Developer to Architect

Think, Design, and Lead Like a Software Architect

Shameel Ahmed Z

DEVELOPER TO ARCHITECT

FREE SAMPLE

Think, Design, and Lead Like a Software Architect

Shameel Ahmed Z

Preface

In early 2023, I published a five-part article series about transitioning from software developer to IT architect, as part of the Red Hat Enable Architect program. The series was a compressed account of what I had learned making that transition myself: change your mindset, develop soft skills, acquire technical breadth, master the human side of the role, and give back to the community that helped you get there. Each article was designed to be readable in the time a developer had during a lunch break or a commute.

The response surprised me. Engineers wrote to say the series had named something they had been trying to articulate for months. Developers forwarded articles to their managers as evidence that what they were pursuing was a real and mappable thing. A number of people asked where they could find more.

That question is the seed of this book.

An article can point at a problem. A book can go inside it. What the series sketched in a few hundred words each, this book examines in depth: the specific mechanics of how architectural decisions get made and documented, the ways architects fail in their first interactions with senior stakeholders, the technical vocabulary that separates architects who are trusted from those who are not, and the practical realities of the first ninety days in a role where nobody has written down what you are supposed to do.

There is a question that surfaces repeatedly across engineering careers, usually at the point where a developer is good enough at their work to wonder whether there is something larger they should be doing. The question takes different forms in different organizations, but it is always some version of the same thing: what does an architect actually do that I am not already doing?

The answer, which most people are not told directly enough, is that the transition from developer to architect is not a promotion. It is a change in the fundamental unit of work. As a developer, the unit of work is code: a function, a class, a service, a test. As an architect, the unit of work is a

decision, one that shapes how code is written by others, across time, often long after the architect has moved on to something else.

That shift is harder than it sounds. It requires a different relationship with certainty. Developers are rewarded for getting things right, for finding the solution, for closing the ticket. Architects are required to make decisions under genuine uncertainty, to commit to a direction before all the information is available, and to accept that some of those decisions will turn out to be wrong. The coping mechanism for that discomfort is not confidence. It is process: knowing how to make decisions well, how to document them, how to revisit them when the world changes, and how to communicate them to people who need to act on them.

This book addresses that transition directly. It does not describe an idealized architecture practice that functions cleanly in consulting engagements and conference talks. It describes what architecture looks like when the system is in production, the team is under pressure, and the decisions that seemed right six months ago are starting to show their costs. The frameworks are tools to adapt, not templates to follow. The stories are invitations to recognize situations the reader has already encountered, or will encounter soon.

Architecture is a craft. It can be learned, practiced, and improved. The architects who build long careers in this discipline are not the ones who make the fewest mistakes. They are the ones who learn fastest from the mistakes they make, share what they learn with the teams around them, and build systems that are honest about their own limitations.

If you read the original article series, this book is the version without a word limit. If you are coming to this fresh, the articles remain a useful orientation and are freely available on the Red Hat Enable Architect site. Either way, read this like a practitioner: with a specific system in mind, a specific problem in view, and a willingness to test the ideas against your own experience rather than accept them at face value.

Original article series: **redhat.com/en/authors/shameel-ahmed**

Introduction

The transition from developer to architect is not a promotion. It is a change in the fundamental unit of work: from code to decisions that shape how other people's code is written, across time, often long after the architect has moved on. Most developers who make this transition are not told that clearly enough, early enough. This book is an attempt to fix that.

I made that transition deliberately. After close to a decade of building software, having moved from desktop applications to enterprise web development with .NET and ASP.NET, I had come to see architecture as the natural next step. The decisions I found most interesting were not in the code. They were in the system: how components related to each other, how the whole would behave under pressure, and what the team building it would need to be able to change without breaking everything else. Before I moved into architecture full time, I spent two years in a technical manager role, a position that put me in charge of both the technology and the people delivering it. That hybrid experience turned out to be an education I did not know I needed.

What I had underestimated was the shift in the fundamental unit of work. Managing technology alongside managing people had prepared me for accountability and for delivery pressure. It had not fully prepared me for what architecture actually required. I still reached for the code instinctively. I still thought first about implementation. What teams actually needed from me was something different: not to implement, but to tell them what to build, and why, and in what order, and what the tradeoffs were, and to do that in a way that held up when the requirements shifted and the team changed and the original assumptions turned out to be wrong.

