# Introduction

I started writing Salesforce code in 2010. This book is the version of architecture advice I wish I'd had at year four — when I was technically capable but kept making decisions that cost the team six months of cleanup later.

The platform's official documentation tells you *what* the features do. There's a lot of it. Most of it is accurate. None of it tells you *which feature to use, when, and what it'll cost you in maintenance two years later*. That's the question I've been trying to answer for myself across about a dozen production orgs. The answers in this book are the patterns I've settled on, the war stories that taught me the patterns, and the cases where the pattern still doesn't work and I'm not sure what does.

## Who this book is for

You've finished Trailhead. Maybe finished it twice. You've shipped Apex and Flows in production. You've debugged a real incident that took a real evening. You're in the part of your career where the title says senior or lead or architect, or it's about to.

You want a senior practitioner's mental model of Salesforce architecture. Not a tutorial. Not a feature reference. The frame for thinking about decisions, with enough specific patterns and stories that the frame becomes practical.

## Who this book isn't for

People who are completely new to Salesforce. There are better books for the platform basics — Trailhead is genuinely good, and several Salesforce-specific intro books exist. This book assumes you can read Apex without a glossary and have configured a sharing model at least once.

Pure admins who don't write code. Some of this book will be relevant — the chapters on data modelling, sharing, and team dynamics — but a lot of it is developer-flavoured. There are admin-focused architecture books that fit better.

CRM strategy generalists looking for vendor evaluation help. This book assumes you've already chosen Salesforce and you're now trying to use it well.

## How to read it

The chapters mostly stand alone. You can pick the one that matches the problem you're staring at and read it first. They're not strictly sequential.

That said, **Chapter 1 (The Architect's Mindset) is the lens for the rest.** Without that frame, the specific patterns in the later chapters can read as just another set of best practices. With the frame, they become tools you choose deliberately.

Most of the chapters are technical. **Chapter 10 (Working With Teams) isn't.** It's a single long story about people, because architecture is half technical decisions and half communication, and I wanted to put one chapter in that took the people side seriously.

The chapters with the strongest opinions are 1, 2, 4, 7, and 10. The chapters with the most code are 5 and 8. Pick whatever you need first.

## A note on the war stories

Every chapter has stories. The stories are real, with composite details. Names are first names — sometimes the actual person's name (with their permission), sometimes a stand-in for someone whose anonymity I want to protect. Numbers are real where I remember them and approximate where I don't; *"about 18 months"* and *"around 4 hours"* are real measurements I genuinely don't remember exactly.

Some stories don't have clean resolutions. *"We never fully fixed the picklist drift, we added monitoring and called it good"* is a real outcome, and the book is honest about those rather than tying every story up in a bow. Real engineering archives are full of half-fixes and pragmatic compromises. Pretending otherwise would have made the book less useful.

## What you should walk away with

Each chapter has its own takeaway. The composite goal of the book is something like: a frame for asking the right questions before any architectural decision, plus enough specific patterns and stories that you have starting points for most of the common problems, plus an honest sense of where my frame doesn't work and where you'll need to figure things out yourself.

If I had to summarise the whole thing in one paragraph: choose boring patterns, document the reasoning, build for the team that comes after you, and be honest about trade-offs. Most of the rest follows from there.

## A challenge before you start

Pick the worst architectural decision in your current org — the one that's been bothering you, the one that constrains every new feature, the one that everyone has agreed not to talk about. Find the chapter most relevant to it. Read that chapter first.

The book exists to help you make decisions like that one differently next time. The reading is mostly useful when you're carrying a real problem with you.

Off you go.

---

# Chapter 1: The Architect's Mindset

Three weeks after my first project as Architect (capital A) went live, Daniel pinged me on Slack at about 9pm.

> ***Daniel:*** *mate I think your rule engine is broken* ***Daniel:*** *like, silently* ***Daniel:*** *I tried to add a rule and it just didn't fire* ***Daniel:*** *no error, no log, nothing*

I asked him what he'd done. He sent me a screenshot. The metadata record looked correct. The rule should have fired. It didn't.

I spent the next two days tracing it through the framework I'd built. The framework was very clever. It had abstractions for abstractions. It had a custom metadata-driven configuration layer feeding into a rule evaluator that handled hierarchical conditions, stored its execution history in a Big Object, and had a UI component for non-technical users to author rules without code. It was, technically, a triumph.

It was also producing a silent SOQL row limit failure the moment someone tried to add their first rule, because in production the rule evaluator was loading every metadata record at runtime and the org had grown past the 50,000-row mark in the time between my testing and Daniel's first attempt.

Daniel was generous about it. I rewrote the whole thing as three Apex classes and a custom metadata type with maybe seven fields. Took me an afternoon. Has been working ever since. Daniel still occasionally messages me — usually around the time someone in the team is about to over-engineer something — with a screenshot of the original framework's directory structure and the text "remember this?" Helpful, in its way.

That was the project where I learned that becoming an architect isn't a title change. It's a decision-making shift. The senior developer asks "can I build this?" The architect asks "*should* anyone build this, in this org, with this team, expecting this kind of maintenance for the next five years?" Most decisions become harder once you start asking the second question, but most outcomes get better.

This chapter is the lens I now apply to every architectural decision. None of it is novel. All of it was earned painfully.

## The four questions before any design

When a stakeholder brings me a request — and "stakeholder" here means anyone, from the sales team to a fellow developer to the CEO — I run it through four questions before I touch a sandbox or

open Flow Builder. The order matters.

**1. What problem is this actually solving?**

Not "what does the request say." What's the underlying business outcome they're trying to reach. Most requests describe a *solution* the requester has already designed in their head — "we need a custom object for X" or "we need a button that does Y." The architect's first job is to translate the solution back into a problem and then propose alternatives.

I've stopped accepting requests in the form "we need feature X." I push back politely until I have a sentence starting "we want to be able to…" or "we lose time when…" or "we get errors when…" Once the problem is in that form, the solution space is much wider than the requester realised, and the request often dissolves into a 15-minute config change. Sales managers especially appreciate this once they realise it gets them what they actually wanted faster than what they originally asked for.

**2. Does the platform already do this?**

Standard Salesforce features cover an embarrassing percentage of what people ask for. Validation rules, formula fields, list views, report types, dashboards, approval processes, page layouts, dynamic forms, record types — these handle the vast majority of "we need…" requests with zero custom code. Before I design anything new, I check whether something I'd be embarrassed to mention as the answer is the answer.

The default outcome of this question is *no, the platform doesn't do this exactly* — but partial standard features combined with a small piece of custom logic almost always beat a fully custom build. I've watched architects (including me, earlier in my career) propose custom Lightning components for problems that a properly configured List View would have solved.

**3. If I build this, who maintains it after I leave?**

This is the question that distinguishes architects from senior developers. A senior developer optimises for what they can build well. An architect optimises for what the team can maintain after the architect moves on.

The answer reshapes the design. If the team is two people — one senior dev and one admin who's only been on Salesforce for a year — the architecture has to be three boring custom fields and a Flow. If the team is fifteen senior developers with a CI/CD pipeline, the architecture can be more sophisticated. But it still pays to ask whether the sophistication is earning its keep.

I had this question pointed out to me by an architect named James, who was about ten years senior to me at the time and had a reputation for shipping unimpressive code. I said something about wanting to use the new feature in a Spring release. James said *"Will Priya understand it in six months when you're not here to explain it?"* — Priya being the senior admin who'd be maintaining whatever I built. I said no. James said *"Then the answer is no."* I argued for a bit. He was right. I didn't build the thing.

**4. What's the cost of changing this in two years?**

Some decisions are cheap to reverse. A custom field can be deleted, a Flow can be turned off, a permission set can be unassigned. Others are expensive: a sharing model rewrite is a six-month project, a data model change with millions of records requires a migration, a custom object that's been integrated to seven other systems can't be retired easily.

When the cost of change is low, decide quickly and iterate. When the cost of change is high, slow down, document the reasoning, and get more eyes on the decision. The art of architecture is recognising the difference. Most projects fail because they apply the same urgency to both.

## Build for the person who joins after you leave

Every decision has a half-life. Whatever you write today is the cryptic legacy code someone curses at next year. The trick is to make decisions that age well — not because the decisions are timeless, but because they're legible to someone with two years' less context than you have right now.

I default to boring. Single trigger per object, named the same way every time, doing the same kind of thing in the same order. Standard naming conventions, even when I have a strong opinion about a better one. Custom metadata for configuration, so the next person doesn't need to deploy code to flip a flag. Comments that explain *why* the unusual choice exists, not what the code does. None of this is innovative. All of it adds up.

The hard version of this rule is letting go of patterns I personally find elegant when they're going to cost the team. I once spent two weeks designing a fluent builder API for a custom service — the kind of thing where you'd write `OpportunityService.query().byAccount(a).withStage('Closed Won').execute()`. Beautiful from my side. Total mystery to Liam, the junior dev who needed a list of opportunities six months later and didn't know what a fluent API was. He spent an hour trying to figure it out, then quietly wrote a regular SOQL query in his own helper class. Smart move on his part. I refactored my version back to plain static methods after that. Less elegant. Far more usable. The team thanked me for it three months later, when they'd added six more methods without asking me a single question.

The test I now apply: would a developer with two years' less experience than me, looking at this code for the first time, be able to predict what it does and modify it without breaking it? If the answer is no, the cleverness is costing more than it's earning.

There's a related thing about naming. I once joined a project with an Apex class called `OpportunityServiceManagerHelperUtil2_v3.cls`. The "v3" was because there'd been a v1 and a v2. The "Final" wasn't in the name but might as well have been. Inside, it was 4,000 lines of unrelated static methods someone had been bolting on for years. The original author had left. I asked the team what was in it. Nobody knew. I asked who used it. Several people said *"I think I use one method? I'm not sure which one."* We deleted about half of it after a quiet two-week audit. Nothing broke. Someone — I think it was Sanjay — eventually wrote a proper service class for the

parts that mattered. The original `OpportunityServiceManagerHelperUtil2_v3` is still in the codebase as a thin shim that re-exports the few methods anyone still calls. We've been meaning to delete it. We probably won't.

## The org is the goal, not the pattern

In my first two years as an architect, I had favourite patterns. The trigger framework. The selector layer. The custom metadata-driven configuration. I was happy when I got to use them and frustrated when the project didn't justify them. I was wrong about which one was the goal.

The org is the goal. The patterns are the means. A pattern that fits well makes the org better. A pattern that doesn't fit well makes the org worse, even if the pattern itself is correct.

I worked on a financial services project where I started rolling out the trigger framework I bring to every engagement. Three weeks in, I noticed the team was confused — they had three triggers across the entire org, none of them complex, all of them written by Daniel, who'd been there for about a decade. The framework I was introducing solved problems they didn't have. Worse, it added levels of indirection that made the simple cases harder to understand. Daniel was too polite to say anything directly, but he kept asking me questions in code review that boiled down to *"is this actually going to make things better?"* Eventually I caught on. I pulled the framework out, added a one-page document explaining when they'd want to introduce something like it in the future, and shipped what they actually needed: a fourth trigger that fit the existing patterns. Daniel said *"good call"* and went back to whatever he was actually working on.

Pattern-first thinking treats every engagement as a chance to apply the architect's library. Org-first thinking treats every engagement as a chance to leave the org better than you found it, with whatever tools fit. The second one is the right frame.

This doesn't mean abandoning standards. It means recognising standards are local. A pattern that's right in one org is wrong in another, and the architect's job is to read the org before reaching for the pattern.

## Boring is the strategy, not the style

When I look back at the architectural decisions I'm proudest of, almost none of them are clever. They're decisions that look obvious in hindsight — the kind that make someone reading the code say "of course, what else would you do?" That's the goal. Not "wow, look at that," but "of course."

Boring code is faster to write, faster to review, faster to debug, and faster to extend. Clever code is faster to feel pleased with for an afternoon. The aesthetic preference for clever solutions is one of the most expensive habits a developer can have, and the transition from senior developer to architect is largely about un-learning it.

There's a temptation, especially early in an architect's career, to demonstrate sophistication. To pick the harder pattern, to use the newer feature, to express the same logic in fewer lines. I've done all of these. I now pick the easier pattern, the older feature, and the more obvious code path almost every time. The career outcome is identical. The maintenance outcome is much better. The team outcome is dramatically better.

Boring isn't a personality trait. It's a strategic choice — a recognition that most of the cost of software is in maintenance, not authorship, and the boring version is cheaper to maintain. When I write a class that someone could have written, I've succeeded. When I write a class that only I could have written, I've left a trap.

The exception is the genuine novel problem — the case where the standard patterns don't fit and a real architectural innovation is required. These are rare. I've worked on dozens of projects and counted maybe four genuinely novel design problems across all of them. The rest were variations on patterns the platform community had already solved.

## When the best decision is not deciding yet

The instinct of a new architect is to decide things. A meeting is held, a question is raised, and the architect feels the pressure to answer. Sometimes the right answer is *"we don't have enough information to decide this responsibly — let's do X for now and revisit when we know more."*

This isn't avoidance. It's recognising that some decisions are cheap to defer and expensive to make wrong. If a feature could be implemented as a Flow now and refactored to Apex later if performance becomes an issue, the right answer is the Flow plus a documented plan for revisiting at a specific trigger (e.g. "if this Flow processes more than 50,000 records per night, we move it to Apex"). The deferred decision is more honest than a premature one.

I keep a running document on every project called *deferred decisions*. Each entry has the question, the temporary answer, the trigger condition for revisiting, and the date the decision was deferred. It's a few lines per item. It saves an enormous amount of arguing because nobody can claim the decision was forgotten.

The hardest version of this rule is deferring decisions when the team wants closure. Engineering teams generally prefer "this is the answer" over "we'll figure it out when X happens." That preference is mostly cultural; engineering teams that've been burned by deferred decisions become risk-averse to deferring. The architect's job is to communicate clearly that *some decisions are cheaper to defer responsibly than to make poorly* — and to make sure the deferral is genuinely responsible, with a documented trigger and owner.

The decisions I most regret are the ones I made too early because I felt the pressure of the meeting. The ones I'm most proud of are sometimes the ones I deliberately didn't make for two months while we gathered usage data.

## What an Architecture Decision Record looks like

The most useful single document I produce on a Salesforce project is the Architecture Decision Record, or ADR. It's a short, structured note that captures *why* a decision was made — not what the decision was, which is visible in the code, but the reasoning that produced it.

Every ADR I write has the same five sections:

```
# ADR-007: Use Custom Metadata Types for Trigger Bypass Settings

## Status
Accepted — 2024-03-12

## Context
We need a way for admins to disable specific trigger handlers during
data migrations and bulk loads, without deploying code changes. The
team operates under a release freeze model, so any solution requiring
a deployment to flip a flag is unworkable.

## Decision
Use a custom metadata type `Trigger_Setting__mdt` with one record per
trigger handler. Each record has a `Disabled__c` boolean and an
optional `Bypass_User__c` reference. Trigger handlers consult this
metadata in their static initialiser and short-circuit if disabled.

## Alternatives considered
- Custom Setting (hierarchy): rejected — data, not metadata, doesn't
  travel with deployments, hard to version.
- Custom permission: rejected — affects user-level access, doesn't
  generalise to "disable for everyone during migration."
- Code-level boolean flag: rejected — requires deployment to flip,
  defeats the purpose.

## Consequences
- Admins can flip the metadata in 30 seconds via Setup, no deploy.
- Metadata records are version-controlled with the project.
- One additional metadata read per trigger transaction (~5ms,
  cached after first read in transaction).
- Adds a layer of indirection that needs documenting in the team
  onboarding guide.
```

That's it. About 250 words. Takes 15 minutes to write and saves a huge amount of rehashing later. When a new dev asks *"why are triggers structured this way?"* I point them to ADR-007 and they

understand the full context in 90 seconds. Without the ADR, the same question generates a 30-minute conversation, often with someone (sometimes me) misremembering the original reasoning.

The format matters less than the discipline. Pick one and use it consistently. The first ADR is awkward. By the tenth, it's habit.

## What over-engineering looks like in practice

I'll close with the over-engineered framework I mentioned at the start of this chapter, because it's the cleanest example of the lesson.

The requirement was simple: when an Opportunity stage changes, send an email to the account owner. Three Salesforce features could have handled it: a record-triggered Flow, a process builder, or a six-line Apex trigger. The right answer was the Flow. Any of them would have shipped in a day.

