and you're almost guaranteed to build it again. The discipline here is to lower expectations on the big numbers at the start, and to sequence their construction after the upstream has stabilized rather than racing to show an aggregate that can't yet be trusted.

This is where change management and proper communication matter. You know how finance people work: they want the perfect report, and they want it now, or they'll raise their hand to stop the go-live. The pedagogy matters as much as the technology here. You have to reassure them, and help them accept an unfinished report, or fewer reports, for a short period. That conversation is far easier had early than discovered at the go/no-go.

That holds for any single report. The next reason is about what happens across all of them at once.

Reason three: no common foundation

The instinct, under deadline pressure, is for each report owner to charge ahead and build their own report in isolation. It feels faster. It isn't. When everyone builds alone, the same business logic gets recoded report by report — a margin definition here, a cost allocation there, each one slightly its own. Then a rule changes, as rules do all through a greenfield build, and the change has to be made in every report that encoded it. The rework multiplies by the number of reports.

The alternative is to take the time to build a common catalog of CDS views first, shared across all the reports that need the same logic. When the rule changes, you change it in one place. Released CDS views are stable across upgrades, which is what makes the catalog worth treating as an asset rather than scaffolding. It's slower to start and far faster across the life of the program, and it's the single most effective thing a reporting stream can do to stop building the same thing twice.

Concretely, how do you do it? It's easier than it sounds. Once the final list of reports for go-live is settled, you build all the mock-ups, and before coding anything, you sit down with the consultant to define the CDS views behind each one. That's where you see the common patterns, the views that can be shared across several reports. It won't be perfect, and the catalog will be amended over time, but you start on solid ground rather than on a pile of one-off reports.

Knowing what to mutualize depends on knowing what the reports are and which ones matter, which is where the organization comes in.

Define early, build late: the sequencing the business owns

There's a subtle move here that's easy to get wrong in both directions. The business has to define its reporting need as early as possible, not so the reports can be built early, but so the common CDS views and the technology choice can be shaped around real requirements rather than guesses. The need is the input to the foundation.

But defining early is not building early. Once the need is on the table, the next move is to rank it, and the ranking deserves a matrix of its own rather than a gut feel about what's urgent. Four tiers cover most cases:

TIER	WHAT IT COVERS	TIMING
Statutory	Legal and regulatory reporting you cannot operate without	Required at go-live
Business critical	What's needed to invoice and keep the business running	Required at go-live
Important	Reporting the business relies on but can wait a few weeks	Shortly after go-live
Nice to have	Analysis that improves decisions but isn't load-bearing	When the foundation allows

The first two tiers are the go-live scope, the handful of reports genuinely required to operate on day one. The rest come afterward, in order. This is what reconciles the two pulls that otherwise look contradictory: you define the need early to get the foundation right, and you produce the reports late and ranked, so you're never building on data that hasn't settled. Early definition, late and prioritized production. The business owns both halves, and when it owns neither, reporting inherits all the downstream instability and gets built in a post-go-live scramble.

What to take from this

Reporting is the workstream most exposed to rework, which makes it the one that most rewards discipline in sequencing. The three reasons the reports get built twice come down to one principle: stabilize the foundation before you produce the reports. A mature technology rather than the newest one, a build