I got a lot of that wrong early on. I over-specified things that should have been left to the team. I under-specified things that needed to be pinned down. I communicated designs to developers in the same way I would have explained them to myself, then wondered why the resulting system was not what I had pictured. I agreed to SLA commitments I had not thought through carefully enough and spent months dealing with the

consequences. I treated stakeholders as people to inform rather than people whose understanding was part of my job.

None of those were catastrophic failures. They were the ordinary mistakes of a developer trying to think like an architect without a map. What made the difference, over time, was finding the map: learning that architecture is a discipline with its own vocabulary, its own decision frameworks, its own ways of communicating to different audiences, and its own methods for managing the uncertainty that is always present when you are designing systems for problems that are not yet fully understood. That map is what this book tries to give you.

Who This Book Is For

This book is for senior developers who are asking themselves what an architect actually does that they are not already doing, or who have recently crossed into architecture and are working out what the role genuinely requires.

More specifically, this book is for you if: you are the go-to person on your team for technical decisions but are not sure how to formalize that into an architecture practice; you have recently moved into an architect role and feel the gap between what the title implies and what you are currently equipped to do; or you can see the next level clearly enough to know you want it but cannot yet see how to get there. If you are targeting a Staff Engineer, Principal Engineer, Technical Lead, or Solutions Architect title, or a specialized role in cloud, security, data, or integrations, the thinking in this book applies directly.

How to Use This Book

The book is divided into six parts. Part I covers the mindset shift that the transition requires. Part II covers the technical foundations that architects need beyond their development background. Part III covers the craft of design: decisions, patterns, diagrams, quality attributes, governance, and reference architecture. Parts IV and V cover architecture as it plays out across organizations and teams:

communication, leadership, stakeholder management, and trade-offs under pressure. Part VI covers the career transition itself: closing the skills gap, navigating internal politics, surviving the first ninety days, and planning what comes next.

The chapters build on each other, so reading in sequence is recommended for readers who are new to the role. Experienced architects may prefer to use the table of contents as a diagnostic: find the area where you feel least confident and start there. Each chapter ends with a "What to Take Away" section summarizing the key ideas. Some chapters include structured reflection exercises to apply the ideas to your own work and context.

A Note on the Stories

Every story in this book is drawn from real experience across real projects and teams. Details have been changed, timelines compressed, and organizations described by context rather than by name, to protect confidentiality. Where composite stories draw on more than one situation, I have noted that. The patterns are real. The mistakes are ones I made or watched colleagues make. The lessons are ones I lived through, often more than once.

Companion Resources

The companion site for this book is *devtoarch.com*. It contains downloadable templates for every framework and process described in these pages, including Architecture Decision Record templates, SLO worksheets, threat model frameworks, stakeholder maps, RFC templates, and a 90-day plan starter. It also includes interactive tools, a skills self-assessment, a certifications roadmap, and a community space for readers making the transition. All resources are free and do not require registration. A full index of resources, organized by chapter, is in Appendix I and at devtoarch.com/resources.

Shameel Ahmed Z

June 2026

Contents

1

C H A P T E R

1 What Architects Actually Do

I did not plan to become an architect. I planned to get better at writing software, and I spent the first decade of my career doing exactly that. It started with desktop applications: Visual Basic 5, then VB6, building internal tools and ActiveX controls, sharing component libraries with other development teams, learning what it means to design software that other people depend on. The web changed everything. I moved to .NET and ASP.NET, joined a major US telecom, and worked my way through a content management system, an enterprise case management tool, and eventually the enterprise telephony division: a platform that powered a high-volume reporting system used by hundreds of thousands of agents across the country.

The promotion happened the way most things do in large organizations. You become the person people come to with the hard questions. You start making design decisions, informally, because you understand the system and the team trusts your judgment. You spend a weekend rearchitecting a piece of the platform because nobody else has the context to do it, and the next week your manager schedules a meeting. By the time I was formally given the title of Senior Architect, I had been doing parts of the job for over a year.

I had been writing code professionally for over a decade. I had built integrations, designed databases, led the team through production incidents at 2 a.m. I had opinions about distributed systems. I had survived three major refactors and one data migration that I still do not like to think about. I was, by any reasonable measure, a senior engineer who knew his craft.

And yet, sitting at my desk with a brand-new title, I felt like a fraud. Because I had no idea what I was supposed to do on Monday morning that was different from what I had done last Monday.

1.1 The Mythology of the Ivory Tower

Here is the story most developers have been told, usually by watching the architects they've worked with: the architect is the person who draws boxes and arrows on whiteboards, attends a lot of meetings, hasn't written production code since the previous decade, and periodically descends from on high to declare that the team should be using

microservices, or not using microservices, depending on what conference they attended last quarter.