Instead, I built this:

```
public abstract class NotificationHandler implements
Database.Batchable<SObject>, Database.Stateful {
    public abstract Database.QueryLocator getQuery();
    public abstract NotificationConfig getConfig();
    public abstract Set<Id> getRecipients(SObject record,
NotificationConfig config);
    public abstract Messaging.Email buildEmail(SObject record, Id
recipientId);

    public Database.QueryLocator start(Database.BatchableContext bc) {
        return getQuery();
    }

    public void execute(Database.BatchableContext bc, List<SObject>
scope) {
        NotificationConfig config = getConfig();
        List<Messaging.Email> emails = new List<Messaging.Email>();
        for (SObject record : scope) {
            for (Id recipientId : getRecipients(record, config)) {
                emails.add(buildEmail(record, recipientId));
            }
        }
        if (!emails.isEmpty()) Messaging.sendEmail(emails);
    }

    public void finish(Database.BatchableContext bc) {
        System.debug('Notification batch complete: ' + bc.getJobId());
```

```
        }
    }
```

Plus subclasses, plus a custom metadata-driven configuration layer, plus a scheduling mechanism, plus a logging utility. About 600 lines of code total, in service of *"send an email when a stage changes."*

It worked. It was, technically, configurable. It was also completely opaque. About a year later, an admin named Mei needed to send an email when a Case was closed. She stared at the framework for an hour, asked Daniel what it did, Daniel didn't know, and she eventually wrote a one-line Flow. She was right to do that. Daniel told me about it months later, gently, the way someone tells you there's spinach in your teeth. I rebuilt the framework as three lines in a record-triggered Flow. The original is still in the codebase as a deprecated stub. We've been meaning to delete it. We probably won't.

The version that should have shipped in the first place was Mei's: filter on stage change, then a Send Email action. Not interesting. Not extensible. Perfectly maintainable. Would have prevented the "fork around the framework" pattern that emerged the moment I wasn't there to maintain it.

I tell this story because the temptation to build the framework is real and recognisable. It feels like the architect's job. It's exactly what the architect's job isn't.

## The mistakes I made in my first two years as an architect

A few things, specifically:

**Over-engineering for hypothetical future requirements.** The configurable rule engine. The notification framework. The fluent builder API. All of them built for "we'll need this eventually." We never did. The team I left behind inherited code they didn't understand, solving problems they didn't have. I'd have made the org better by writing three boring Apex classes that did exactly what was asked.

**Treating code review as style enforcement.** Every PR came back with twenty comments, most about formatting, naming, or framework alignment. The team learned to dread my reviews. The senior developers eventually started just merging without my approval because the cycle time was too painful. Daniel, again being kind about it, eventually said *"mate, can you just leave a thumbs up unless something's actually wrong?"* I tried. It was a hard habit to break. Style and naming should be enforced through automated tools (linters, formatters) or documented standards. Manual review should be reserved for substantive concerns: correctness, security, maintainability, governor limits, missing tests. Reserving review for substance means the team takes the comments seriously when they come.

**Ignoring the soft side of the job.** Architecture is half technical decision-making and half communication. I undervalued the communication half for years. I'd produce a beautifully reasoned ADR and email it to the team, expecting them to absorb 1,500 words of dense technical reasoning. Most of them didn't. The decisions I'd documented were "official" but never actually adopted, because nobody had walked the team through the reasoning in a way that produced shared understanding. The fix wasn't more documentation. It was more conversation. Architecture decisions need a verbal explanation, ideally in a 30-minute discussion where people can ask questions and push back. The written ADR is the artefact that survives the conversation, not the substitute for it.

These three mistakes cost me real time. Every architect I've talked to has made some version of all three. The fastest way to make the transition from senior developer to architect is to recognise these patterns early and adjust before they cost a project.

## Closing

The architect's job, distilled to one sentence: make the org better than you found it, with patterns the team can extend without asking you, and decisions documented well enough that the next architect doesn't have to relitigate them.

Most chapters in this book are about specific architectural domains — data, sharing, automation, integration, security, packaging, DevOps. Those chapters give you the tools. This chapter gives you the frame for using them. Without the frame, the tools become solutions in search of problems. With the frame, they become unobtrusive — the right one used at the right moment, no more.

If you take one thing from this chapter, take this: the architect's job isn't to be the smartest person in the room. It's to make the room smarter for not having you in it. Daniel, if you're reading this, that's also the answer to the question you've been asking me at every catch-up.

---

# Chapter 2: Designing the Data Model

I once joined a project as architect three months in, after the original architect had quietly moved to another team. The data model already had 23 custom objects. I asked the team why. The lead developer — a sharp woman named Priya who'd been on the org for six years — pulled up a whiteboard photo someone had taken, looked at it for a moment, and said *"I don't know. I think we just kept making them."*

It took us about four months to consolidate down to eight. Most of the cleanup was Priya's — she'd been wanting to do it for a year and finally had the architectural cover. We deleted maybe 800 metadata records, archived data into a dozen Big Objects, and rewrote three integrations that had been pointing at objects we no longer needed. Nothing broke that we couldn't fix. The team got faster almost immediately because there were just fewer things to think about.

The data model is the most expensive thing to change later. Every other architectural decision sits on top of it. Get it right early or pay forever. Most of the architect's leverage in week one is in saying no to objects that don't need to exist.

This chapter is about how to make those decisions — what counts as a good schema, where the costs hide, and which traps are worth avoiding even when the request seems reasonable.

## Default to standard objects

The first question I ask when someone proposes a custom object: *"Could we use the standard Account object for this?"* Or Contact, or Lead, or Opportunity, or Case, depending on what they're describing.

Most of the time the answer is yes. Standard objects come with built-in features the team will eventually need — mass actions, list views, mobile support, reporting connectors, integration hooks. Replicating any of these on a custom object is real work. Replicating all of them is six months of work the team didn't budget for.

The argument for custom usually goes: *"We need fields that don't fit the standard object."* That's almost never true. Salesforce orgs have absurd custom field limits — 800 on most standard objects. The right answer is to add fields to the standard object and use record types or page layouts to scope which fields show in which context. Not to create a parallel custom object.

The argument I do accept: *"This represents a fundamentally different concept that will have its own lifecycle, ownership model, and integrations."* Then yes, custom object. But "fundamentally different concept" is doing a lot of work in that sentence. A "Sales Lead" is not fundamentally

different from a "Lead" — it's a Lead with a record type. A "Customer Project" is not fundamentally different from an Opportunity — it's an Opportunity post-Closed Won. A "Service Request" is sometimes a Case and sometimes its own thing, depending on how the support team operates.

The test I use: if I asked five different people on the team what this object represents, would they all give roughly the same definition? If the answer is no, it's not a coherent concept yet, and creating an object will calcify the confusion. Better to push the team to define what they mean before adding to the schema.

## What every field type actually costs

Field types aren't free. Some of them carry costs that don't show up until much later, when refactoring is expensive.

**Text fields** are usually fine. Pick a reasonable length and move on. The trap is using Text(255) for everything because it feels safe — you end up with addresses crammed into Text(255) when you should have a structured address, names crammed into Text(255) when you should have a Name compound, and so on.

**Long Text Area** is what people reach for when Text isn't enough. The cost: Long Text fields can't be filtered in reports, can't be referenced in formula fields, can't be used in validation rules in any practical way, and don't sync cleanly to most external systems. They're also expensive in storage. Use them when you need long-form prose — Description, Notes, that sort of thing — and not as a fallback for "this field might be long sometimes."

I once joined a project where someone (not anyone I want to embarrass by naming) had created a field called `Phone__c` as Long Text Area, because they wanted to store multiple phone numbers in it. Validation rules can't operate on Long Text. Reports can't filter on it. The data was unusable for about a year — not unrecoverable, just unqueryable in any sensible way. We eventually parsed the field into a child Phone object. The migration was straightforward. The wasted year was the cost.

**Picklist fields** carry the most subtle cost: picklist drift. Over time, values get added by admins, sometimes by integrations, occasionally by Apex. Records reference values that no longer exist on the picklist (Salesforce keeps them as "inactive values" rather than rejecting them at load time). Reports that filter on picklist values miss records with old or misspelled values. Formula fields that switch on picklist values produce surprising results when a new value is added.

The fix isn't avoiding picklists — they're often the right choice — it's having a discipline. Use Global Picklist Value Sets for any picklist that appears on more than one object. Add a "validate picklist value drift" item to the org's quarterly health-check rotation. Don't allow integrations to insert arbitrary picklist values. Every drift I've inherited has been preventable.

**Formula fields** are useful and cheap until they aren't. The cost: formula fields recalculate on every record access (depending on the formula), they can't index well, and they're easy to nest. I've seen formula chains five levels deep where field A references field B references field C and so on, all calculated on the fly, all making the record list view slow. The fix is to flatten — use a real field updated by a Flow or trigger when the source data changes, rather than a formula chain that calculates dynamically.

**Hierarchy fields** (a Lookup to the same object) sound clever for representing parent-child structures. They're also one of the slowest things to query. SOQL doesn't have recursive queries; you have to walk the hierarchy iteratively in Apex. For shallow hierarchies (3-5 levels) this is fine. For deeper ones, performance degrades. If you find yourself building a recursive walker for a hierarchy field with more than 6 levels, consider whether the data should actually be in a separate structure.

## Junction objects, self-lookups, and the many-to-many trap

When two objects have a many-to-many relationship, the standard pattern is a junction object — a custom object with two Master-Detail (or Lookup) fields, one to each side.

```
Project__c <---> Project_Contact__c <---> Contact
                      (junction)
```

The junction has both lookups, plus optionally a unique constraint to prevent duplicate links, plus any fields that describe the relationship itself (role, start date, allocation percentage).

The trap: people forget the junction is a real object. It needs its own sharing model, its own permissions, its own validation rules. If a user can see Project__c and Contact but not Project_Contact__c, they can't see the relationship — and Salesforce's standard sharing inheritance doesn't always do what you expect.

Code-wise, the junction pattern is straightforward:

```
public class ProjectAssignmentService {
    public static void assignContactsToProject(Id projectId, Set<Id>
contactIds) {
        // Avoid duplicates by checking existing assignments first
        Set<Id> existingContactIds = new Map<Id, Project_Contact__c>([
            SELECT Contact__c FROM Project_Contact__c
            WHERE Project__c = :projectId
            AND Contact__c IN :contactIds
        ]).keySet();

        List<Project_Contact__c> toInsert = new
```

```
List<Project_Contact__c>();
        for (Id contactId : contactIds) {
            if (!existingContactIds.contains(contactId)) {
                toInsert.add(new Project_Contact__c(
                    Project__c = projectId,
                    Contact__c = contactId
                ));
            }
        }
        if (!toInsert.isEmpty()) insert toInsert;
    }
}
```

Add a unique constraint at the platform level by creating a hidden formula field that concatenates the two lookups ( `Project__c & '|' & Contact__c` ) and marking it as Unique. This prevents the same pair from being inserted twice even if the application code misses the check.

The mistake I see most often: people use a Master-Detail-Master-Detail junction (both relationships are master-detail) and then realise they can't move records between parents because Master-Detail relationships are immutable. If the relationship needs to be changeable, use Lookups, not Master-Detail. The trade-off is you lose roll-up summaries, but you can get those back via a trigger or a Flow.

Self-lookups (a field that references the same object) are how you build hierarchies. Account.ParentId is the standard example. The trap: nothing in the platform prevents circular references unless you add a validation. If A's parent is B and B's parent is A, the record renders fine until something tries to walk the hierarchy and stack-overflows. Validate this in a before-save Flow or a trigger.

## Person Accounts (when and why not)

I have to write about Person Accounts because they generate more architectural debate than any other single Salesforce feature, and I've made the call both ways.

Person Accounts merge the Account and Contact objects when an Account is a person rather than an organisation. The use case is consumer-facing businesses — retail, banking, healthcare, anywhere your customers are individuals not companies. Once enabled, Person Accounts can never be turned off. That alone makes the decision a one-way door.

Arguments for Person Accounts:

- Single record per consumer, simpler to reason about than the Account-Contact pair
- Better support in many AppExchange packages designed for B2C

- Marketing Cloud and some other Salesforce products integrate more cleanly with Person Accounts
- The data model matches the business model — a person is an Account, not an organisation that happens to have one Contact

Arguments against:

- Some standard features behave differently with Person Accounts (Salesforce documentation doesn't always make this obvious)
- Custom integrations need to be Person-Account-aware from day one — code that does `[SELECT Id FROM Account WHERE Name = …]` may behave differently than expected because the Person Account's "Name" is a compound field
- Once enabled, can never be disabled. If you turn it on and three years later realise the business is actually B2B with consumer ancillary, you can't undo it.
- AppExchange packages that aren't Person-Account-aware will break in your org

I made the call to enable Person Accounts on a project in 2018 and I've thought about it occasionally over the years since. I'd probably make the same call again, but the trade-offs aren't as clean as the docs suggest. The integration work to make every external system Person-Account-aware was substantially more than I'd budgeted for. Every report builder on the team had to learn that Account.Name is sometimes "Acme Corp" and sometimes "Hannah Smith" and that the latter has special render rules. We absorbed it. It worked. It also cost time I hadn't planned for.

If you're considering Person Accounts: don't decide it on day one of an implementation. Spend two weeks evaluating the integration impact, the AppExchange packages you'll install, the reporting model the team will live with. Then decide.

## External IDs and the idempotency pattern

External IDs are the single most underused feature in Salesforce data architecture. They're how integrations stay sane.

An External ID field is a custom field marked with the External ID checkbox. Salesforce automatically indexes it, and — critically — you can use it in upsert operations as the matching key. This means an integration can perform "create or update" against a record without ever needing to know Salesforce's internal Id.

```
List<Account> incomingAccounts = parseFromIntegration(payload);
// Each account has its External_Id__c populated from the source
system
upsert incomingAccounts External_Id__c;
```

That `upsert ... External_Id__c` syntax is the magic. If a record with that External_Id__c exists, it gets updated. If not, a new record is inserted. The integration is *idempotent* — running it twice doesn't create duplicates.

The pattern compounds. Once every record has an External_Id__c that maps deterministically to the source system, you can re-run integrations safely, recover from partial failures, migrate data without losing track of what's already been migrated, and rebuild from source if anything goes badly wrong.

The mistake I see: External_Id__c set up as a free-text field that the integration tries to populate but doesn't enforce. People type in values manually. The values drift. The matching fails silently. The integration creates duplicates because it can't find the existing record. Fix: make External_Id__c required, set it to be unique, and prevent manual edits via permission.

Also: don't reuse Salesforce's standard `Id` field as the matching key, even though it's tempting. Salesforce IDs change between sandboxes (and on full data migrations), so any integration that relies on them breaks the first time someone refreshes a sandbox. External IDs survive sandbox refreshes. Always use External IDs for cross-system matching.

## Designing for volume before you have it

The first time I worked on an org that genuinely had volume — millions of records, daily integrations pumping in tens of thousands of new ones — I was asked by Liam, who was new to the team, *"do we really need to plan for 5M records when we have 50k now?"*

The honest answer was *"if we don't plan for it now, we'll be paying for it in three years when we hit it."*

Volume changes the architecture. Reports that work fine at 50k start timing out at 5M. SOQL queries that return acceptable result sets become Selective-Query-required. Apex batch jobs that process all records in one transaction need to switch to chunked Batch Apex. List views need indexed filter fields. Custom indexes (which require a Salesforce Support case to request) take 2-4 weeks to be approved.

Designing for volume is mostly about three things:

1. **Know which fields will be filter targets and ensure they're indexed.** Standard fields that are commonly filtered (Account.Name, Opportunity.StageName, Case.Status) are indexed by default. Custom fields aren't unless you explicitly request indexing. If you're designing an integration that will filter on a custom field, file the index request with Support during the design phase, not three months later when the query starts timing out.
2. **Avoid wide records.** A record with 800 fields is harder to query than a record with 80 — even if you only SELECT a few. The platform's record cache, the network payload, the

audit trail size — they all scale with field count. Push less-used fields to a related object if the record gets too wide.

3. **Plan archival from the start.** Records have a lifecycle. Cases close, opportunities end, leads convert. After they're no longer active, do they need to live in Salesforce forever? If not, design an archival strategy — maybe Big Objects, maybe an external warehouse, maybe a custom long-term storage solution. The archival decision is much cheaper to make early than late, when you're already at 50% of your storage allocation and Salesforce starts billing you for overage.

I told Liam yes, we did need to plan for 5M. We designed the data model with indexed External IDs, a documented archival strategy, and a custom index request lodged in week three. Eighteen months later when the org hit 4.2M records, queries were still fast and reports still ran. He's now an architect at a different org and has told me he asks the same question to every junior on his team.

## A data-model review checklist

Before any new object goes into production, I run through this list. It takes about ten minutes per object and catches most of the recurring problems.

- **Does the standard data model already cover this?** Account, Contact, Lead, Opportunity, Case, Order, Product. If yes, use them. Don't add a custom object.
- **Is the object name singular?** `Project_Contact__c`, not `Project_Contacts__c`. Salesforce convention. Sounds trivial. Catches integration bugs months later.
- **Are all required fields actually required, or just "we'd like them to be filled in"?** Required fields prevent integrations from creating records when the source data is incomplete. Be deliberate.
- **Is there an External_Id__c field? Is it Unique and Indexed?**
- **Are picklists Global Value Sets or local?** Use Global if the values appear on more than one object.
- **Are formula fields necessary, or could they be flattened to real fields?**
- **Is the sharing model OWD set?** Default of "Public Read/Write" is rarely right.
- **Are there any Long Text fields that should actually be structured?**
- **Is there a record retention/archival plan?**
- **Has someone other than the original designer reviewed the schema?**

The list isn't exhaustive but it catches enough of the common mistakes that it earns its keep. The "has someone else reviewed it" point is the most important — most data model problems are obvious to a fresh pair of eyes and invisible to the person who designed it.