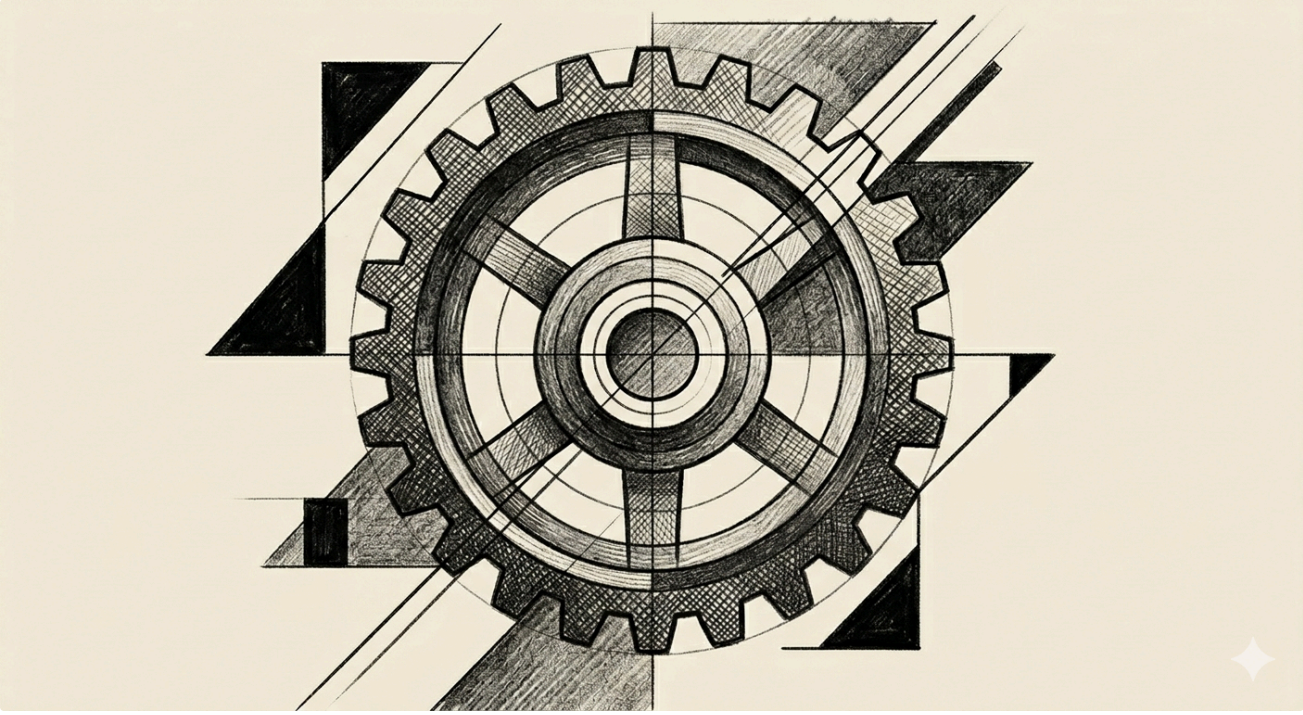
that waits for the data to settle, and a common CDS catalog rather than a scramble of isolated reports. Define the need early to shape that foundation, then build late and in order of priority.

This chapter showed, with the finance population especially, that change management can be the deciding factor. It prepares the business process owners and key users first, so they can steer a greenfield program properly, and then the end users, so they adopt the new solution quickly.

CHAPTER 10

Change

Preparing the business



Change management gets mistreated before it gets mismanaged. It sits on the plan, it has a lead, it shows up in the steering meetings, and it spends most of those meetings being talked over. This is partly a question of timing. When the build is still open and the design is still moving, a change consultant asking who's going to be trained and when reads as premature, an administrative concern raised while the real work is unfinished. So the stream gets a polite nod and very little authority, and the program tells itself it will get serious about change later.

It's worth being blunt about a second reason, because it shapes everything downstream. Change management runs on human skill more than on anything you can inspect, and human skill is hard to grade. A configuration is right or it fails a test; the verdict is external and quick. Change work has no equivalent. It's difficult to fail visibly at it, which quietly makes it the place a program parks a consultant who underdelivered somewhere the shortfall was measurable. No one plans this. It's a consequence of the fact that value on this stream is nearly impossible to assess in real time, so weak performance hides better here than anywhere else on the program.

That's the standing change management starts from. Everything that follows happens on top of that deficit.

Who actually becomes a key user

In a big organization the selection runs on availability: you get the people who can be spared, or who put their own hand up, and a department releasing someone for months is rarely releasing its strongest. There's a mirror image higher up, where the sponsoring executives treat the same transformation as a career move worth being seen on. The program is a springboard at the top and a staffing residual at the bottom, and the people sent to represent the business day to day come from the bottom.

What that produces is specific: a disengaged key user is a cost, but a key user who doesn't genuinely master their own process is a fault line. The requirements they give IT don't hold still, and it isn't because the business is genuinely undecided. It's because the person speaking for the business can't see their

own process clearly enough to pin it down. IT then reworks the same requirement again and again, and reads the churn as scope instability when the source is a selection decision made months earlier and three steps upstream.

Even the right person, chosen for the right reasons, hits a wall that has nothing to do with capability.

The job they still have to do

The key user almost never gets to stop doing their real job. They carry the project on top of it, and when the two collide, the day job wins. It has to. The invoices go out, the customer gets an answer, the close happens on the calendar it always happens on, and none of that waits for a workshop. The project is the elastic thing in the schedule, so the project is what gives. A program that assigns key users without buying back a real share of their time hasn't secured their contribution. It has scheduled a conflict it will lose every time the ordinary week gets busy.