This archetype exists. I have met people who fit it. They are not what this book is about.

The **ivory tower architect** is a failure mode, not a job description. It's what happens when someone smart is promoted away from the work they understood and never develops the new skills they need to stay relevant. The distance from implementation that comes with the title can, if left unchecked, become permanent. The architect stops understanding what it costs to build what they're specifying. The gap widens. The credibility evaporates. The team starts ignoring the architecture documents.

I have been guilty of elements of this. Not the full picture, but enough of it to recognize the pull. When you're no longer responsible for shipping the code yourself, it becomes easy, almost dangerously easy, to propose elegant solutions that look clean on a diagram and are hell to actually implement.

So let's clear the air: the architects who matter, the ones whose decisions actually improve systems and teams, are deeply connected to the reality of building software. They may not be the ones committing code every day, but they understand what it takes to do so. They keep that understanding current, deliberately and consistently. Their value isn't in being above the work. It's in being able to see the work at multiple altitudes simultaneously.

> ***The architects who matter are deeply connected to the reality of building software. Their value isn't in being above the work. It's in seeing it at multiple altitudes simultaneously.***

1.2 What a Day in the Life Actually Looks Like

If you're a developer, your day has a kind of satisfying structure to it. There is a backlog. There are tickets. You pick something up, you build

it, you mark it done. The feedback loop (write code, run tests, see it work) is tight and rewarding. Progress is visible.

An architect's day looks nothing like that. And if you aren't prepared for the difference, it will drive you mad.

Here is a more honest picture of what my weeks look like, many years in. On any given day, I might spend a morning in a design review for a system I won't be building: reading diagrams, asking questions, pushing back on assumptions, and trying to surface risks the team hasn't seen yet. I might spend the afternoon writing: an architectural decision record, a proposal for how we should handle authentication across a new set of services, or a short document explaining to a non-technical director why the proposed approach to the data warehouse will cost twice as much to operate as they think.

There will be at least one conversation where I am not the most technical person in the room and need to listen more than I speak. There will be at least one conversation where I am the most technical person and need to translate without condescending. If I'm lucky, there's a ninety-minute block where I can actually sit with a codebase and get my hands into something real.

I often end the day uncertain whether I accomplished anything. That uncertainty is the hardest part of the transition for most developers. The feedback loops are long. The impact is indirect. You make a decision today whose consequences won't be visible for six months. You say something in a meeting that shapes how a junior engineer thinks about a problem, and you won't know for years whether you pushed them in the right direction.

This ambiguity is not a bug in the job description. It is the job. The craft of architecture is making good decisions under uncertainty, at a scale where you can't see all the consequences in real time. Learning to be comfortable with that, building confidence in your judgment without requiring immediate validation, is one of the deepest shifts you will make.

1.3 The People You Work With

One of the biggest practical surprises for developers moving into architecture is how many different kinds of people they are suddenly expected to communicate with effectively. As a developer, most of your day-to-day interactions are with people who share your technical vocabulary. As an architect, that changes significantly. You will be the bridge between technical and non-technical worlds, and the quality of that bridge is a significant part of your value. Understanding who is on each side, what they care about, and what they need from you shapes almost every conversation you will have.

1.3.1 The Engineering Side

Developers are your closest working relationship, and the most important one to get right. They are the people who will implement the decisions you make. If they do not understand the architecture, or do not believe in it, the gap between what is designed and what is built will widen steadily over time. Your job with developers is not to hand down decisions but to build shared understanding: explain the reasoning behind choices, invite challenge, and create an environment where engineers feel safe raising concerns about a design before they are committed to implementing it. The architects who developers trust are the ones who listen as much as they direct.

Technical Leads (also called Tech Leads or Lead Engineers, depending on the organization) are senior developers who carry responsibility for the implementation quality of a specific team or service area. They sit at the intersection of architecture and delivery. In practice, the technical lead is often the person who translates architectural decisions into implementation detail, who raises practical constraints the architect may not have considered, and who owns the code quality of what actually gets built. A good working relationship with your technical leads is one of the most valuable things you can develop. They will tell you what is actually hard to build, which is information you cannot afford to be wrong about.

1.3.2 The Product Side

Product Managers own the "what" and the "why" of what gets built. They are accountable for the product roadmap, for prioritization decisions, and for the business outcomes the product is meant to deliver. The architect's relationship with the product manager is fundamentally about translation: you need to understand the business problem clearly enough to make sound architectural choices, and they need to understand the technical constraints clearly enough to make sound prioritization choices. The most common failure in this relationship is both parties operating in silos and discovering too late that the roadmap assumes something the architecture cannot support.