## Closing

Schema is the foundation. Every other architectural decision in the chapters that follow assumes the data model underneath is sane. If it isn't — if there are 23 objects when 8 would have done, or Long Text fields holding structured data, or picklists drifting unchecked — the cleanups in those later chapters become harder.

Most data-model problems aren't dramatic. They're small accumulations: one extra field here, one premature picklist there, one missing External ID. Each is forgivable. Together they're a maintenance tax on every future feature.

Spend the time in week one. The team that comes after you will get a better org because of it. Priya did most of the cleanup on that 23-object project; the team is still mostly using the consolidated model three years later. Sometimes the architect's biggest contribution is the things that didn't get built.

---

# Chapter 3: Sharing and Permissions

The phone call came at about five past eight on a Wednesday morning. I was still finishing my coffee.

It was Anneliese, the delivery manager. She didn't bother with hello.

> *"Shubham. Hannah from comms can't see her own account record. Her manager can't see it either. The CEO is asking why. Can you call me back in five?"*

I called her back in three.

By 8:20 I had three browser tabs open — the production org's setup, the change history for sharing rules, and a Slack message I was halfway through writing to Daniel asking him to check production. By 8:30 Daniel had confirmed it wasn't just Hannah; about forty other users couldn't see records they should have been able to see. By 8:45 we'd narrowed it to a sharing rule we'd written about 18 months earlier — the rule shared certain Account records with users in a specific role hierarchy if a phone field on the Account matched a particular format.

The Spring '20 release had changed how phone numbers rendered on the Account compound name field. The change was subtle. The sharing rule hadn't been updated for it. The criteria-based sharing was now matching on a string that looked correct visually but had different invisible whitespace characters than before. Records that previously matched no longer did. Records that hadn't previously matched now did. Hannah was caught in the first group; for about ninety minutes the CEO's account had been mis-shared to the entire org because it now matched a criteria nobody had intended.

We disabled the rule by 9:15. Patched the criteria by 11:30. Re-ran the sharing recalculation by 2pm. The CEO never noticed his account had been over-shared because nobody who shouldn't have seen it bothered to look. Hannah was nice about the whole thing — she even offered to take me through the comms team's other workflows in case I wanted to find more bugs.

I declined.

That morning is the version of sharing I want every architect to internalise. The sharing model isn't a feature. It's a graph of dependencies that runs through every object, every field, every record. When something in the graph shifts — a release, a data change, a process update — bits of it break in ways that don't fail loudly. They just quietly stop showing the right things to the right people.

This chapter is about how to design sharing well enough that those graph-level breakages happen rarely, and when they do happen, you can find them.

## The three layers, briefly

Salesforce's sharing model has three layers, and most architects can recite them by rote. I'll be brief because the docs cover them well; what matters is how the layers interact.

**Org-Wide Defaults (OWD)** set the baseline. For each object, OWD is one of: Public Read/Write, Public Read Only, Private, or Controlled by Parent. The OWD answers the question "if I don't say otherwise, who can see this record?"

The right answer for most objects is Private or Controlled by Parent. Public Read/Write is the default Salesforce gives you for new custom objects, and it's almost always wrong. Public Read/Write means every user in the org can read and modify every record of that object. The first thing I do on a new project is review every OWD and tighten anything that's at Public Read/Write without a deliberate reason.

**Sharing rules** open access above the OWD baseline. They come in two forms: ownership-based (records owned by users in role X are shared with users in role Y) and criteria-based (records where field X equals value Y are shared with users in role Z). The Hannah story above was a criteria-based sharing rule that misfired.

**Manual and Apex sharing** open access for individual records, by individual user or group. Manual sharing is what users do via the "Share" button on a record. Apex sharing is what code does, programmatically inserting `Share` records.

These three layers compose. A record's effective access is the union of OWD, sharing rules, and manual/Apex shares. A user can lose access only if all three deny it.

## Profiles vs Permission Sets vs Permission Set Groups

Permissions and sharing are different but adjacent. Sharing controls *which records* a user can see. Permissions control *which objects, fields, and features* a user can interact with. Both have to permit an action for the action to be allowed.

Profiles used to be the primary permission container. Each user had exactly one profile, and the profile granted object permissions, field permissions, app access, page layouts, login hours, and so on. The model was simple to explain and impossible to keep clean as orgs grew. Every team needed slightly different access. Every "slightly different" became a new cloned profile. Six months in, you had twenty profiles. A year in, you had thirty. Nobody could explain why.

Salesforce has been steering teams away from profile-based permissions for years. Object and field permissions are moving to permission sets. Profiles will eventually be reduced to a thin layer covering page layouts, login hours, and record type defaults — the things permission sets still can't do.

The migration is real work. I covered the broad strokes in the article on moving from profiles to permission sets, so I won't repeat the full migration playbook here. The architectural point is: design for permission sets from day one if you can. Retrofitting later is more painful.

## The layered permission model

The pattern I bring to every project: three permission set layers, applied via permission set groups.

**Base layer**: one permission set every user gets. Standard tab access, read access to common reference data, basic app visibility. Nothing role-specific. The minimum a brand-new employee needs to log in and not see an empty screen.

**Function layer**: role-specific permission sets. "Service Agent" grants access to Case, Knowledge, Entitlements, the Service Console app. "Sales User" grants access to Opportunity, Lead, Quote, the Sales app. These are composable — someone working across both roles gets both permission sets. No cloning.

**Exception layer**: one-off elevated access. Maybe a team lead needs delete permission on a specific object. Maybe a data steward needs edit on a normally-read-only field. These are small, named, and ideally time-bound. Don't make them permanent unless they're permanent.

In practice, you compose these via Permission Set Groups. Instead of assigning five individual permission sets to every Service Agent, you create a "Service Agent" permission set group bundling the base set, the service function set, and any exceptions. One assignment per user. Clean.

Permission set groups also let you use **muting sets** — a permission set inside the group that explicitly *removes* permissions that the included sets would otherwise grant. This is the feature that makes the pattern actually work for real teams, because it lets you compose without duplication.

Concrete example: your Service Agent function set grants edit access to Case. But for a specific team that should only have read access to Cases from other business units, you add a muting set to their group that revokes edit on Case. Same base permission sets, different effective access. No cloned permission sets.

I've talked to teams who'd never heard of muting sets. They generate a lot of architectural relief once people see them.

## The Hannah problem, revisited

Let me come back to the Spring '20 morning, because it illustrates something important about sharing.

The sharing rule that misfired wasn't badly written. It was written against the platform's behaviour at the time. The platform changed underneath it. Eighteen months later, on a Wednesday morning,

the rule started doing something different.

This is the recurring failure mode of complex sharing models. Sharing rules accumulate. Most of them work for years. A few of them depend on platform behaviours that change in ways the rule's author didn't anticipate.

There are two defences:

**Audit sharing rules quarterly.** Pull the list of all sharing rules, their criteria, their target groups, and their last-modified date. Rules older than two years that nobody on the current team can explain are candidates for removal or rewrite. Most orgs have at least three of these. Some have dozens.

**Test sharing rules during release preview.** Salesforce ships a preview sandbox before each major release. If you have sharing rules with anything resembling fragile criteria — string matches, picklist values, formula field comparisons — test them in the preview sandbox before production rolls. Most teams skip this. The teams that don't skip it have boring release weekends.

We didn't do either of these on the org where Hannah's record went sideways. We started doing both immediately after.

## Apex managed sharing

Sometimes sharing rules aren't expressive enough. The criteria you need don't fit in the platform's criteria-based sharing language. Or the sharing logic involves multiple objects in a way the standard tools can't represent.

Apex managed sharing is the escape hatch. You write Apex code that creates `Share` records — `AccountShare`, `OpportunityShare`, custom `MyObject__Share` — programmatically.

```
public static void shareWithProjectTeam(Set<Id> projectIds) {
    List<Project_Contact__c> assignments = [
        SELECT Project__c, Contact__c, Contact__r.User__c
        FROM Project_Contact__c
        WHERE Project__c IN :projectIds
        AND Contact__r.User__c != null
    ];

    List<Project__Share> shares = new List<Project__Share>();
    for (Project_Contact__c assignment : assignments) {
        shares.add(new Project__Share(
            ParentId = assignment.Project__c,
            UserOrGroupId = assignment.Contact__r.User__c,
```

```
            AccessLevel = 'Edit',
            RowCause = Schema.Project__Share.RowCause.Project_Team__c
        ));
    }

    Database.SaveResult[] results = Database.insert(shares, false);
    // Check results for errors — Apex sharing inserts can fail
silently
    // if the user doesn't have base access to the object
}
```

The pattern requires a custom RowCause ( `Project_Team__c` in the example), which is configured in Setup before code can use it. Custom RowCauses are how you distinguish "this share was inserted by my code" from "this share was inserted by the standard sharing engine."

Apex sharing is powerful and easy to get wrong. The standard mistakes:

- **Forgetting to delete shares when the relationship ends.** If a contact leaves a project, you need to delete the `Project__Share` record for them, or they'll keep having access. The deletion is rarely automatic.
- **Inserting shares without checking the user has base access.** Salesforce silently rejects share records for users who can't see the object at all.
- **Sharing recalculation gaps.** When OWD changes or the role hierarchy shifts, Salesforce recalculates standard sharing automatically. Apex managed shares don't get recalculated unless you write the recalculation code.

Use Apex sharing when you have to. Don't reach for it as a default.

## Field-Level Security as a first-class decision

FLS controls which fields a user can see and edit. It's set per-field, per-profile (or per-permission-set), and it's the part of security architecture that most orgs treat as an afterthought.

The trap: a field can have CRUD access (the user can read the object) without FLS access (the user can't read this specific field). The user gets a record with the field grayed out or missing. From the user's perspective this is confusing. From the integration's perspective it's worse — most integrations assume FLS access if CRUD is granted.

When designing a custom object, design FLS at the same time. For each field, decide:

- Who needs to see this field?
- Who needs to edit this field?
- Is the default state "visible to all" or "hidden by default"?

The defensible default is hidden by default. Add FLS via permission sets only to the roles that need the access. Salesforce defaults new custom fields to "no FLS for any profile," which is fine if you remember to configure it. The trap is forgetting and shipping a field nobody can see except the System Administrator. Many fields do exactly that for months because nobody had a use case that exposed the gap.

There's a related Apex trap I cover in the article on CRUD/FLS enforcement, so I'll only mention it here briefly: Apex code runs with the executing user's CRUD/FLS unless you explicitly bypass it. If your code assumes it can read every field on a record but the user can't, you'll get incomplete data and possibly silent errors. Always audit Apex against the FLS model.

## The audit pattern

The most useful defensive pattern I run is a quarterly sharing-and-permissions audit. It takes about half a day. It's saved me from at least four incidents I can specifically remember.

The audit covers:

- **OWD review.** Has any object's OWD changed? Why?
- **Sharing rules.** List all of them. Last modified date. Author. Owner. Are any older than two years and unattributed? Are any using fragile criteria (formula fields, picklist values that have shifted, integration-populated fields)?
- **Permission set assignments.** Who has each permission set? Are there permission sets nobody uses? Permission sets that have grown — accumulated permissions that nobody removed when the original need passed?
- **Apex sharing.** What custom RowCauses exist? What code creates and deletes shares? Is the deletion code actually being called?
- **Profile drift.** Even with permission sets, profiles still hold some access. Have profiles drifted from each other in unintended ways?

I write the findings as a short memo. Three or four pages. The memo gets reviewed by whoever owns the security model — often me, sometimes a security architect like James, sometimes a committee. Decisions get logged as ADRs. The next quarterly audit checks whether the previous decisions are still being honoured.

Most quarterly audits surface nothing dramatic. The point is they exist. The Hannah morning would have been preventable if we'd been auditing sharing rules quarterly; we'd have flagged the fragile phone-format criteria and rewritten it before Spring '20 broke it.

## Closing

Sharing isn't a feature you bolt on at the end. It's the entire visibility model the org runs on. The decisions you make in week one — OWD settings, role hierarchy, sharing rule architecture, permission set layering — determine how every future feature behaves for every future user.

The Hannah story is the one I tell new architects when they're tempted to simplify the sharing review during design. *"It's just a sharing rule, we can fix it later"* is the framing that produced our Wednesday morning. We didn't fix it later. The platform changed underneath it, and we found out in the worst possible way.

Hannah was nice about it. Most users in that situation would have been less nice. Plan for the user who isn't.

---

# Chapter 4: Automation Architecture

Mei thought there was one trigger on the Account object. There were 23.

It took me about a week to work out the discrepancy. Mei was the admin on this project, and she was very good at her job in most respects. The Setup UI in classic Salesforce groups Apex Triggers by object — under "Apex Triggers > Account" you see one collapsible row. The collapsible row contains all the triggers, but unless you click in and look at the file count, you don't see how many. Mei hadn't clicked. She'd been adding triggers via change sets when developers asked her to deploy them, and each developer had thought they were modifying the existing trigger, and none of them had checked. The deployments were change set deployments, which don't fail when you accidentally create a new file with the same name.

The triggers ran in a non-deterministic order. About half of them depended on field values that other triggers in the same execution were modifying. The behaviour was, technically, defined — Salesforce ran them in a particular sequence based on metadata API timestamps — but the sequence was effectively random from the team's perspective.

Mei felt terrible when I told her. She'd been doing what the developers asked. The developers, several of whom had since left, had been doing what the previous architect had set up. The previous architect was the original sin; she'd left the change-set deployment workflow in place after the team had moved to source-driven development and never updated the runbook. Mei had been running the runbook.

We rebuilt the trigger structure over about three weeks. One trigger per object, the framework I've described elsewhere. Mei was fine — slightly mortified, but Mei is the kind of person who turns mortification into competence. By the end of the year she could give a better account of the trigger framework than half the developers.

This chapter is about automation architecture, which means it's mostly about which tool to reach for, in what order, and what happens when you reach for the wrong one. Salesforce gives you six tools (Flow, Process Builder, Workflow Rules, Apex Triggers, Async Apex, Platform Events) and most "best practice" docs tell you to pick one. You can't pick one. You pick all of them — but you pick them deliberately.

## The decision tree: config first, declarative next, code last

The order matters. Most teams skip ahead to code because the developer leading the project is most comfortable there. The order I run is:

**1. Config.** Validation rules, formula fields, page layouts, record types, approval processes. Can the requirement be met with platform configuration? Most can. The team that maintains config is broader (admins, business analysts, sometimes power users) than the team that maintains code (developers). Config-based solutions have lower maintenance overhead.

**2. Declarative automation.** Flows. Specifically record-triggered Flows for most things, screen Flows for guided processes, scheduled Flows for periodic work. Process Builder is deprecated; don't start anything new in it. Workflow Rules are deprecated; same. If declarative covers the requirement, use declarative.

**3. Apex.** When the requirement involves logic that declarative can't express cleanly, multi-object orchestration with complex branching, callouts to external systems, or work that genuinely needs the control Apex provides.

The trap is jumping from step 1 to step 3 because the developer thinks Apex is "more powerful" or "more maintainable." It's neither, in most contexts. A Flow that does what's needed is a better answer than an Apex trigger that does what's needed, in 80% of cases. The remaining 20% is where Apex earns its keep.

## Why three triggers on one object is a code smell

When I see multiple triggers on the same object, I assume one of three things:

1. The team didn't realise they were creating new triggers (the Mei case)
2. The team had a deliberate reason but didn't document it
3. The team is using a deprecated pattern from a tutorial that hasn't been updated since 2018

Salesforce doesn't guarantee execution order across multiple triggers on the same object. The order is *defined* — the platform runs triggers in metadata API timestamp order — but the order is fragile. Adding or modifying a trigger changes its timestamp. Deploying an unrelated trigger can shift the order. Behaviour that worked in dev fails in production for reasons nobody can trace.

The fix is the same as it's always been: one trigger per object, doing nothing except dispatching to a handler class. The trigger file should be five to ten lines long.

```
trigger AccountTrigger on Account (
    before insert, before update, before delete,
    after insert, after update, after delete, after undelete
) {
    new AccountTriggerHandler().run();
}
```

That's the entire trigger. All the actual logic lives in `AccountTriggerHandler`, which extends a base class that knows how to dispatch to the right method based on `Trigger.operationType`.

## The trigger framework, in full

I've written about this elsewhere — the trigger framework that actually scales covers it in detail — but for completeness here's the structure I bring to every project.