Training that works, and training that only reassures

The standard approach is train-the-trainer: you pick a set of users, put them through structured sessions, and rely on them to carry it to their colleagues. The material tends to be either slides or SAP Enable Now, and the gap between those two is wider than it appears. Enable Now is the more dynamic of the two, and for people who've never touched SAP, or who've only known the old back-end transactions, it does something slides can't: it lets them build a feel for how Fiori is meant to work instead of memorizing a sequence of clicks they'll have lost by next week.

But the deciding factor isn't the tool. It's when the training happens. Run it too far ahead of go-live and it evaporates before anyone applies it to real work. The remedy is almost embarrassingly simple: put the training right up against go-live, just before or just after, so the gap between learning the system and using it is measured in days, not weeks. Knowledge that gets exercised immediately stays. Knowledge that gets shelved does not.

Measuring readiness without lying to yourself

Programs reach for a questionnaire to prove the training worked, and a quiet distortion enters right there. The change managers scoring their own effort would rather the results looked good, so the questions drift toward the easy end. Adding more of them makes it worse, not better. Users already treat training as time stolen from their real work, and a longer survey just sharpens that irritation while telling you nothing more reliable than a short one would.

The honest signal isn't what people say about their own confidence. It's what they've already done. A trainer who sat through UAT has driven the system under something close to live conditions, and that's worth more than any self-assessment, because it happened under pressure rather than in a demo. When those trainers are also your testers, the order of operations becomes a lever: schedule their training just before UAT and the two events feed each other, one preparing the ground the other tests. You end up reading readiness off behavior instead of off a form, which is the only version of it that survives contact with go-live.

None of this reads as a crisis while the program is still not in production. It reads as one afterward.

Where the under-investment lands

An underfunded change stream stays quiet until go-live, then presents its bill to the two teams least able to defer it: the business, and the IT support desk. Support drowns in tickets filed as defects that are nothing of the kind, just users who were never properly trained reaching for the only help channel they can find. The business side produces a darker signal. People book sick leave to sit out the first period-end close, choosing absence over being the one who has to run a process they were never made ready for.

Neither is a technical failure. Both are the training and readiness work that should have happened weeks earlier, arriving late and wearing the disguise of an operational incident.

What to take from this

Strip these mechanisms down and none of them are really about how you train people. They're about decisions the program made without quite admitting they were decisions: who it sent to stand in for the business, whether it freed them enough to actually do the work, and whether it checked readiness against what people had done or only against what they'd claim. Bad training isn't what a program pays for at the end. It pays for choosing the wrong people to prepare, and for keeping change management too far from the table to object while there was still time to change course.

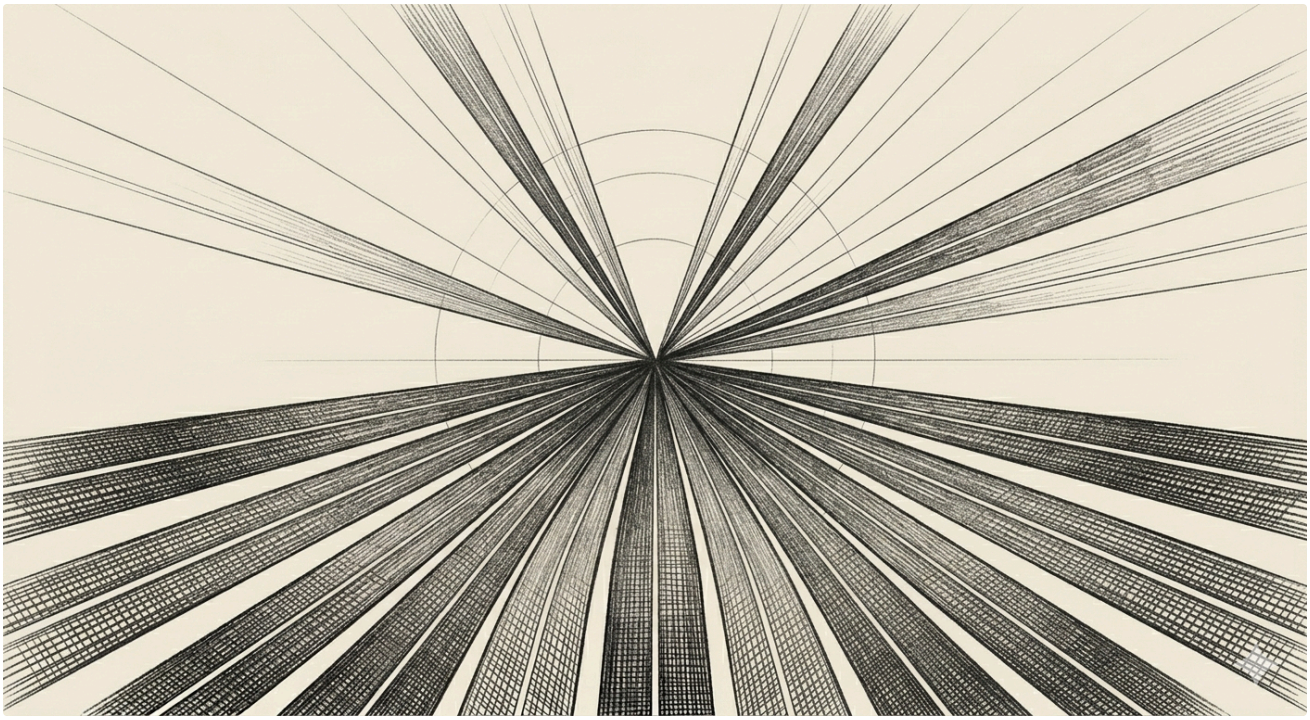
And yet, a program that makes every one of these mistakes can still come out ahead, because plenty of programs skip change management altogether and walk straight into the post-go-live problems this chapter has already described.