The base handler class:

```
public abstract class TriggerHandler {

    private static Map<String, Boolean> bypassedHandlers = new
Map<String, Boolean>();
    private Set<Id> processedIds = new Set<Id>();

    public void run() {
        if (isBypassed()) return;

        switch on Trigger.operationType {
            when BEFORE_INSERT { beforeInsert(); }
            when AFTER_INSERT { afterInsert(); }
            when BEFORE_UPDATE { beforeUpdate(); }
            when AFTER_UPDATE { afterUpdate(); }
            when BEFORE_DELETE { beforeDelete(); }
            when AFTER_DELETE { afterDelete(); }
            when AFTER_UNDELETE { afterUndelete(); }
        }
    }

    protected virtual void beforeInsert() {}
    protected virtual void afterInsert() {}
    protected virtual void beforeUpdate() {}
    protected virtual void afterUpdate() {}
    protected virtual void beforeDelete() {}
    protected virtual void afterDelete() {}
    protected virtual void afterUndelete() {}

    private Boolean isBypassed() {
        String handlerName =
String.valueOf(this).substringBefore(':');
        if (bypassedHandlers.containsKey(handlerName)) {
```

```
            return bypassedHandlers.get(handlerName);
        }
        Trigger_Setting__mdt setting =
Trigger_Setting__mdt.getInstance(handlerName);
        Boolean disabled = setting != null && setting.Disabled__c;
        bypassedHandlers.put(handlerName, disabled);
        return disabled;
    }
}
```

A concrete handler:

```
public class AccountTriggerHandler extends TriggerHandler {

    protected override void beforeInsert() {
        AccountValidator.validateNew((List<Account>) Trigger.new);
        AccountEnricher.enrichDefaults((List<Account>) Trigger.new);
    }

    protected override void afterUpdate() {
        AccountSyncService.syncToExternal(
            (List<Account>) Trigger.new,
            (Map<Id, Account>) Trigger.oldMap
        );
    }
}
```

The handler delegates to service classes. Each service does one thing. The trigger handler itself contains no business logic — only dispatch.

The bypass mechanism uses `Trigger_Setting__mdt`, a custom metadata type with a `Disabled__c` boolean per handler. An admin can disable a specific handler during a data migration or bulk load by flipping the metadata record. No code change. No deployment. Auditable via Setup Audit Trail.

That bypass is the feature most non-architects don't know to ask for. It's the difference between a data migration that takes a day and one that takes a week of working around the existing automation.

## Record-triggered Flows vs Apex

The conventional wisdom — that Flows are easier to maintain than Apex — is mostly true and frequently misapplied. Flows are easier to read for declarative changes. They're harder to debug when something goes wrong, harder to test with confidence, and slower in some bulk scenarios.

The decision framework I use:

**Use a record-triggered Flow when:**

- The logic is straightforward field updates or related-record creation
- The team includes admins who'll maintain it
- The volume is moderate (under a few thousand records per transaction)
- There's no callout, no complex branching, no error-handling path

**Use Apex when:**

- The logic involves callouts (Flows can do callouts via Invocable Apex but it's awkward)
- You need fine-grained governor limit management
- You're doing complex bulk processing where Apex's collection performance matters
- You need to throw and catch typed exceptions
- You're orchestrating across multiple objects with conditional branching that would be unreadable in Flow Builder

A pattern I like: Flow at the top, Apex via Invocable when Apex is needed. The record-triggered Flow handles the simple cases declaratively, then calls an Invocable Apex method for the parts that need code. The team can see the Flow and understand the high-level orchestration. The developers maintain the Apex. Both serve their audience.

```
public class OpportunityActions {
    @InvocableMethod(label='Sync Opportunity to External System')
    public static List<Result> syncToExternal(List<Request> requests)
{
        // ... Apex logic
    }
}
```

Mei's archive of 40-some Flows had several that should have been Apex — they were doing things Flow couldn't do well, like multi-step retry logic and callout error handling. We didn't migrate them all. We migrated the worst three and left the rest. Some of them are still called things like `Acct_Notif_v2_FINAL_USE_THIS`. Mei jokes about it. The naming is bad. The Flows work. We let it be.

## Async patterns

Some work doesn't belong in the synchronous transaction that triggered it. Long-running computation, callouts that might be slow, work that needs to span more than the synchronous governor limits — these get pushed async.

Salesforce gives you four async tools: Future methods, Queueable Apex, Batch Apex, and Scheduled Apex. I've covered the decision tree in detail in the async Apex article, so I'll just give the short version here:

- **Default to Queueable.** Modern, flexible, chainable, returns a job Id you can track.
- **Use Batch Apex when you genuinely need to process more than 50,000 records.** Below that, Queueable handles it cleanly.
- **Use Scheduled Apex only for cron-style timing.** Don't use it as a delay mechanism.
- **Avoid Future methods.** They're legacy. The few cases where Future is the only answer are rare and worth documenting explicitly.

The trap I see most often: someone uses a Future method because the syntax is shortest, hits the 50-Future-per-transaction limit when bulk operations come through, and now has a production incident at the worst possible time. Aaron, a new grad on a team I led, did exactly this his first week.

He'd written a `@future(callout=true)` method to send a notification when an Opportunity changed stage. In dev, it worked. In production, on the first day a sales manager bulk-edited 60 opportunities, it failed silently for the records past the Future limit. Sales support got 47 notifications instead of 60. Aaron noticed because his own demo opportunity didn't get a notification. We rolled it back to a Queueable that processed all the records in one async invocation. Aaron has been very thorough about Friday deploys ever since. He's a senior dev now.

## Don't

Future methods. That's it. That's the section.

You can use them — there are still a few specific cases where Future is the right answer — but the default should be no. Queueable does everything Future does, with better ergonomics, better testing, and better failure handling.

If you're maintaining a codebase with existing Future methods, plan a migration. It's a small migration; most Future methods can be converted to Queueable with about 10 lines of structural change. The migration prevents a class of bugs (the 50-per-transaction limit, the no-chaining limitation, the no-monitoring problem) that you'll otherwise hit eventually.

## When the framework isn't the answer

I'll be honest: not everything belongs in a trigger. Or in Apex at all.

The trigger framework I've described is heavyweight. It assumes you have an architectural reason to need it — multi-object orchestration, complex business rules, integration callouts. If your org has three triggers across the entire codebase, none of them complex, all of them written by the same senior developer, the framework is overhead. Don't introduce it.

I made this mistake on a financial services project. Daniel — same Daniel from the rule engine in Chapter 1, by the way, we ended up working together again at a different place years later — was the senior developer on the team. The org had three triggers. They were tidy, well-tested, and Daniel maintained them all personally. I came in and started rolling out the framework. By week two, Daniel was asking me, very politely, whether the indirection was actually solving a problem they had.

It wasn't. I rolled it back. Wrote a one-page document explaining when they'd want to introduce something like the framework in the future — the kind of "future-state" guidance that lets the team adopt the pattern when they actually need it, not when an architect forces it. Daniel was relieved. The team kept their three tidy triggers. The org has eight triggers now, four years later. They added five more without ever needing the framework. The framework would have been a tax on every one of those additions.

The framework is for the org with 23 triggers, not the org with three.

## Closing

Mei's Flows mostly work. We replaced the worst three of them. The rest are fine, even though I'd rebuild them differently. Some of them are still called `Acct_Notif_v2_FINAL_USE_THIS`. Mei has retired most of the v1s and v2s but the v3s persist because the v3s are the ones that work and renaming them would require regression testing she doesn't have time for.

I checked back with her last year. She told me one of the FINAL_USE_THIS Flows was still doing exactly what it had been doing in 2019. *"It's not pretty, but it works,"* she said. I think that's right.

---

# Chapter 5: Apex Patterns That Scale

There's an Apex class on a financial services org I once worked at called `OpportunityServiceManagerHelperUtil2_v3.cls`.

The "v3" was because there'd been a v1 and a v2. The "Final" wasn't in the name but might as well have been. Inside, it was about 4,000 lines of unrelated static methods someone had been bolting on for years. The original author had left the company. I asked the team what was in it. Nobody knew. I asked who used it. Several people said *"I think I use one method? I'm not sure which one."*

We deleted about half of it after a quiet two-week audit. Nothing broke. Sanjay — a developer who'd joined the team after the original author left — eventually wrote a proper service class for the parts that mattered. The original `OpportunityServiceManagerHelperUtil2_v3` is still in the codebase as a thin shim that re-exports the few methods anyone still calls. We've been meaning to delete it. We probably won't.

That class is the Apex anti-pattern in physical form. Naming that admits the file has been "finalised" multiple times. Length that exceeds anyone's working memory. Static methods bolted on by people who didn't know what was already there. No coherent responsibility. No tests that actually mean anything.

Most Apex codebases drift toward something like that without architectural discipline. This chapter is about the patterns that prevent the drift, the patterns that work in production at scale, and the test discipline that catches the bugs you actually need caught.

## The Selector / Service / Domain layer pattern

Apex codebases that survive five years tend to have one structural thing in common: they organise code by responsibility rather than by file size or proximity-to-the-trigger.

The pattern I run is three layers:

**Selectors** know how to query. One selector per object. Every SOQL query against Account goes through `AccountsSelector`. Every query against Opportunity goes through `OpportunitiesSelector`. The selector exposes typed methods like `selectOpportunitiesForAccount(Set<Id> accountIds)`, returns proper collections, and knows about the field set the rest of the codebase needs. When a query needs to change — add a field, filter differently, optimise — there's exactly one place to change it.

**Services** know how to do things. Multi-object orchestration, business logic that touches several layers, transformation work. `OpportunityRenewalService` handles the renewal flow. `AccountMergeService` handles account consolidation. Services are stateless, take collections as input, perform their work, and exit. They don't know about triggers. They don't know about UI. They get composed by the trigger handlers and the controllers.

**Domain classes** know about a single object's behaviour. `Opportunities` (plural — the collection) knows what an Opportunity is, what fields are required, what state transitions are valid. The domain class is the place where "an Opportunity must have a Close Date" lives — not in a validation rule (though a validation rule may also enforce it), and not in a service.

This is not novel. The fflib library popularised the pattern years ago. You can use fflib if you want; you can also write the layers yourself in about 200 lines of base classes. The pattern matters. The library doesn't.

A concrete example, simplified:

```
public class OpportunitiesSelector {
    public List<Opportunity> selectByAccountIds(Set<Id> accountIds) {
        return [
            SELECT Id, Name, AccountId, StageName, Amount, CloseDate
            FROM Opportunity
            WHERE AccountId IN :accountIds
            WITH SECURITY_ENFORCED
        ];
    }
}

public class OpportunityRenewalService {
    public void processRenewals(Set<Id> opportunityIds) {
        OpportunitiesSelector selector = new OpportunitiesSelector();
        List<Opportunity> opps = selector.selectByIds(opportunityIds);
        // ... transformation logic
        update opps;
    }
}
```

Trigger handler dispatches to the service. Service uses the selector. Selector handles the query. Three roles, three classes, no business logic in the trigger and no SOQL in the service.

The first three times you write this pattern, it feels like ceremony. The fourth time, you'll wonder how you ever lived without it.

## SOQL patterns that don't kill governor limits

I've covered the SOQL antipatterns in detail in the SOQL article, so this section is a summary. The key things:

**Queries inside loops are still the most common bug.** Even when there's no obvious loop, watch for helper methods called from inside loops. The query is one stack frame down but it still fires per-iteration.

**Selectivity is data-dependent and changes over time.** A query that's selective today can become non-selective when the data shifts. Filter on indexed fields where possible. CreatedDate is your friend.

**Aggregate queries beat looped counting.** `SELECT SUM(Amount) FROM Opportunity` runs in the database. Looping records to compute a sum is slower and hits row limits.

**SOQL `for` loops handle bulk processing cleanly.** `for (Account a : [SELECT ... FROM Account])` chunks 200 records at a time without you having to manage it.

**`WITH SECURITY_ENFORCED` is the cheapest CRUD/FLS enforcement you can write.** Use it on every selector query unless you have a documented reason to bypass it.

A pattern I've started using that's not in the article: write SOQL queries with explicit field lists, never `*`-style queries. SOQL doesn't have wildcards, but people sometimes write extremely wide queries because "we might need all the fields." Wide queries cost more in heap, more in network, and cache less effectively. Pick the fields you need. Add fields when the need arises.

## Test patterns that catch real bugs

The test suite on that financial services project — the one with `OpportunityServiceManagerHelperUtil2_v3` — was 92% covered. The team was proud of the coverage number. The first time I worked through the test classes with anything resembling rigor, I found that several of the tests were asserting on the wrong field name. The field had been renamed about 18 months earlier. The tests had been silently passing on null values ever since. Coverage was high. Validation was zero.

The fix wasn't more tests. It was tests that actually meant something. I covered the patterns in the Apex test patterns article, but the architectural points worth repeating:

**Test the contract, not the implementation.** A test that asserts "the method returns something non-null" is testing that the method exists. A test that asserts "the method returns a record with

`Type = 'Premium'` when given an Account with `AnnualRevenue > 5,000,000`" is testing the contract.

**Use a test data factory.** Don't construct test records inline in every test method. Build a `TestDataFactory` class with sensible defaults plus an override pattern, and use it everywhere. When a validation rule changes and the default Account data needs an extra field, you update the factory once. All tests adapt.

**Always include the bulk case.** 200 records is the default. If your code processes integration data that can come in larger batches, test with the realistic upper bound. The bulk test catches the SOQL-in-loop bug, the DML-in-loop bug, the heap-blowup bug, the recursion-depth bug. None of these show up in single-record tests.

**Assert governor limits explicitly.** `Limits.getQueries()`, `Limits.getDmlStatements()`, `Limits.getHeapSize()`. The platform tracks them; your tests should too. A test that processes 200 records and uses 300 SOQL queries is a test that's about to fail in production.

The Sanjay story belongs here, because it's about test failure rather than code failure.

Sanjay had been on the team for about a year and was leaving for another job. His last task was a refactor — clean up the service layer, consolidate some redundant code, improve the test data factory. He was thorough and technically excellent. On his last day, he committed his final changes, ran the test suite locally (all green), pushed, and went to his farewell drinks.

The CI pipeline ran the tests. Four of them failed. The pipeline emailed the team the failure report. The email went to a distribution list — `dev-team@`. The distribution list had been deleted six months earlier when the team had been reorganised. Nobody had updated the CI config. The failure email bounced into a void.

Sanjay's PR sat as "merged but failing tests" for about 48 hours before someone noticed. By then he was gone. The four failing tests turned out to be testing real bugs in his refactor. We rolled the changes back, fixed them with two days of careful work, and re-merged. Sanjay still apologises in his Christmas messages, even though it wasn't really his fault — the broken email routing was the actual failure mode.

The lessons:

- **Test failures need a path to a human.** A failing test that emails into a void is worse than no test, because it creates the illusion of safety.
- **Verify the alert path quarterly.** The team that did the reorg should have updated the email distribution. They didn't. The CI config never got reviewed. Six months later, real failures were silent.

- **Don't merge with failing tests, ever.** The PR shouldn't have been mergeable with red tests, regardless of whether anyone got the email. Branch protection rules prevent this. Use them.

## Error handling beyond try-catch

Apex's exception model is fine for simple cases — wrap the risky operation, catch the exception, log or rethrow. Most error-handling code I see does roughly that. Most of it is not very good.

The problems with naive try-catch:

**Swallowed exceptions.** `catch (Exception e) { System.debug(e); }` is the most common antipattern. The exception is captured, logged to the debug log (which nobody reads), and the calling code continues as though nothing went wrong. Six months later, the data is corrupt and nobody knows when it started.

**Generic exception messages.** `throw new MyException('Error processing record')` tells the caller nothing useful. The next developer who has to debug this gets the message, the stack trace, and zero context.

**Lost context in async code.** A Queueable that fails silently because nobody's monitoring AsyncApexJob is a recurring pattern. The job fails. The transaction rolls back. Nothing notifies anyone. The data that should have been processed isn't.

The pattern I run:

**Custom exception hierarchy.** A base `AppException` class with subclasses for specific failure modes — `ValidationException`, `IntegrationException`, `ConfigurationException`, etc. Each carries enough context (record IDs, field values, source system, timestamp) that the next developer can actually debug.

```
public abstract class AppException extends Exception {
    public Map<String, Object> context = new Map<String, Object>();

    public AppException withContext(String key, Object value) {
        context.put(key, value);
        return this;
    }
}

public class IntegrationException extends AppException {}

// Usage:
```

```
    throw new IntegrationException('Failed to sync Account to Xero')
        .withContext('accountId', a.Id)
        .withContext('xeroResponseCode', response.getStatusCode())
        .withContext('xeroResponseBody', response.getBody());
```

**A logging service.** Don't `System.debug` errors. Write them to a custom `Application_Log__c` object (or send them to an external log aggregator). Include the user, the context, the stack trace, the input data. The log object is queryable. The log object is dashboardable. The log object survives transactions.

**Async failures notify someone.** If a Queueable fails, write to the log object before rethrowing. If a Batch Apex job's `finish` method runs without errors but processed records have errors, write those to the log. Set up a daily report or alert on `Application_Log__c` records with severity = error. Someone needs to see it.

This pattern adds maybe 40 lines of code per project. It pays back the first time something goes wrong in production.

## Dynamic SOQL done safely

Sometimes you have to build SOQL strings dynamically. Generic search components, admin tools, integration endpoints that take filter parameters from external callers. These cases exist. Most of them are misuses, but the legitimate cases are real.

The rules:

**Bind variables, not concatenation.** Always.

```
// Wrong
String soql = 'SELECT Id FROM Account WHERE Name = \'' + userInput +
'\'';

// Right
String soql = 'SELECT Id FROM Account WHERE Name = :userInput';
List<Account> results = Database.query(soql);
```

The right version is automatically protected against SOQL injection. The wrong version is one apostrophe in `userInput` away from disaster.

**Whitelist field names.** If the dynamic part of the query is a field name (e.g. user picks which field to filter on), don't trust the input. Validate it against a known list.

```
private static final Set<String> ALLOWED_FILTER_FIELDS = new
Set<String>{
    'Name', 'Industry', 'Type', 'BillingCountry'
};

public static List<Account> filteredAccounts(String fieldName, String
value) {
    if (!ALLOWED_FILTER_FIELDS.contains(fieldName)) {
        throw new ValidationException('Invalid filter field: ' +
fieldName);
    }
    String soql = 'SELECT Id, Name FROM Account WHERE ' + fieldName +
' = :value';
    return Database.query(soql);
}
```

**Use `Database.queryWithBinds` or `WITH SECURITY_ENFORCED` for FLS.** Dynamic SOQL bypasses the standard FLS enforcement unless you opt in. The security scanner will flag your code if you don't, and rightly so.

The default should be: don't use dynamic SOQL. Static SOQL is faster, safer, and easier to read. Reach for dynamic SOQL only when there's a genuine reason — the field set or filter is unknown until runtime — and document the reason in the code.

## Custom Settings vs Custom Metadata Types

This one is short because the answer is almost always Custom Metadata.

**Custom Settings** are key-value pairs stored as data. They're admin-configurable, fast to read, and integrate with the platform's caching. They don't deploy with metadata. They don't version-control cleanly. They're invisible to most architecture review tools.

**Custom Metadata Types** are key-value pairs stored as metadata. They deploy with the project. They version-control. They show up in change sets, in source-driven deployments, in metadata API exports. They're slightly slower to read (one SOQL per type per transaction, cached after first read), but the difference is measured in milliseconds.

For anything that's *configuration* — the kind of data that should travel with code through environments — use Custom Metadata. For anything that's genuinely *data* that admins need to set in production without involving developers — use Custom Settings.

Most things that look like Custom Settings should be Custom Metadata. The exceptions are: per-user overrides (Custom Settings have a hierarchy mechanism that's hard to replicate in Metadata),

runtime-changing values that admins need to flip without a deployment, and integration credentials that genuinely shouldn't ship with code.

The trigger bypass example from Chapter 1 is the canonical case for Custom Metadata. The bypass is configuration. It travels with deployments. It's auditable. Custom Settings would be wrong here even though they'd technically work.

## Closing

Apex isn't a general-purpose language. It runs in a multi-tenant container with hard limits, and the patterns that work elsewhere often break here. The patterns that work here often look weird elsewhere. Learn the local rules.

The codebase that survives five years is the one where someone enforced the layering, kept the tests honest, and resisted the urge to bolt static methods onto a `_Helper_v3.cls`. That's not novel architectural thinking. It's just discipline applied consistently — and the novelty most developers reach for is what fills the gap when discipline gets hard.

The shim version of `OpportunityServiceManagerHelperUtil2_v3` is still in that codebase, by the way. Sanjay sent me a screenshot of it last Christmas with the message "*it lives*." I think we're going to delete it next year. We've been going to delete it next year for about three years.

---

# Chapter 6: Integration Architecture

The integration broke at 11pm on a Sunday. I know the time precisely because the page came through to my phone while I was watching TV, and I remember thinking *"that's a really specific 11pm."*

It was a financial services org, and the integration was the one that synced customer records to a legacy banking platform. The platform was the one that mattered — if it didn't have current data, the customer service team couldn't help anyone, and the bank's call centre would start the next morning with several thousand pending requests. The error message that came through was technically informative and practically useless: *"Connection failed. Trust verification error."*

I called Daniel — same Daniel, by then we were working at the same client again — and asked him to help diagnose. He found the problem in about 90 minutes: the bank had rotated their TLS certificate over the weekend as part of a routine compliance update. They'd notified their internal IT team. They had not notified the integration teams. The new certificate had a slightly different chain, our code was pinning against the old one (a previous developer had hardcoded the chain validation), and every callout was failing the verification.

Daniel disabled the chain pinning. Reran the integration. Recovered the records that had been queued for retry. Went to bed at 3am. He sent me a Slack the next morning saying *"all good, but we need to talk about whoever pinned that certificate."*

We had the talk. The developer was no longer at the company. The fix involved removing the pin and adding proper monitoring. I wrote an ADR explaining why we don't pin certificate chains in callouts. Six months later a different team did the same thing on a different integration, citing a different security paper, and we had the conversation again. Sometimes the lessons don't take.

This chapter is about integration architecture, which is mostly about contracts. Every integration is a contract with another system. Most failures happen when one party changes the contract and forgets to tell the other party. Architects can't fix that. We can architect for it.

## The three integration modes

There are roughly three ways Salesforce talks to other systems. Each has different trade-offs.

**Real-time** — the calling system needs an answer immediately. Salesforce makes a callout, waits for the response, and continues based on what came back. Or another system calls Salesforce and expects a synchronous answer. This is the model for things like address validation, payment processing, real-time pricing, anything where the user is staring at a screen waiting.

Real-time is fragile. The integration is only as reliable as the slowest external system. If their service is down, your transaction fails. If their service is slow, your user waits. If their service rate-limits you, you get cascading failures. Real-time integrations need timeouts, circuit breakers, and graceful degradation.

**Batch** — the calling system processes records in bulk on a schedule. Nightly customer data sync, weekly pricing updates, monthly reconciliation. The integration runs, processes a defined window of records, completes. Failures are easier to recover from because the next run picks up where the previous one left off.

Batch is the workhorse of Salesforce integration. Most enterprise integrations should be batch by default unless there's a specific reason for real-time. The reasons are usually clear ("the user can't wait"); when they're not clear, batch is cheaper to operate.

**Event-driven** — something happens on one side, the other side gets notified. Platform Events. Change Data Capture. Webhooks. Custom pub-sub patterns. The producer doesn't wait for the consumer. The consumer can be slow, can be down, can replay events from earlier — the producer doesn't care.

Event-driven is the most architecturally clean for systems that need to stay loosely coupled. It's also the most operationally complex. Events can be dropped if subscribers can't keep up, replayed out of order, duplicated under retry. Designing event-driven architectures well requires more upfront thinking than batch, but the runtime resilience is dramatically better.

The right choice depends on the use case. Real-time for user-facing latency-sensitive operations. Batch for bulk data work. Event-driven for system-to-system coupling that needs to be reliable but not synchronous. Most large orgs have all three. The architectural mistake is using one mode for everything because that's the team's habit.

## REST API patterns and the inbound/outbound question

Most integrations involve REST APIs at some point. Salesforce has rich REST API support both as the caller (Salesforce calls another system) and as the callee (another system calls Salesforce). The patterns are different for each direction.

**Outbound (Salesforce calls external):**

Always use Named Credentials. Always.

```
HttpRequest request = new HttpRequest();
request.setEndpoint('callout:Xero_API/api.xro/2.0/Invoices');
request.setMethod('POST');
request.setHeader('Content-Type', 'application/json');
request.setBody(payload);
```

```
Http http = new Http();
HttpResponse response = http.send(request);
```

That `callout:Xero_API` is a Named Credential. The credential stores the endpoint URL, the auth method, the OAuth token, the certificate trust. None of these live in code. None of them can be accidentally checked into source control. None of them get hardcoded in a way that breaks when the URL changes.

The previous developer who pinned the certificate? That happened because they weren't using a Named Credential properly. They'd built their own HTTP request construction, hardcoded the trust chain, and shipped. If they'd used the platform's Named Credential — which handles trust at the platform level — the certificate rotation would have been a non-event.

**Inbound (external calls Salesforce):**

Salesforce gives you a few options: REST API endpoints (built-in), Apex REST classes (custom), or Connect API for community-related operations.

For most cases, Apex REST classes are the right answer:

```
@RestResource(urlMapping='/customer/*')
global class CustomerAPI {

    @HttpGet
    global static Customer__c getCustomer() {
        String customerId =
RestContext.request.requestURI.substringAfterLast('/');
        return [
            SELECT Id, External_ID__c, Name, Email__c
            FROM Customer__c
            WHERE External_ID__c = :customerId
            WITH SECURITY_ENFORCED
            LIMIT 1
        ];
    }

    @HttpPost
    global static String createCustomer(Customer__c customer) {
        // Validate, transform, insert
        upsert customer External_ID__c;
        return customer.External_ID__c;
```

```
    }
}
```

The REST class becomes available at `https://yourorg.my.salesforce.com/services/apexrest/customer/...`. The caller authenticates via OAuth, just like any other Salesforce API consumer.

The traps with inbound integrations:

- **Authentication is the responsibility of the caller, but authorisation is yours.** Check that the calling user has access to the data they're requesting. `WITH SECURITY_ENFORCED` helps here.
- **Rate limiting.** The platform has limits on inbound API calls. Heavy callers will hit them. Plan capacity, or use Bulk API for high-volume integrations.
- **Versioning.** External callers will continue calling your old endpoint forever. Version the URL (`/api/v1/customer`, `/api/v2/customer`) and never break a versioned endpoint.

## OAuth 2.0 flows — which one, when

OAuth has multiple flows. Most teams use one (User-Agent or Web Server) and don't realise the others exist or matter. The decision tree, briefly:

**Web Server flow** — user-facing apps that need to call Salesforce on behalf of a logged-in user. The OAuth dance involves a browser redirect. Use this for SaaS apps, customer-facing portals, anything where a human is initiating the request.

**JWT Bearer flow** — server-to-server, no user interaction. Salesforce calls another system, or another system calls Salesforce, using a signed JWT instead of a refresh token. No browser involved. Use this for system integrations.

**Username-Password flow** — deprecated. Don't use it. Salesforce will eventually disable it.

**Client Credentials flow** — service-to-service via a connected app. Newer than the other flows. Use it where it fits — usually replacing JWT Bearer for simpler setups.

For most server-to-server Salesforce integrations, JWT Bearer is the right answer. The setup involves generating a key pair, registering the public key with Salesforce, and signing JWTs with the private key when calling. The work is upfront. Once it's done, the integration runs without user intervention indefinitely.

The trap I see: teams using Username-Password because the docs they read were old. Migrate. The migration is not large. The risk of leaving it alone is that one quarter Salesforce decides to enforce the deprecation and your integration breaks at 2am.

# Platform Events vs Change Data Capture vs custom pub-sub

These three are related but different. The decision is usually based on what's emitting the event and how much control you have.

**Platform Events** — Salesforce-defined events that you emit explicitly via `EventBus.publish()`. The publisher controls the schema (it's a custom event type in the metadata). Subscribers can be Apex classes, Flow Builder Flows, or external systems via the CometD/Pub-Sub API.

Use Platform Events when:

- You control both ends and want explicit eventing
- You need typed, structured payloads
- You need replay (subscribers can request events from a particular position)

**Change Data Capture (CDC)** — Salesforce automatically emits events when records of supported objects change. No Apex needed; turn it on for the object, subscribers get notified.

Use CDC when:

- You want to react to standard CRUD events on standard or custom objects
- You don't need a custom payload (CDC events are constrained to record changes)
- The volume is moderate (CDC has limits on events per day)

**Custom pub-sub** — you build your own event distribution using a custom object and a polling subscriber. Or via a middleware platform like MuleSoft or Workato.

Use custom only when:

- Platform Events and CDC don't fit
- You need eventing semantics the platform doesn't support
- You're integrating with a system that has its own pub-sub conventions

I had a project where the team built a custom pub-sub on top of a custom `Event__c` object before realising Platform Events would have done the job. The migration took about three weeks. The team learnt to check the standard tools first.

# Middleware vs direct integration

The question of "should Salesforce talk directly to System X, or should we put a middleware layer between them?" comes up on every multi-system project.

The argument for direct: simpler. Fewer moving parts. Lower latency. Lower cost (no middleware platform fees). The argument for middleware: decoupling. Both systems can change without affecting each other. Middleware can transform, route, queue, retry. Operations team can monitor a single layer.

The right answer depends on:

- **How many integrations are there?** Two systems direct-connected is fine. Five systems with twenty pairwise integrations is a recipe for chaos. At some point — usually around 5-7 systems — middleware pays for itself.
- **How stable are the systems?** A volatile system that changes APIs frequently benefits from a middleware abstraction. A stable system can be direct-connected.
- **Who operates each system?** If the same team owns both, direct may be fine. If they're owned by different teams (or vendors), middleware gives each team an interface they control.
- **What's the volume?** Middleware adds latency. For high-volume real-time integrations, the latency may be unacceptable.

The heuristic I use: start direct, plan to add middleware when a third system needs to participate or when the integration is causing operational pain. Don't add middleware on day one because a senior architect at a big firm said you should. Most projects don't need it for the first year.

When you do need middleware, MuleSoft (Salesforce-owned) is the obvious choice for Salesforce-heavy environments. Workato is a strong alternative for less complex integration needs. Roll-your-own (a custom Heroku app, AWS Lambda functions, that sort of thing) is rarely the right answer unless you have a specific reason and the engineering capacity to maintain it.

## Retry, idempotency, and failure recovery

Every integration fails sometimes. The architectural question is how it fails and how it recovers.

**Idempotency** is the property that running the same operation twice produces the same result as running it once. Critical for retry. If your integration creates a record on each call, retrying on failure creates duplicates. If it upserts based on an External ID, retrying is safe.

Always design for idempotency. Use External IDs. Use upsert instead of insert. Use HTTP methods that have well-defined idempotency semantics (GET, PUT, DELETE are idempotent; POST isn't unless you explicitly design it to be).

**Retry strategies** depend on what failed:

- **Transient failures** (network blip, temporary unavailability, rate limit) — retry with exponential backoff. After 3-5 attempts, escalate.
- **Permanent failures** (validation error, missing reference data, bad payload) — don't retry. Log and alert.

- **Ambiguous failures** (timeout where you don't know if the operation succeeded) — retry only if the operation is idempotent. Otherwise check the destination system to see if it succeeded.

The Daniel-and-the-TLS-certificate case was a transient failure presenting as a permanent one. The retry logic kept trying. The retries kept failing because the certificate chain was the actual problem. The retries weren't useful — but they also weren't harmful, because the integration was idempotent. The records that didn't sync stayed queued. When Daniel fixed the underlying issue, the queue drained. The retry mechanism had bought us recovery time without making things worse.

**Failure recovery** is what happens when the retry isn't enough. The integration's been failing for an hour. Records are queued. Some external system hasn't received the data it needs.

Plan for this in advance. Write a runbook. The runbook should answer:

- How does the team know the integration is failing? (Alerting, dashboards, the page that wakes Daniel up at 11pm.)
- Where are failed records held? (A retry queue object, an Application_Log__c with status = 'Failed', a custom replay table.)
- How do you reprocess failed records? (A button on the queue object, an admin batch job, a Flow.)
- How do you tell stakeholders the integration is failing? (Slack channel, email distribution list — and make sure the distribution list isn't deleted, see Chapter 5's Sanjay story.)

Most integrations don't have a runbook. The first time something breaks, the architect-on-call writes the runbook in real-time at 11pm. Better to write it during design.

## The offshore team that interpreted the spec literally

I want to tell one specific story because it captures a class of integration failure that doesn't fit anywhere else in this chapter.

A few years ago I worked on a project where the integration build was being delivered by an offshore team. The lead developer was Ravi, who I'd never worked with before but who had a reputation for being thorough. The integration was syncing customer data between Salesforce and a homegrown legacy system, including phone numbers.

The spec — written by an architect who, fortunately, was not me — said:

> *"Phone numbers are stored as integers in the legacy system. Sync as integers."*

Phone numbers have leading zeros. International prefixes have plus signs. When you cast `+61 2 9000 0000` to an integer, you lose the leading characters. You also lose the plus, which means you can't tell whether the number is Australian, international, or what.

Ravi flagged it on the PR review. Specifically he wrote: *"Are we sure about integers? This will lose leading zeros and prefixes. Australian mobile numbers all start with 04 — those zeros will disappear."*

The architect overruled him. *"That's the spec. We don't deviate from spec."*

Ravi documented the concern in the PR comments and shipped the integration to spec. About six weeks later, Australia phone numbers were all in E.164-format-minus-the-leading-characters. We caught it before go-live but only just. Ravi was right.

The lesson is not "always trust offshore teams" or "always question the spec." The lesson is that *the person closest to the implementation often knows the spec is wrong before anyone else*. Architectural reviews need to listen to those flags. When the developer doing the work raises a concern, the right response is to stop and check, not to point at the document and continue.

I've worked with Ravi on two more projects since. He's now a tech lead at a different consulting firm. He told me recently that he uses the phone-number story when he interviews architecture candidates. He asks them what they would have done if their developer had flagged the issue. He says about half get it wrong.

## Closing

Every integration is a contract with another system. Read the spec. Then read it again. Then ask the people closest to the implementation if anything feels off.