Doing it imperfectly beats not doing it. The point isn't to hold out for a flawless change stream, it's to stop treating the stream as optional.

CHAPTER 11

Cutover

Switching off the old system with no time left to fix a mistake



The illusion of the perfect score

Cutover enjoys a unique status in the collective imagination of ERP projects. It is the program's "showcase" phase — the one where you display three-thousand-line Excel spreadsheets, broken down into fifteen-minute slots, where every consultant, developer, and infrastructure expert is assigned a granular task. People speak in terms of "sequencing," "critical paths," and "go-live control rooms." It is, by a wide margin, the most meticulously choreographed phase.

It is also the least understood.

The fundamental error steering committees make is confusing the precision of the document with the certainty of the outcome. The hour-by-hour plan creates an illusion of total control because it is highly quantifiable and visual. But an Excel line has never coordinated anyone on its own. When task 142 encounters a two-hour delay because an extraction script from a legacy system fails, the plan merely mathematically pushes subsequent tasks closer to the red zone. The plan shows the disaster; it does not resolve it.

Agency as the real job of cutover

Cutover is not an exercise in passive administration. It is an exercise in real-time political and operational engineering. At the center of this storm stands the cutover lead. Their added value does not lie in their ability to turn cells green or red on a shared screen, but in their agency: their capacity to act, decide, and move the needle.

At this stage of the project, the lead is the sole point of convergence between worlds that ordinarily only speak to one another via support tickets: offshore developers, network infrastructure teams, functional experts, and executive management.

Consider a classic example: three distinct teams are all dependent on the exact same technical milestone to kick off their respective go-live sequences. Team A (infrastructure) has finished its part, but Team B (interfaces) hits a blocking bug, while Team C (business) sits waiting in front of their screens, watching their operational window shrink. If you follow standard protocol — opening a high-priority ticket, waiting for a three-hour incident triage meeting, analyzing the root cause — the go-live is dead.

This is where agency changes the game. The lead does not wait for standard governance. They pick up the phone, call the expert on Team B directly, cut through corporate silos, reallocate immediate troubleshooting resources, and, if solving the issue requires a scope trade-off, escalate to the steering committee within ten minutes to force an arbitration that standard paths would take two days to yield. The real job of a cutover lead is to break procedures to preserve the cadence of the score.

Rework is inherited, not created

Another common misconception is believing that cutover breeds its own crises. It does not. Cutover invents nothing; it merely exposes what was swept under the rug or poorly addressed over the preceding eighteen months. It is the program's ultimate funnel.

The nature of your project deployment completely changes the stakes here. In a brownfield approach (system conversion), cutover is relatively serene. The business rules are known, the organization is stable, and the volume of net-new structural data is minimal. Rework is low because the system is switching over foundations already heavily run by the business. In a greenfield approach (new implementation), the cutover absorbs full-frontal instability it had no part in creating. If business rules were not firmly arbitrated and locked down upstream — a direct echo of the shortfalls detailed in the data migration and reporting chapters — everything is still moving when you are ready to press the button. Teams discover during the final cutover weekend that account mappings or organizational structures validated three months prior no longer square with the messy reality of the production floor.