Daniel still mentions the TLS certificate every now and then, usually when someone in a design review proposes pinning a chain. He doesn't argue. He just says *"oh, we tried that once,"* and lets the architect work out the rest from his face. The legacy banking integration has run cleanly for years now. The integration with the system that pushed phone numbers as integers got rebuilt to use proper string formatting before go-live, with Ravi's PR comment cited as evidence the original design was wrong. Ravi got his name on the post-mortem document, which is a small thing, but the kind of small thing he was happy to have.

I've been meaning to call Daniel about something else for about two months. I should do that.

---

# Chapter 7: Security Architecture

I'm going to write this chapter and then end it without resolving it, because the honest version of security architecture has at least one decision I'm still not sure I got right.

The decision was Shield Platform Encryption, on a financial services org I worked at for about three years. We turned it on. The security team was happy. The analytics team was, eventually, less happy. Looking back, I'm not sure we needed it. I'll come back to this at the end of the chapter.

But first, the framework that mostly works.

## Five layers, briefly

Salesforce security has five layers. Most architects can list them; what matters is how they compose.

**1. Network.** IP whitelisting, login hours, session settings, mutual TLS for inbound API calls. The platform's first line of defence. Configure it once per org, audit it quarterly.

**2. Authentication.** Password policy, MFA enforcement, SSO configuration, OAuth flows. Who can prove they're who they say they are.

**3. Authorisation (sharing).** Covered in Chapter 3. Who can see which records.

**4. Field-Level Security (FLS).** Who can see which fields on records they have access to.

**5. Encryption.** Whether the data at rest is readable by anyone who happens to query the database. Shield is the platform's answer here.

Most security architecture work happens in layers 3-4. Most security incidents happen because of layer 1 (someone bypassed network controls) or layer 2 (someone reused a password). The architect can't directly fix incidents in those layers — they're usually operational or organisational — but the architecture should make them less likely.

## CRUD/FLS enforcement: the `Security.stripInaccessible` pattern

I covered this in detail in the CRUD/FLS article. The short version:

Apex code runs with the executing user's CRUD/FLS unless you bypass it. By default, an Apex class doesn't enforce CRUD/FLS — meaning code can read fields the user can't, write to fields the user

shouldn't, and so on. The security review process catches this. Real production incidents catch it less often, but they do catch it.

The pattern that passes review and works in production:

```
List<Account> accounts = [SELECT Id, Name, Industry FROM Account LIMIT
100];
SObjectAccessDecision decision = Security.stripInaccessible(
    AccessType.READABLE,
    accounts
);
List<Account> safeAccounts = (List<Account>) decision.getRecords();
// safeAccounts has fields the user can't see stripped out
```

For DML:

```
SObjectAccessDecision insertDecision = Security.stripInaccessible(
    AccessType.CREATABLE,
    newAccounts
);
insert insertDecision.getRecords();
```

For SOQL queries, `WITH SECURITY_ENFORCED` is cleaner:

```
List<Contact> contacts = [
    SELECT Id, Email, Phone
    FROM Contact
    WHERE LastName = :surname
    WITH SECURITY_ENFORCED
];
```

`WITH SECURITY_ENFORCED` throws an exception at runtime if the user doesn't have access to any of the queried fields. It's strict; some teams find it too strict because it fails closed (any FLS gap is a runtime failure). I prefer it because the failure is visible. The alternative — silently stripping fields with `Security.stripInaccessible` — can mask bugs where the code expects a field that's not coming through.

## The centralised security utility

Across multiple AppExchange packages I've seen the same pattern emerge: a centralised `SecurityUtils` class that wraps every CRUD operation. Every `insert`, `update`, `delete` goes through the utility. Every dynamic SOQL goes through the utility. The utility handles the FLS enforcement; the rest of the code stays clean.

```
public class SecurityUtils {

    public static List<SObject> queryAccessible(String soql) {
        return Database.query(soql + ' WITH SECURITY_ENFORCED');
    }

    public static List<SObject> insertAccessible(List<SObject>
records) {
        SObjectAccessDecision decision = Security.stripInaccessible(
            AccessType.CREATABLE,
            records
        );
        insert decision.getRecords();
        return decision.getRecords();
    }

    public static List<SObject> updateAccessible(List<SObject>
records) {
        SObjectAccessDecision decision = Security.stripInaccessible(
            AccessType.UPDATABLE,
            records
        );
        update decision.getRecords();
        return decision.getRecords();
    }
}
```

The benefits compound:

- The security scanner sees explicit security-checked operations and stops flagging.
- New developers can't accidentally write unsafe code — the utility is the only path.
- If the security model changes, you fix it in one place.
- Code review becomes "did you go through `SecurityUtils`" — a yes/no question.

Ben — a developer I worked with on an AppExchange project — had been the original author of one such utility, on a package that came back from security review with **187 CRUD/FLS findings** on first submission. The findings were almost all in classes that didn't go through the utility. We rewrote those classes to go through it. Resubmitted. Passed on second attempt. Ben was, briefly, very annoyed. Then very thorough.

The utility is now mandatory on every AppExchange project I work on. If the team won't adopt it, that's a signal that the project doesn't have the security discipline to pass review.

## SOQL injection and bind variables

I covered this in Chapter 5 and won't repeat the details. The summary: dynamic SOQL with string concatenation is unsafe. Bind variables are safe. The discipline is to never concatenate user input into a query string, ever.

The platform's security scanner flags concatenation reliably. The runtime doesn't catch it; SOQL injection in production is silent until someone notices the data has been read or modified by an unauthorised query.

Audit your codebase for `Database.query(` calls. Check that every one uses bind variables. Make this a code review rule. Make it a CI check if your CI tooling supports it. The fix is mechanical; the cost of missing one is significant.

## Audit trail and Event Monitoring

Security architecture isn't only about preventing bad things. It's also about being able to investigate when something happens.

**Setup Audit Trail** logs every metadata change in the org. Who created the trigger, who modified the validation rule, who changed the sharing model. Available for the last six months by default. Export it monthly to long-term storage; you'll want it for incident investigations more than once.

**Event Monitoring** is a paid feature that logs application-level events: API calls, login attempts, report exports, file downloads. If your org handles regulated data, Event Monitoring is essentially required. If you don't handle regulated data, it's a nice-to-have until you have an incident, at which point it becomes essential.

**Field History Tracking** is the per-field audit trail. Limited to 20 fields per object and 18 months of retention by default. Use it on high-sensitivity fields (anything that triggers compliance review).

**Custom audit logging** via an `Application_Log__c` custom object covers what the platform's built-in mechanisms don't. Trigger handlers can log to it. Apex services can log to it. The

log is queryable, dashboardable, exportable. It's the safety net under the platform's audit capabilities.

The architectural rule I run: every action that affects regulated data — financial transactions, healthcare records, government data — must be logged in a way that's retrievable two years later. The default platform mechanisms don't always meet this standard. The custom logging usually does.

## Designing for compliance

Australian Salesforce orgs commonly need to handle:

- **APRA-regulated data** (financial services). Strong access controls, full audit trail, data residency requirements.
- **NDIS data** (disability services). Privacy Act obligations, quality standards.
- **NSW or federal government data**. PROTECTED-level handling for some classifications.
- **Privacy Act data generally** (any organisation with personal information). Notification of breaches, consent management.

The architecture for each of these isn't dramatically different. The patterns are similar: tight FLS, strong sharing, centralised CRUD enforcement, comprehensive audit trail, regular review. The differences are in the details — APRA wants quarterly access reviews, NDIS wants particular consent records, government wants data classified at field level.

When designing for compliance, the architect's job is to:

1. Read the actual regulatory requirements, not the consultant's summary.
2. Map each requirement to a specific platform feature or pattern.
3. Document the mapping in an ADR.
4. Build review processes that verify the mapping is being honoured.
5. Build a paper trail that demonstrates compliance during audits.

The trap is treating compliance as something to bolt on at the end. Compliance has to be designed in. The retrofitting cost is dramatically higher than the upfront cost.

Tara, a financial services PM I worked with for a year and a half, used to push back on me about Shield encryption. Her team had to use the data for regulatory reporting, and Shield-encrypted fields couldn't be aggregated in the way her reports needed. She wasn't wrong. We compromised. Some fields were encrypted, others weren't. The regulatory reports worked. The security team was satisfied. The compromise lived in an ADR explaining why specific fields weren't encrypted, with the regulatory reporting requirement documented as the justification.

Compromises are fine. Undocumented compromises are dangerous.

## A few specific things I've watched go wrong

**Apex code checking `UserInfo.getProfileId()` for years before someone noticed the profile cloning had broken the assumption.** This was a long-running project where the original assumption — that the "Service Manager" profile would always have a specific name — held for about three years. Then a migration cloned and renamed profiles, and the Apex started failing for the new "Service Mgr" profile (note the abbreviation). The fix is custom permissions: create a custom permission for the capability, assign it via permission set, and check `FeatureManagement.checkPermission()` instead of profile name.

**`without sharing` used as a default rather than an exception.** The keyword `without sharing` makes Apex run with full data access regardless of the user's sharing model. Useful in specific cases — system processes, integration callers — and dangerous as a default. I've seen codebases where every class is `without sharing` because the original developer didn't know the difference. Audit. Fix.

**Stored XSS via Lightning components rendering custom field content as HTML.** A custom field that holds user-input text gets rendered into a Lightning component without escaping. User puts a script tag in the field. Component runs the script. Modern Lightning frameworks auto-escape, but custom components and HTML formula fields don't always. Audit anywhere user input gets rendered.

**Hardcoded credentials in Apex.** API keys, OAuth secrets, integration credentials checked into source control. Security review will catch this. Production won't, until a developer accidentally commits the credentials to a public repo. Use Named Credentials. Use protected custom metadata. Don't put secrets in code.

**Sharing rules that depend on data formats that can change.** The Spring '20 Hannah story from Chapter 3 is the canonical example. Fragile criteria-based sharing rules accumulate. Audit them quarterly.

These are the recurring patterns. Most security incidents I've investigated trace back to one of them.

## The Shield Platform Encryption decision (the unresolved one)

OK. The story I said I'd come back to.

Financial services org. Three years there. The conversation about Shield happened about six months in. The security team had identified about a dozen sensitive fields — names, contact details, account references, a few free-text fields that occasionally contained sensitive information — and

asked us to encrypt them. The ask was reasonable. The regulatory framework arguably required field-level encryption for at-rest data.

Shield costs a lot. Implementation took about three months. We hit several issues along the way: deterministic encryption was needed for filterable fields but it weakens the encryption guarantee; some report types couldn't aggregate encrypted fields; CRM Analytics dataflows needed to be rebuilt; some integrations had to be reworked because the encrypted field representation in the API was different.

We got it working. Tara's team adapted their reporting. The security team was happy. The auditors were happy. The bank shipped its compliance attestations.

Three months after Shield went live, a routine release update broke a CRM Analytics dataflow that was aggregating Shield-encrypted fields in a way that worked yesterday but not today. The dataflow was the source of three executive dashboards. They were broken for about 36 hours while we worked around it. We worked around it. We documented the workaround. We added monitoring so we'd see it next time. There was a next time, eight months later, with a different dataflow.

Two years after Shield went live, I sat down to think about whether we'd actually needed it.

The regulatory framework that drove the decision was about data protection generally — encryption was one of several acceptable controls. We'd implemented Shield because Shield was the obvious answer, and because the security team wanted the strongest available encryption. Looking back, what the regulator actually required was *proof of access control and audit trail*. We had those independently — the sharing model was tight, the FLS was clean, audit logging was comprehensive. We could have argued, in writing, that the existing controls met the requirement. The regulator might have agreed. The audit might have passed without Shield.

Or it might not have. I don't actually know. The auditors might have insisted. The security team might have won the internal argument anyway. The compliance environment might have shifted in ways that retroactively justified the decision.

What I know is the team that has to query the data has spent dozens of hours working around the encryption. The reporting capability is more constrained than it would have been. New analytics features take more design work. We've made it work. It just costs.

I think we'd do something different now. Some combination of selective encryption (only the truly sensitive fields, not the moderately sensitive ones), better access controls, more comprehensive audit logging. The something different would also be imperfect. It would have its own costs. Sometimes architecture is just picking which problem you'd rather have.

It's been four years since we made that decision. The data is encrypted. The team that has to query it has spent dozens of hours working around the encryption. I think we'd do something different now. But the something different is also imperfect.

I still don't know if we got it right.

# Chapter 8: Building for AppExchange

I shipped my first AppExchange package about six years ago, and the security review came back clean on first submission. I want to say it was because I knew what I was doing. Mostly it was because I had a very thorough list, written by another architect named Ben, and I just followed the list. Most of what I now know about AppExchange came from preparing that submission and then doing it three more times for different clients.

The thing nobody tells you about AppExchange before you start: the packaging itself is mechanical. The hard part — where the project length actually lives — is security review, and the second-hardest part is the multi-tenant subscriber design that you don't realise you've got wrong until customers start installing your app and seeing it behave in ways you didn't predict.

This chapter is about both. If you're considering building for AppExchange, this is the version of the playbook I wish I'd had before my first submission.

## 2GP vs 1GP — migrate, full stop

This used to be a debate. It isn't anymore. Salesforce has been steering everyone toward 2GP (Second-Generation Packaging) for years, and the few advantages 1GP retained have largely closed.

If you're starting a new package today: 2GP. No exceptions. If you have a 1GP package: plan a migration. The migration is not trivial — you can't just "convert" a 1GP package to 2GP, you have to recreate it — but the longer you wait, the harder it gets.

The benefits of 2GP that matter:

- **Source-driven development.** Your package is a directory in your Salesforce DX project. It deploys like any other code. Version control works. CI/CD works.
- **Better dependency management.** 2GP packages can declare dependencies on other packages cleanly. 1GP can't, really.
- **Faster iteration.** Creating a new package version takes a few minutes. In 1GP it could take hours.
- **Modern Salesforce features.** Some newer platform features (some Lightning capabilities, certain OmniStudio features) only work in 2GP packages.

The migration takes weeks of work, not days. Plan it as a project, not a quick task. Most ISVs I've worked with under-estimate it by 50% or more.

## The namespace decision (and why you can't change it)

Before any code, before any package version, you need a namespace. The namespace is the prefix that gets attached to every custom object, field, and Apex class in your package. Once you've registered it, you cannot change it. Every customer install has the namespace baked into their data forever.

Constraints:

- 1-15 characters
- Letters and numbers only (no underscores, hyphens, special characters)
- Globally unique across all of Salesforce
- Not used as the start of an existing custom field in customer orgs (rare but happens)

Pick something:

- Short (15 characters is too long for a prefix you'll see thousands of times)
- Generic enough to outlast pivots
- Not too close to your current product name (because product names change)

I've seen ISVs name their namespace `acme1` because they thought their product would be called "Acme" forever. They pivoted six months later to something completely different. The namespace stuck. Every custom object in every customer org now has `acme1__` prefixes that don't match the new brand.

The registration happens in a Dev Hub org through the namespace registration form. It takes a few business days. Don't start packaging work until this is done; you can't change later.

A specific tip from someone who's submitted packages from multiple namespaces: choose a namespace you'd be comfortable having printed on tee-shirts. Because effectively, that's what you're doing.

## The project structure that works

A packaging project is, at its core, a Salesforce DX project with packaging metadata. The structure I use:

```
my-package/
├── sfdx-project.json          # Package definition
├── force-app/
│   └── main/
│       └── default/           # Source code
├── data/
│   ├── seed/                  # Test data scripts for scratch orgs
```

```
│   └── samples/                # Demo data for customer onboarding
├── scripts/
│   ├── package-create.sh       # Create new package version
│   ├── install-locally.sh      # Install latest in personal org
│   └── promote.sh              # Promote version to released
├── docs/
│   ├── security-review-prep/   # Security questionnaire drafts
│   └── post-install/           # Setup steps for subscribers
└── .github/workflows/
    ├── pr-validate.yml         # Run tests on PR
    ├── package-version.yml     # Build package version on merge
    └── promote.yml             # Manual workflow to promote
```

The minimal `sfdx-project.json`:

```
{
  "packageDirectories": [
    {
      "path": "force-app",
      "default": true,
      "package": "MyPackage",
      "versionName": "ver 0.1",
      "versionNumber": "0.1.0.NEXT",
      "definitionFile": "config/project-scratch-def.json"
    }
  ],
  "namespace": "myns",
  "sourceApiVersion": "60.0",
  "packageAliases": {
    "MyPackage": "0Ho..."
  }
}
```

The `0Ho...` ID gets generated when you first create the package via `sf package create`. After that, all subsequent versions are tracked through `packageAliases`.

## The version pipeline

The cadence for an active ISV product:

**On every PR:** scratch org spin-up, source push, Apex tests run, results posted to PR. Catches obvious failures before review.

**On merge to main:** a new Beta package version is created automatically. Package version ID committed back to `sfdx-project.json`. Every merge becomes installable for testing.

**On a manual workflow trigger:** Beta is promoted to a released version. Promotion is one-way. Released versions can never be deleted. This is the gate I'm careful about.

The critical thing about promotion: once a version is released, it's installed in subscriber orgs forever. You can release a newer version that supersedes it, but the old version exists permanently in the Salesforce ecosystem. Don't promote unless you're sure.

The Annature project — the one I worked on at Quantum Associates — had a strict rule: no promotion on Fridays, ever. The reason was the support burden. If a promotion went out and immediately revealed a bug in subscriber installs, weekend support was painful. We promoted on Mondays. By Friday, if there was a problem, the team could see it during business hours and fix it.

## Ancestor versioning, get this right once

When a customer has v1.0 installed and you want to install v1.1, the platform needs to know v1.1 is an upgrade of v1.0 — not an unrelated package. Ancestors handle this.

In `sfdx-project.json`:

```
{
  "packageDirectories": [
    {
      "package": "MyPackage",
      "versionName": "ver 1.1",
      "versionNumber": "1.1.0.NEXT",
      "ancestorVersion": "1.0.0.LATEST"
    }
  ]
}
```

The `1.0.0.LATEST` says "the latest released version in the 1.0.x line is the parent." When v1.1 is installed over v1.0, the platform performs an upgrade.

The trap: if you ever release a version without an ancestor, the chain breaks for that branch. Customers on v1.0 can't upgrade to v1.1 without uninstall-reinstall, which is painful. Always set the ancestor for every released version.

## The security review process

The packaging is mechanical. The security review is where the project length lives.

What to expect:

**Submission preparation.** You submit a package through the Partner Portal along with a security questionnaire. The questionnaire is several dozen questions about your architecture: how you handle CRUD/FLS, how you authenticate, how you store credentials, how you handle multi-tenant data isolation, what external systems you integrate with, etc. Each question has free-text response space. The reviewers read your responses.

**The scanner.** Salesforce runs a security scanner over your code. It identifies CRUD/FLS gaps, SOQL injection risks, hardcoded credentials, sharing violations, XSS vectors, and a few dozen other patterns. The findings come back as a list with file/line references.

**The review cycle.** First submission almost never passes for first-time submitters. Plan for 2-4 cycles of remediation. Each cycle is "scanner findings + reviewer questions + your response and fixes + resubmission." Each cycle takes a week or two.

**The most common findings:**

- CRUD/FLS not enforced (covered in Chapter 7)
- SOQL injection vectors (concatenation instead of bind variables)
- Hardcoded credentials
- `without sharing` used without justification
- Stored XSS in custom Lightning components or Visualforce pages
- Insecure direct object references (passing record IDs in URLs without verifying access)

**The questionnaire matters more than people realise.** Reviewers read it. Vague responses lead to follow-up questions. Specific responses with concrete patterns ("we centralise CRUD/FLS through `SecurityUtils.queryAccessible` — see this code reference") get accepted.

The Annature submission passed first review largely on the strength of the questionnaire. Ben had spent two weeks on it. Every architectural decision was documented. Every scanner finding had an explanation or a fix. The reviewer's response was unusually short: a few clarifying questions, then approval.

The same patterns work consistently. Document. Centralise. Be specific. Don't try to argue with the scanner; it's faster to fix than to debate.

## Designing for multi-tenant subscribers

Once your package is in customer orgs, you discover the constraints you didn't think about during development.

**Field-Level Security defaults to denied.** When a custom field in your package is installed in a subscriber org, no profile or permission set has FLS access to it by default. Customers have to grant access manually.

The fix: ship a permission set inside your package that grants FLS to the relevant fields. Customers assign the permission set. Their users get access. Document this clearly in your post-install guide.

**Sharing rules don't transfer.** If your package's records need specific sharing, you can't ship sharing rules — they have to be configured per-org. Document the sharing model the package expects. Make it clear in the post-install guide.

**Subscriber Apex limits are aggregate.** Subscribers have aggregate Apex code limits across all installed packages. A bloated package can prevent subscribers from installing other packages or writing their own Apex. Watch your compiled Apex size. The limit is generous (a few MB) but reachable.

**Standard fields can be edition-specific.** If your code references `Account.OwnerId`, that's standard everywhere. If it references something edition-specific (Person Account fields, Service Cloud fields, Field Service fields), it might not exist in all subscriber editions. Use `Schema.SObjectType` introspection to check at runtime:

```
if
(Schema.sObjectType.Account.fields.getMap().containsKey('IsPersonAccount'
{
    // Person Accounts is enabled in this org
}
```

I had a subscriber where a hardcoded reference to a field that exists in Person Accounts editions but not standard editions broke install. The fix took a week — we had to re-architect to be edition-aware. Always introspect.

**Custom metadata is the configuration layer.** Subscribers customise your package via custom metadata records, not by editing your code (which they can't anyway). Design your package's behaviour to be metadata-driven. A subscriber should be able to enable, disable, or reconfigure features without involving you.

## The LMA, push upgrades, and subscriber support

The License Management App (LMA) is what you use to track which customers have your package installed and what license tier they're on. Setting it up is fiddly:

1. Request access to the LMA from Salesforce Partners. Takes a couple of business days.
2. Install in your packaging org.

3. Configure the License object to track your customer base.
4. Set up Apex hooks in your package to enforce license tiers (if you have multiple tiers).

The LMA is also how you push upgrades. Once a version is promoted to released, you can push it to subscriber orgs of a particular customer or tier. Powerful, easy to misuse.

The discipline:

- Push upgrades only after the version has been in the wild for at least a week with active customer testing.
- Never on a Friday.
- Never to the entire customer base at once. Start with a small subset (an internal demo org, friendly customers, a few specific tiers).
- Have a rollback plan. Push upgrades that break customer orgs are a support nightmare.

I almost did a push upgrade on a Friday once. The decision was driven by an internal urgency that, looking back, wasn't actually that urgent. I stopped myself. Did it Monday morning. No incidents. Even if there had been incidents, the team was at full strength on Monday rather than scrambling on the weekend. Good call.

## What I'd skip on a first package

A few things to defer when you're shipping v1:

**Multi-tier licensing logic baked into Apex.** Build v1 with a single license tier. Add tier differentiation in v2 once you understand which features customers actually want gated. The wrong tier strategy is more painful to undo than to avoid.

**Custom domain and connected app for OAuth.** Use the platform's built-in OAuth flow for v1. The custom domain is a security review surface area that's not worth the cost early.

**Internationalisation.** Custom labels are fine for v1. Translate when you have non-English customers, not before.

**Unit tests for non-public methods.** Test through the public API. Pop the hood only when there's a specific bug. The security scanner doesn't care about internal coverage; it cares that the public surface is safe.

**Push upgrades.** v1 should be an opt-in install. Subscribers who want the package install it. They upgrade when they're ready. Skip the push upgrade infrastructure until you have a version that genuinely benefits from being pushed (a critical security fix, for example).

## Closing

An AppExchange package is software you'll never directly debug in production. Every customer's install runs in their own org, behind their authentication, with their data. You can't SSH in. You can't tail their logs. You can rarely even reproduce their bug locally.

Plan for that. Build instrumentation that customers can self-serve — a debug logging mode they can turn on, an export tool they can use to send you their state. Build documentation that answers their first ten support questions before they ask. Build a customer support pipeline that catches issues fast, because the alternative is a customer with a broken install and no way to diagnose it, and that kills your reputation faster than any individual bug.

The Annature package has been live for several years now. I'm not on it day-to-day anymore. Ben still is — he runs the platform team that maintains it. The patterns from the original submission are mostly still in place. The team has improved on a few of them, deprecated one or two, and added some I wouldn't have thought of. That's the right outcome for a package that's still actively being sold.

I keep an eye on the install count occasionally. It's healthier than I expected when we shipped v1.

---

# Chapter 9: DevOps for Salesforce

Aaron deployed to production at about 3pm on a Friday afternoon. He was four months into his first job out of university. He'd been working on a small feature — a Future method that sent a notification when an Opportunity changed stage — and the code had passed review, the tests had passed locally, the change set was ready, the deploy button was right there.

He pressed it.

The deploy succeeded. Production was live with the new code. Aaron got up to make coffee and take the small win to his manager. By the time he got back to his desk, his Slack had three new messages from the support team about emails that hadn't been sent. By 4pm we'd worked out the Future method limit had been hit. By 5pm we'd disabled the trigger that called the method. By 6pm Aaron and I were on a video call, him slightly pale, me trying to explain that this was fixable and he hadn't broken anything permanent. By 9pm we'd rolled out a Queueable replacement and the notifications were flowing again.

Aaron has been very thorough about Friday deploys ever since. He's now a senior dev at a different organisation. He told me last year that he uses the Future-method incident as a teaching story for the juniors he mentors.

The story isn't about Aaron. It's about the fact that production deployment was a button he could press at 3pm on a Friday with no organisational friction, no deployment window, no peer-review enforcement, no rollback plan rehearsed, and no automated check to flag the Future-method anti-pattern. The architecture failure wasn't Aaron's. It was the absence of guardrails.

This chapter is about the guardrails. DevOps for Salesforce is mostly about making the boring choice — careful, repeatable, automated deployments — easier than the exciting choice. When the boring path is the path of least resistance, deployments stop being events. They become routine.

## The three-stage pipeline

Every Salesforce DevOps pipeline I've built has three stages. The work is in the details, not the structure.

**Stage 1: Validate (per pull request).** Every PR triggers automated validation. The pipeline pulls the changed metadata, deploys it to a CI environment (scratch org or persistent sandbox), runs the Apex tests, and reports back on the PR. The PR cannot merge if validation fails. This is non-negotiable.

The validation should be fast. PR validation that takes 20+ minutes gets bypassed by developers who learn to merge without waiting. Cut to under 5 minutes via delta deployments.

**Stage 2: Deploy to staging (on merge to main).** When PRs merge, the pipeline automatically deploys to a staging sandbox. Staging mirrors production as closely as possible — same metadata, same Apex tests, same automation. Staging is the integration testing environment.

The staging deploy should run all local Apex tests. If tests fail, the pipeline alerts the team and blocks the production deployment. In a healthy pipeline this almost never fails because individual PRs were already validated.

**Stage 3: Deploy to production (manual approval).** Production deploys are not automatic. They're triggered manually by an authorised person, after staging has been verified. The deployment uses Quick Deploy from a previously-validated package, so the actual production change takes seconds rather than minutes.

The manual approval is the discipline. Auto-deploying to production sounds modern and reduces friction. It also removes the human moment where someone says *"yes, do this, now."* That moment matters. Reserve it.

## Scratch orgs vs persistent sandboxes

The platform offers two main flavours of dev environment, and the team is going to argue about which to use. The right answer is "both, deliberately."

**Scratch orgs** are disposable. You spin one up via the CLI in a few minutes, push your source, run tests, throw it away. Perfect for CI. Perfect for individual developer workspaces. Perfect for testing branches without contaminating shared environments.

The trade-off: scratch orgs don't have your real data, don't have your full org's metadata (you have to install dependencies), and don't replicate every Salesforce feature. Some things — Knowledge articles, certain managed package configurations, specific feature licenses — are awkward in scratch orgs.

**Persistent sandboxes** are stable. They have your real metadata, your test data, your integrations pointing at them. They're slower to refresh (a full sandbox refresh can take hours), but they're closer to production reality.

The hybrid approach I run:

- **Local development:** scratch orgs. Each developer has one. They throw it away when they're done.
- **PR validation:** scratch orgs. Spun up by CI, used briefly, destroyed. (Or, if your team has bumped against scratch org limits, a persistent CI sandbox with check-only deployments.)
- **Integration testing:** persistent sandbox. The team's "this is real" environment. Refreshed monthly from production.

- **UAT:** persistent sandbox. Stakeholders use it. Refreshed before each major release.
- **Staging:** persistent sandbox. Production-like. Last stop before production.

Scratch org limits matter. The default limit is 10 concurrent active scratch orgs and 200 daily creations across the Dev Hub. Active teams will hit this. Salesforce will increase the limit if you ask, but the request takes a couple of weeks. Plan for it.

## Delta deployments

The single biggest thing that makes Salesforce CI fast: delta deployments. Don't deploy the entire codebase on every PR. Deploy only what changed.

The tool I use is `sfdx-git-delta` (the `sgd` plugin). It compares two git refs (your PR branch vs target branch) and generates a `package.xml` containing only the metadata that actually changed. The CI pipeline deploys just that.

A typical PR validation workflow:

```
- name: Generate delta package
  run: |
    sf sgd:source:delta \
      --to "HEAD" \
      --from "origin/main" \
      --output ./delta \
      --generate-delta

- name: Deploy delta to scratch org
  run: |
    sf project deploy validate \
      --source-dir ./delta/force-app \
      --test-level RunLocalTests \
      --target-org scratch-org-alias

- name: Run only relevant tests
  run: |
    # Test class names extracted from changed files
    sf apex run test \
      --tests $(extract-test-classes ./delta) \
      --target-org scratch-org-alias \
      --result-format human
```

That brings PR validation from 20-25 minutes (full deploy + all tests) down to 3-5 minutes (delta + relevant tests). The speedup is the difference between developers waiting for CI and developers ignoring CI.

The trade-off: delta deployments can mask cross-PR conflicts. If two PRs each work individually but conflict when combined, both individual deltas pass but the staging deploy fails. The full deploy on staging catches this. The delta on PR validation just optimises for speed.

## Quick deploy

Quick Deploy is the feature that makes production deploys fast. It works like this:

1. You validate a deployment against production using `sf project deploy validate` (which runs the deployment as a check-only, including all the tests).
2. The validation succeeds. You now have a deployment ID.
3. To actually deploy, you call `sf project deploy quick --job-id <id>`.

Quick Deploy uses the previously-validated deployment, skipping the test runs. The deployment that took 30 minutes during validation now takes 2 minutes during the actual deploy.

The team that doesn't use Quick Deploy is doing 30-minute production deploys for every release, with the team waiting to see if tests pass. The team that uses Quick Deploy is doing 2-minute deploys and using the saved time for actually validating the deploy worked.

A specific war story: I worked with Anneliese, the delivery manager, on a project where production deploys were a full ceremony — go-no-go meeting, deploy window, everyone watching the build, commit to backing out if tests didn't pass. The deploys took 30 minutes. The team was burned out from the ceremony. Anneliese asked if we could cut the deploy time. We added Quick Deploy. Production deploys went from 30 minutes to 2 minutes overnight. The ceremony evaporated because there was nothing to watch. Anneliese said *"we should have done this two years ago."*

## Destructive changes — the manual review rule

Adding metadata is easy. Removing metadata — deleting a field, renaming a class, removing a Flow — requires a destructive changes manifest. `sfdx-git-delta` generates this automatically.

The risk: an automated destructive deploy can delete a field that has live data in it. Once the field is deleted, the data is gone. Salesforce's recycle bin retention is 15 days. After that, recovery is essentially impossible.

The rule I run: destructive changes always go through a manual review.

The CI pipeline generates the destructive manifest and posts it to the PR as a comment. A human reviews it, approves it, and only then does the deployment proceed. The pipeline does not auto-deploy destructive changes ever, regardless of whether all other checks pass.

Anneliese — same Anneliese as the Quick Deploy story — caught a destructive deploy that would have removed a custom field with live data. She had a habit of running diff reports manually before approving deploys, because she didn't trust automation completely. The field had been marked for deletion in a PR that nobody had connected to the actual data on the field. The PR validation passed because the metadata change was syntactically valid. The destructive review caught it because Anneliese looked.

We changed the process to require human review on destructives that day, formalising what Anneliese had been doing informally. Anneliese got a small bonus and a written commendation for the catch. She's still at that org, still running diff reports, still slightly distrustful of automation.

## Apex test strategy in CI

Test execution in CI has tradeoffs that aren't obvious from the docs.

**Run all local tests on every PR? Or only changed tests?**

Running all local tests catches more regressions but takes longer. Running only changed tests is fast but misses bugs in code you didn't change but depend on.

The pattern I prefer: PR validation runs only changed-and-related tests (using the test class names that share package directories with changed files, plus any test class explicitly mentioned in the PR). Staging runs all local tests. Production validation runs all local tests too.

This gives developers fast PR feedback while still ensuring full test coverage before production. The intermediate "staging" step is where comprehensive tests happen.

**Apex test parallelism.** Salesforce can run Apex tests in parallel by default. Enable parallel test execution in your dev hub settings. Most test suites cut runtime by 50-70% with parallelism. Some suites have race conditions that only appear in parallel runs — those need to be fixed, not avoided.

**Test data factory in CI.** Every CI test run starts with no data. Your tests need to construct what they need. The TestDataFactory pattern from Chapter 5 is essentially mandatory for CI to work; tests that depend on pre-existing data don't run reliably in scratch orgs.

## The release cadence question

How often should you deploy to production?

There are two extreme answers. "Continuously, on every merge" is the modern Web-app default. "Quarterly with a release ceremony" is the traditional enterprise default. Both are wrong for most

Salesforce orgs.

The right cadence depends on:

- **How risky are individual changes?** If most changes are small (fields, validation rules, minor automation), continuous works. If changes are large (data model migrations, sharing model rewrites), batch them.
- **What's the team's testing capacity?** Continuous deployment requires automated tests that catch regressions reliably. If your tests are weak, you're going to ship bugs continuously.
- **What's the regulatory environment?** Some industries (financial services, healthcare) have change-management requirements that make continuous deployment hard. You can still do it; the audit trail just gets more complex.
- **What's the user impact tolerance?** If users actively hate change-related disruptions, batch into less frequent releases.