Manual activities without a safety net: the master data trap

The true breaking point of a cutover rarely lies within automated pipelines or code transports. The lethal danger to a go-live sits in manual activities, specifically the case of incomplete master data.

Despite multiple simulation cycles (the mock loads), a gap always persists between the dress-rehearsal environments and true production. Customer profiles missing valid tax codes, material masters without extended accounting views, strategic vendors missing bank routing numbers. These anomalies are frequently discovered in the dead of the night during the live switch, at the exact moment the team attempts to load initial opening balances or run the first transactional health checks.

The tragedy of manual data at this stage is that there is no recovery window left. You cannot ask a team of exhausted accountants to manually correct or enrich ten thousand line items at four in the morning when the core systems must open for business at eight. It is this exact mechanism — the late discovery of accumulated, uncleaned data debt — that dismantles schedules and forces last-second go-live postponements.

A system without a shock absorber

Why is cutover so ruthless? Because it is the only phase of the project that operates with zero buffer.

During design, if a workshop fails, you reschedule a session for the following week. During build, if a development slips, you push out the entry date into integration testing. Even during testing, management can choose to narrow the

insurance policy's coverage to hold the timeline.

When you enter cutover, the clock ticks in real time against the outside world. Production plants will stall, distribution trucks will stop shipping, and invoices cannot be issued. Every approximation from previous chapters — the lax governance of fit-to-standard, the integration dead-ends, the happy-path testing on complex scenarios — converges into an ultra-compressed space-time continuum where no shock absorbers remain. Every minute lost is paid for directly in business downtime.

What to take from this chapter

You do not secure a cutover by investing time in writing an even more granular plan, or by stacking extra rows into a tracking spreadsheet. You secure it by placing a person armed with real authority and high agency at the center of the chessboard, capable of warping formal structures to handle the unexpected.

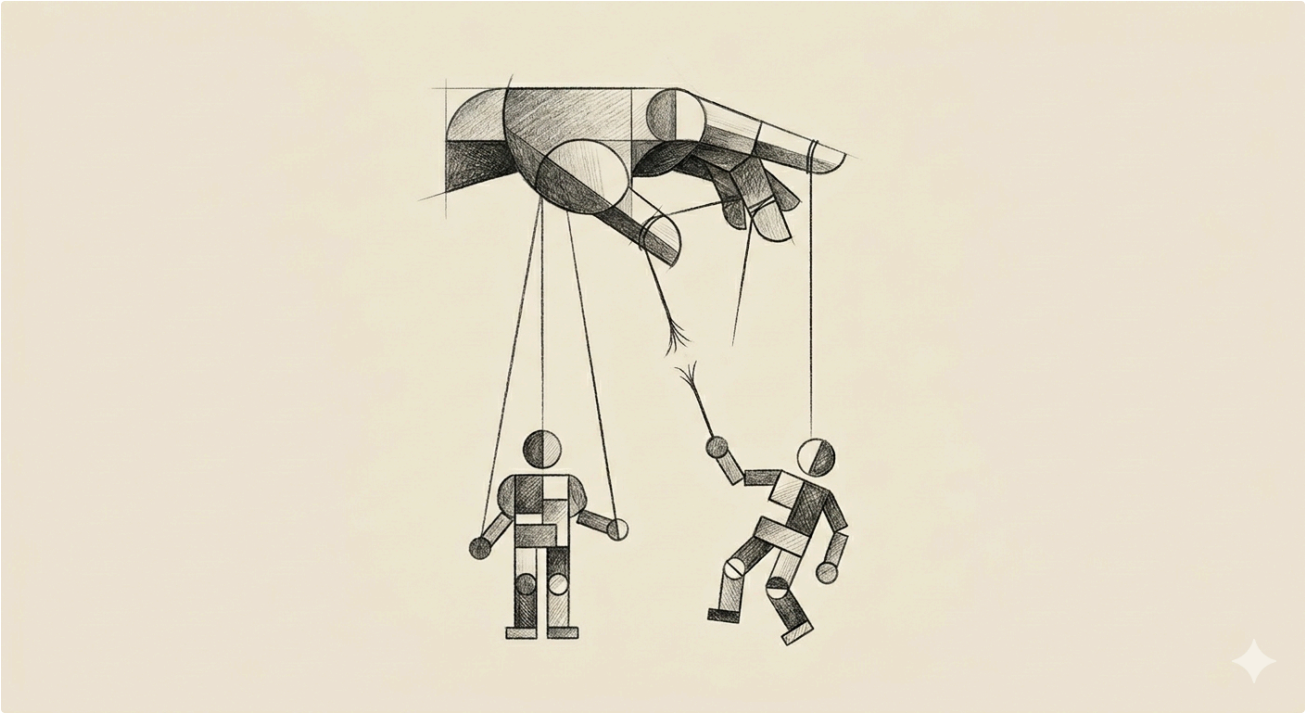
More importantly, you secure it by enforcing an iron discipline upstream: no core data, no critical configuration, and no major human activity should ever rely on luck or last-minute improvisation over the go-live weekend.