The cadence I run for most teams: **weekly production deploys, on Tuesday mornings.** The reasoning:

- Tuesday gives the team Monday to catch up on weekend issues, plus enough of the week to fix anything the Tuesday deploy reveals.
- Weekly is frequent enough that individual deploys are small (low risk) but infrequent enough that there's coordination time between releases.
- Tuesday mornings are quiet for most users, so a brief unintended disruption is less visible.

Daniel, who's now a tech lead at a different bank, runs the same cadence on his team. He told me they tried daily and it was too much, monthly was too risky. Tuesday-morning weekly is the right size of bite.

## Closing

Aaron's Friday-afternoon deploy was the moment I decided I needed to think harder about the guardrails. The deploy itself was on me as much as on him — my pipeline didn't have a "Future method calls" warning, my staging environment hadn't replicated the limit-hitting load, my deployment policy didn't enforce the no-Friday-afternoon rule.

We added all of those. The next year had no incidents like Aaron's. The year after that, on a different project, we caught two similar issues during validation that would have been Friday-afternoon-prod-deploy disasters under the old setup.

The Quick Deploy that cut Anneliese's team's release ceremony to 2 minutes. The destructive deploy review she caught manually. The delta deployment that cut PR validation from 25 minutes to 4. None of these are dramatic. Together, they make the difference between a team that dreads deploys and a team that doesn't notice them.

Boring deploys are the goal. Spectacular deploys are usually a sign someone's about to update their LinkedIn.

---

# Chapter 10: Working With Teams

I'm going to write this chapter as a single story, because that's how I want to leave it. There's no code in this chapter. The previous nine have been about tools and patterns; this one is about people, and people don't fit into a section header followed by a code block.

The story is about a team I led for about two years. The names are real first names — I asked the people involved. The events are real, slightly compressed and reordered for readability, but nothing invented. The conclusions I draw at the end are mine; the team might draw different ones if they wrote this. Maybe they will.

---

When I joined the project, the team was four developers. Daniel, who you've met a few times in this book, was the senior. Hannah from Chapter 3 wasn't on the team — different company — but the architecture was similar. There was Liam, a junior who'd been on Salesforce for about eight months, sharp and quiet. There was Sanjay, mid-level, very good at Apex but slightly distrustful of frameworks (he'd been burned by an over-engineered one at his previous job; understandable). There was Priya, who I worked with at length on the data-model consolidation in Chapter 2 — she'd moved to this project specifically because she wanted to do architecture work rather than admin work, and the project lead had agreed to that path.

Four people. Tight team. Architecture was clean because everyone could hold the whole thing in their head. We had standards because Daniel and Sanjay enforced them through code review without it being a formal thing. The trigger framework was in place — the simple version, not the elaborate version, because we didn't need elaborate. Tests were honest. Production was stable.

Then we won a big bid.

The client wanted six months of work, and our team of four wasn't going to deliver six months of work in six months. The PMO came back with the obvious answer: hire more people, scale the team. This is the part of consulting where most architects learn whether their architecture survives the team doubling.

We doubled to eight in about three months. Anneliese, the delivery manager (yes, the same Anneliese from previous chapters — she rotated onto this project after the previous one finished), came in to coordinate. Aaron arrived as a junior developer about two weeks after his graduation. We hired two contractors — Mei (the same Mei who had the 40-Flow archive on a different project; she was a contractor between full-time roles) and a developer named Liam-the-second (we will not be using "Liam-2"; he ended up going by his middle name "James" within the team, which is

confusing because there's already a James in this book, but bear with me). And we added a fellow architect, Tara, who'd been doing financial services work and was looking for a change.

The first month was chaos.

The four-person team had been operating on shared context. Everyone knew the patterns, the trade-offs, the why-we-built-it-this-way for every decision. The new four didn't have any of that. They had access to the codebase, the documentation (which was sparser than I'd realised), and whatever conversations they could grab in passing.

Aaron, the new grad, made the Friday-afternoon Future-method deploy in week two of his employment. We covered that in Chapter 9. The fix was technical — better CI, better deploy policy, better awareness of the Future-method footgun. The actual lesson was that nobody had told Aaron about the Future-method footgun, because everyone on the original team had absorbed it through practice and we'd never written it down. Aaron didn't know there was something to know.

Mei, the contractor, started building Flows the way she'd built Flows on other projects — her own way, without consulting the team's existing patterns. Her Flows worked. They didn't follow the team's naming conventions, didn't go through the centralised utility, didn't fit the existing architecture. Sanjay flagged it in code review. Mei pushed back, politely but firmly: *"I've done this for six years across thirty projects, this is how I build."* She wasn't wrong about her own competence. Sanjay wasn't wrong about wanting consistency. Both of them were right and neither could fully concede.

Tara, the new architect, started questioning some of the decisions the original team had made. *"Why are we using a Lookup here when this should clearly be Master-Detail?"* The answer was a real one — we'd discussed it eighteen months earlier and chosen Lookup because we needed to be able to reparent records — but the answer wasn't documented. Daniel walked Tara through the reasoning. Tara nodded. The next week she asked the same question about a different decision. Daniel walked her through that one too. By the third question, Daniel was visibly tired.

I noticed the pattern after the third question. The original team had been operating on tribal knowledge. The new team was hitting that knowledge from the outside, as obstacles. Every architectural decision the original team had made on solid reasoning was, to the new team, an arbitrary constraint. We had to externalise the reasoning or the new team would slowly grind it down.

So I started writing ADRs. About a dozen, retroactively, capturing the decisions that had been implicit in the codebase. Each one took about an hour. We held a half-day workshop to review them with the whole team — the original four and the new four. The new four said it was the most useful onboarding session they'd had on any project. The original four said it was useful too, because some of the decisions had been informal enough that even they weren't sure what we'd actually decided.

The ADRs solved Tara's questioning problem, partially. She still asked questions, but she now asked them with context — *"I've read ADR-005, and I understand why we chose Lookup here, but the situation in this new feature is slightly different — does the same reasoning apply?"* Those are

good questions. They led to better decisions. The question without context was friction; the question with context was architectural review.

The ADRs didn't solve Mei's pattern-deviation problem. She continued doing her own thing. Sanjay continued flagging it in code review. The friction continued, low-grade, for about six weeks. Eventually Mei and I had a conversation. I said something like: *"Mei, your stuff works. I'm not asking you to use our patterns because mine are better. I'm asking because the team has eight people now, and we need a consistent codebase or maintenance becomes impossible. Will you adopt the patterns for new work, even if you wouldn't have chosen them on your own?"* Mei thought about it for a day. She came back and said yes, on one condition: she wanted to write a section of the team standards document, specifically about the Flow patterns we used. I agreed instantly. She wrote a really good section. The team's standards were better for it. Mei stayed for the rest of the project. We became friends. She still messages me when she's between contracts.

The framework I'd set up didn't fully survive the team doubling, even with the ADRs and the standards. Aaron, after the Future-method incident, became one of the most thorough developers on the team — but his thoroughness was about not making the same mistake again, not about deeply internalising the framework's why. He followed the rules. He didn't extend them. When new patterns emerged that the framework didn't explicitly cover, he'd ask Daniel rather than infer the answer from first principles.

That's a sign of healthy mentorship; it's also a sign that the framework wasn't owned by the broader team yet. The framework was Daniel's and mine. The team used it. They didn't carry it.

I noticed this most clearly when I left the project. About two years in, my contract ended and I moved on. The team kept going. I checked in occasionally, mostly with Anneliese for delivery updates and Daniel for technical news. Things were fine. Then about six months after I left, Daniel mentioned in passing that the team had stopped using one of the framework's patterns — a centralised event-bus thing I'd built — because they'd found it confusing and replaced it with something simpler.

My first reaction was disappointment. I'd built that pattern carefully. It was technically elegant. It covered cases the simpler replacement didn't.

My second reaction, after sitting with it for a week, was that the team had probably made the right call. They had to maintain it. I didn't. The pattern that the maintainer chose was, by definition, more sustainable for them than the pattern they'd inherited.

The third reaction, longer in coming, was that this was the actual measure of the architecture's success — the team kept evolving it after I was gone. If they'd been frozen in the patterns I'd left behind, the architecture would have been a fossil rather than a living system. Replacing one of my patterns was a sign the team had grown into ownership of the codebase. That was the goal.

I ended up coming back to the project as a contractor about three years after I'd left, on an unrelated piece of work. The codebase was unrecognisable in places. The trigger framework had been simplified. Some of the original ADRs had been superseded by newer ADRs; the decision-

making rationale was still there, just updated. Mei's Flow patterns had become the standard across the org. Aaron, now a senior, was writing code I'd have been proud to ship.

I didn't recognise large parts of the codebase. That, I think, was the actual measure of the architecture working. The team had kept evolving without me. They didn't need me. The framework I'd built had been a starting point, not a finished thing, and the team had used it as a starting point for years of their own decisions.

I was on that project for three weeks, doing the unrelated work. On my last day, the team took me to lunch. Daniel made a small speech about the trigger framework. Sanjay did an impression of me explaining ADRs that was mostly accurate. Mei told a story about the time I'd insisted on a specific naming convention that she'd ignored. Aaron — Aaron was now a senior — gave me a printed copy of an ADR he'd written, declaring that the original framework's event-bus pattern was deprecated. He'd signed it. I framed it when I got home.

The best architecture I've left behind is the one I came back to and didn't recognise. It meant the team kept evolving without me. That was the whole point.

---

There's a smaller version of this lesson in every team, every project, every architectural decision. The architect who insists on the original framework being preserved exactly is, almost always, the architect whose work doesn't outlast them. The frameworks that survive are the ones the team makes their own.

The named characters in this book — Daniel, Priya, Mei, Aaron, Liam, Sanjay, Hannah, Ravi, James, Anneliese, Tara, Ben — are composites and combinations from across my career. Most of them are still working in Salesforce, mostly at different companies than where I worked with them. We catch up sometimes. Some of them have written articles or given talks that I quote without attribution, because some of the best architectural thinking I've encountered came from people who were briefly on a team with me and then went off to do their own thing.

The team that doubled is still mostly together. We catch up sometimes. None of them work on Salesforce anymore — Aaron moved to Apex-flavoured backend roles at a non-Salesforce company, Sanjay became a startup CTO, Mei runs her own small consultancy. Daniel is on his second post-Salesforce role doing data engineering. The four originals have stayed in touch with each other longer than they've stayed in the role we worked together in.

That's also a measure of the team being good. The technical work was real. It also wasn't the most important thing about working together.

---

# Conclusion

I started writing this book sometime around the end of 2024, in the gap between client work and family things. It's now early 2026 and the book is almost done, and the things I think about Salesforce architecture have shifted a bit even in the year and change it took to write.

So this conclusion isn't a tidy summary of the book. It's more of a list of what I'd tell a younger architect — or, more honestly, what I'd tell day-1-architect me if I could send a note back through time. Some of it's in the book. Some of it isn't, because I've only realised it recently.

## Things I'd tell day-1 architect me

The most useful thing isn't a technical pattern. It's the discipline of writing decisions down — not to be a stickler for documentation, but because the act of writing the reasoning forces you to know what the reasoning actually is. Half the architecture decisions I made in my first two years didn't survive being written into an ADR; once I had to articulate them, I'd realise the reasoning was thinner than I thought.

The technical thing I'd most insist on: don't ship clever code. Ship boring code. The boring version compounds. The clever version costs forever. I've never regretted choosing the boring pattern. I've regretted choosing the clever pattern dozens of times.

The thing about people I'd insist on: when a developer flags a concern in code review or design review, take it seriously even if it sounds wrong. Ravi was right about the phone numbers (Chapter 6). Daniel was right about my over-elaborate trigger framework (Chapter 4). Mei was right that her Flows worked even if they didn't fit my patterns (Chapter 10). The senior-developer instinct of "I've thought about this more than you, I know better" is wrong about half the time and right about half the time, and the only way to tell the difference is to slow down and listen.

The thing I'd most warn against: don't conflate the org's success with your personal stylistic preferences. Just because I like the fluent builder API doesn't mean the team should have one. Just because I love the trigger framework doesn't mean an org with three triggers needs it. Pattern-first thinking is a trap. Org-first thinking is the actual job.

## Patterns I'd skip entirely

A short list of things I've spent time on in my career that I now think were largely wasted:

**Memorising every governor limit in detail.** They change. Look them up when you need them. The discipline of writing code that won't hit limits matters more than knowing exactly what each limit is.

**Building "frameworks" for things that don't repeat.** A framework for one use case is just a more complicated version of the use case. If you find yourself building a generic abstraction the first time you encounter a pattern, you're building too early. Ship the specific thing. If the pattern recurs, refactor to a framework on the third or fourth occurrence.

**Polishing demos for stakeholders who weren't going to buy in anyway.** I've spent days on slides for people who'd already made up their minds. Read the room earlier.

**Generic admin tools that "save time" but only get used twice.** I've shipped about a dozen of these across my career. The pattern is always the same: the tool gets built, gets used by its author, doesn't get adopted by the broader team because the team doesn't know it exists or doesn't trust it, falls into disuse, gets deprecated when someone notices it. Just do the manual thing twice rather than building the tool that will eventually be deprecated.

**Defending pet patterns past the point they were useful.** The trigger framework was useful in 2015. By 2020, parts of it had been superseded by platform features. By 2024, all of it could probably have been replaced with record-triggered Flows for at least 60% of cases. I kept defending the framework partly out of pride. The team was right to start drifting from it.

## One question I'm still working on

In week one of an engagement, how do you tell whether the client wants real architecture work or just someone to make their existing org behave?

The clients who want the first kind ask architecture-shaped questions: how should we model this, what are the trade-offs of approach A vs B, what would you do differently. The clients who want the second kind ask feature-shaped questions: when can you have this delivered, how much will it cost, who's accountable for the milestone. Both are legitimate. They want different things from a consultant.

I keep getting this wrong on first engagements. I default to architecture work. Some clients want that and we have a great year. Some clients wanted features and I've over-architected something that didn't need it. The signal is hard to read because most clients say they want both, in roughly the same words, regardless of which they actually want.

I've tried various heuristics. None of them work reliably. The only thing that consistently works is asking the question explicitly in week one: *"Are you looking for architectural change or feature delivery? Both are valid. The work I do is different in each case."* Some clients give an honest answer. Some clients say "both" and don't realise they're asking for feature delivery.

I haven't figured out the answer. I'm including this in the conclusion because it would have been dishonest to write a book this long and not admit there's still something I don't have a frame for.

## Where I think Salesforce architecture is heading

A few things, hedged:

**AI is changing the consulting model.** The boring parts of architecture work — writing tests, generating documentation, scaffolding patterns — are getting absorbed into AI workflows fast. The judgment parts aren't. Architects who spend their days on the boring parts will see those parts compressed. Architects whose value is in judgment will see the value clarified. I've covered this in a separate article but the short version is that architecture is becoming more about thinking and less about doing, and the people who'd been hiding behind doing are about to get exposed.

**Multi-cloud is real.** Salesforce Data Cloud, the integration with non-Salesforce systems, the move toward unified customer data platforms — these are the directions the platform is going. Architectures that assume Salesforce is the only system are aging out. The next generation of Salesforce architects will spend more time on data flows between platforms than on the configuration of Salesforce itself.

**The ISV economics are shifting.** AppExchange has been getting more competitive every year. The free packages I helped build in 2018 wouldn't be commercially viable today. The ones that win are increasingly specialised, increasingly opinionated, and increasingly tied to specific verticals. Generic horizontal apps are getting harder to make economic sense of.

**Architecture-as-a-service.** The full-time architect role is, slowly, being replaced by the architect-on-retainer model. Mid-sized orgs can't justify a full-time architect, but they can justify two days a month from someone good. The market for that is growing. The day-rate will probably keep going up. The number of people who can do the job well is growing more slowly than the demand.

I could be wrong about all of this. Architectural prediction is mostly bad. The trends above are what I'm seeing in 2026; in five years some of them will look obvious and some will look silly.

## A note about the kid

I'm writing the last paragraphs of this book in the weeks before our first child is due to arrive. The baby is the reason this book exists, kind of. About a year ago I realised that my work pattern — full-time job plus consulting on the side plus occasional writing — wasn't going to survive the baby. Something was going to give. The choice was either to drop the consulting (income I didn't want to lose) or to drop the writing (which I love but doesn't pay), or to find a way to do everything more efficiently.

The way I found involved AI tooling, a tighter scope on consulting, and a writing engine that could absorb most of the drafting work. The book is part of the experiment. The kid is the deadline. The deadline forced choices I'd been avoiding.

That's not in the book proper because it didn't fit anywhere. It is, in some way, the actual reason the book exists.

## Closing

If you're reading this, you got through 180 pages. Thanks for that. If the book made one decision easier for you, that's enough. The book exists to help architects make decisions like the ones I've struggled with, slightly more easily, with slightly more context.

Pick a chapter. Read it twice. Do the thing it surfaces. Then go home — your team will be fine.