Once the cutover concludes, the system is opened, users log in, and reality begins. It is the moment all the program's promises must finally survive the daily grind of the enterprise.

CHAPTER 12

Run

Deciding when support can take over from the project team



The illusion of the calendar ceiling

Every large ERP program has a date on a slide marked "Hypercare Exit." In smaller brownfield implementations, this period typically spans a manageable two to four weeks. For complex, global greenfield programs, it stretches to anywhere between one and three months. This timeframe is usually negotiated a year in advance during the scoping phase, driven by industry benchmarks and budgetary constraints rather than actual project realities.

The trap is treating this calendar ceiling as an objective signal of operational readiness. It is not. It is merely a commercial buffer.

When the calendar date arrives, the project team is eager to pack their bags and transition to the next rollout or project phase. The business, meanwhile, is often drowning in a backlog of unresolved issues, yet lacks any measurable framework to contest the transition. Treating hypercare exit as a function of time assumes that stability is something that happens passively as days pass. In reality, an ERP system does not stabilize because the clock ticks; it stabilizes because business processes successfully survive their initial operational cycles.

The two gates of true operational readiness

To secure a stable landing, a program must abandon subjective feelings and calendar dependencies. Hypercare must only conclude when the system and the organization pass two strict, concurrent gates: the quantitative gate and the qualitative gate.

The quantitative gate removes emotion from the room by relying on hard trend lines over a continuous, rolling window — typically ten consecutive business days. You do not exit hypercare because you had one good day. You exit because P1 and P2 backlogs are below a strictly defined threshold that the standard support organization is staffed to handle, the net volume of incoming incidents is strictly decreasing or has plateaued at a sustainable baseline, and the rate of defect resolution outpaces the rate of incident creation.

The qualitative gate, the matrix of firsts, acknowledges a fundamental truth of core enterprise software: some business processes are executed daily, while others happen only once a month, once a quarter, or once a year.

If a greenfield ERP goes live in June, the local support team might successfully handle thousands of daily sales orders and shipping notifications over a four-week period. The hypercare metrics look pristine. But if that organization has not yet executed its first complex month-end close, its first quarterly tax consolidation, or proven its ability to generate year-end reporting assets on live production data, the system is not truly in a "Run" state.

True readiness dictates that hypercare cannot close for a functional stream until that stream has successfully executed its most critical "first-time" operational events under close supervision from the project team.

Structural friction and the handoff failure

The transition from a project environment to an operational environment is rarely smooth, primarily due to structural pressures and cognitive gaps.

In modern ERP delivery, teams are frequently caught in a rolling deployment schedule. The project engine cannot afford to stall; it must move its best resources to the next wave of scoping and design. This creates an institutional rush to force the current wave into the run phase prematurely. When combined with a highly distributed or offshore support model, the friction multiplies.

Offshore support models introduce structural realities that project management regularly underestimates: longer onboarding ramps, high turnover rates, and a cultural aversion to challenging poorly designed solutions under pressure. When the project team attempts a standard "HR transfer" — handing over a mountain of documentation, functional specifications, and a Jira backlog over a series of brief knowledge-transfer sessions — the model breaks.

A successful transition is not an administrative asset transfer; it is a deep cognitive transfer. A project team troubleshoots an incident by looking at code variables, custom tables, and technical configuration. A business user experiences an incident as a broken process: an invoice that cannot be cleared, a truck that cannot leave the warehouse. If the support team is trained only to look at the bug and the variable rather than the upstream business flow, their competence is effectively neutralized. They will absorb technical debt, struggle to diagnose root causes, and slow down operational velocity.

Shift the center of gravity: support-first governance

The most effective organizational counter-measure to handoff failure is simple yet radical: who runs the hypercare must lead the hypercare.

Traditionally, the project team owns the hypercare phase, while the support consultants sit in the passenger seat, observing and taking notes until the formal handoff date. This architecture incentivizes bad behavior. The project team, driven by deadlines, implements quick patches and workarounds to make the hypercare metrics look acceptable, knowing they will not be around to support those patches long-term.

Instead, the permanent support organization must take the steering wheel on day one of go-live. Under a support-first governance model, every incident created after the system cutover lands directly in the support team's queue. The project team is repositioned as a secondary tier of elite engineering support, pulled in only when the core support team identifies a fundamental design flaw or structural defect.

This flip in the organizational matrix forces immediate, active engagement. The support team cannot afford to remain passive spectators because they are actively managing the business's frustration. Concurrently, it forces the project team to deliver cleaner, more sustainable configurations during the build phase, because they know any shortcuts will be immediately rejected by an operational team that holds true veto power over hypercare exit.

The SLA ticketing trap

When a support organization is structured around rigid, outsourced contracts without face-to-face alignment, it inevitably falls into the SLA ticketing trap.

Consider an anonymized scenario from a global manufacturing rollout: a critical interface failure prevents a major supplier from uploading pricing updates, stalling production lines. The ticket is opened with a P2 priority. Under the contractual service level agreement, the support partner must acknowledge the ticket within two hours and provide an update within forty-eight hours.

The offshore support consultant opens the ticket, identifies a minor ambiguity in the data field, and changes the ticket status to "Awaiting Customer Clarification." The SLA clock pauses. Three days later, the business responds. The consultant reviews it, realizes it requires an infrastructure check, and routes the ticket to the Basis infrastructure team. The SLA clock resets or pauses during the transfer. The Basis team reviews it, finds the infrastructure is fine, and routes it back to the functional team with a request for more logs.

In one actual enterprise case, a single high-priority ticket underwent sixteen consecutive internal owner changes over a span of several months. On paper, every single SLA metric was green. Response times were perfect,

acknowledgment rates were flawless, and the dashboard looked spectacular to executive leadership. In reality, the business process remained completely broken for months, until a senior independent consultant was brought into a physical room and resolved the root cause in less than forty-eight hours.

The SLA tool had ceased to be a system for issue resolution; it had become a sophisticated cultural refuge designed to hide accountability.

To shatter the SLA trap, leadership must break the tool's monopoly on communication. You do not fix a broken hypercare phase by adding more fields to a ticketing dashboard. You fix it by establishing high-cadence, small-committee triage meetings held every forty-eight hours. In these sessions, support managers, functional stream leads, and business process owners review open tickets face-to-face. A ticket cannot be passed to another team without verbal arbitration and immediate accountability. The focus shifts entirely from keeping an SLA clock green to clearing the operational roadblocks for the business.

Final reflections on the journey

The run phase is not an afterthought to an ERP program; it is the ultimate judge of its value.

You do not secure a successful operational lifecycle by throwing more money at an extended calendar window or by expecting an offshore ticketing factory to decipher poorly documented custom code. You secure it by enforcing hard quantitative trend lines, respecting the matrix of firsts, establishing support-led governance from the very first hour of go-live, and refusing to let your support organization hide behind the corporate fiction of green SLA dashboards.

With this landing, the long loop that began with scoping concludes. The software, once an abstract list of requirements on a steering committee slide, is now interwoven with the daily survival of the enterprise. The project ends, the program fades, and the continuous flow of standard operations takes its rightful place.

I spent weeks writing these chapters. They're a placeholder for my experience and my curiosity about the field.

I used AI to write them, but not the way you might picture: a few loose sentences thrown at a model, hoping something usable comes back. I built a specific setup for this project, with detailed instructions and context, so the tool could actually challenge my thinking rather than just take dictation. Only once the deep dive, the structure, and the message were clear did a first draft get written. From there I corrected it, sharpened the ideas, and cut whatever slowed the read down, because a piece meant to be a real-world account from the ERP world has to fight two things at once: its subject matter and its own length. Both are dry by default.

I genuinely hope this helps. If you're working through an ERP implementation or architecture decision and could use another set of eyes, you can find me at www.theleverage.tech/contact or on LinkedIn — Michel Escoda.