Product Owners (a role that comes from the Scrum framework) represent stakeholder interests at the team level, own the product backlog, and make day-to-day decisions about what goes into each sprint. In organizations that use Scrum, the Product Owner is often your primary interface for understanding what the team is expected to deliver in the near term, and for surfacing architectural concerns that will affect upcoming work. In practice, the distinction between Product Manager and Product Owner varies a great deal by organization. Some companies use the titles interchangeably; others treat them as distinct roles at different levels of the hierarchy.

Business Analysts sit between business stakeholders and delivery teams, translating business requirements into specifications that developers can act on. In many organizations, the BA produces the detailed requirements that teams build from. For architects, the business analyst is a valuable source of domain knowledge and a useful channel for surfacing constraints early in the requirements process, before they become expensive surprises in design.

1.3.3 The Project and Delivery Side

Project Managers are accountable for delivery: scope, schedule, budget, and risk. Their primary concern is whether the work will be done on time and within cost. The architect's relationship with a project manager is often one of honest constraint-setting: your job is to give the PM accurate information about what is technically possible in the time and budget available, and to flag risks early enough that they can be managed rather than absorbed. Project managers who trust their

architects give them significant influence over how scope is shaped and sequenced. Architects who habitually underestimate complexity or avoid surfacing bad news lose that trust quickly.

Scrum Masters and **Agile Coaches** support the team's delivery process rather than its technical direction. They focus on removing impediments, facilitating ceremonies, and improving the team's ways of working. Architects interact with Scrum Masters primarily around process design: how architecture input should flow into sprint planning, how design reviews fit into the delivery cycle, and how technical debt is surfaced and prioritized. A Scrum Master who understands that architectural work does not always fit neatly into sprint boundaries, and an architect who understands that the team's delivery rhythm matters, makes for a productive working relationship.

1.3.4 The Leadership Side

Directors and **Vice Presidents** are the organizational layer above the day-to-day delivery teams. They own headcount, budgets, and the strategic direction of a product area or engineering function. For architects, directors and VPs are a critical audience for proposals that require significant investment, that cross team boundaries, or that involve strategic technology decisions with long-term implications. The key to communicating with this audience is knowing what they actually need to decide: they are not looking for technical depth, they are looking for risk, cost, and impact framed in terms of business outcomes. An architect who can speak that language credibly has access to decisions that others do not.

CTOs and **VP Engineering** roles vary significantly by organization. In some companies, the CTO is a deeply technical role; in others it is primarily outward-facing and strategic. What is consistent is that the CTO sets the tone for how engineering is valued and how technical decisions are made at the organizational level. Architects who have a productive relationship with the CTO benefit from having their technical direction aligned with the organization's broader strategy. If the CTO cares about platform consolidation and you are proposing further fragmentation, you will spend your energy on the wrong battle. Understanding where the organization's technical leadership is headed, and designing with that direction rather than against it, is basic political literacy for an architect.

1.3.5 The Operational Side

Site Reliability Engineers (SREs) and **DevOps engineers** own the systems that run the software after it has been built. They care about things that are invisible during development: how the system behaves under load, how it recovers from failure, how it is monitored, how it is deployed, and how long an incident takes to resolve. Architects who design without considering the operational reality create systems that are elegant on paper and painful to run. Building a genuine working relationship with your SREs and operations team, understanding their pain points and designing to address them, is the mark of an architect who cares about the full lifecycle of the software, not just the design phase.

Security teams and **Security Architects** are increasingly central to the design process in most organizations. Where security was historically a gate at the end of delivery, it now needs to be a design input from the start. The architect's relationship with security is one of mutual constraint: security teams surface threats and requirements that shape architectural decisions, and architects surface design choices that security teams need to review. The most productive version of this relationship is collaborative and early; the worst version is adversarial and late, when security reviews become blockers rather than inputs.

1.3.6 A Note on Other Architects

If your organization has multiple architects, how you relate to each other matters more than most people expect. Architects who operate as independent authorities on their respective domains, without coordination, produce systems that are locally coherent and globally fragmented. The patterns that govern one service become inconsistent with the patterns that govern another. The data contracts that one team designed without consultation become a source of pain for every downstream consumer. Regular coordination between architects, through Architecture Review Boards, Architecture Guilds, or simply recurring working sessions, is not bureaucracy. It is how you prevent the organization's architecture from becoming a collection of incompatible local optimizations. Chapter 19 covers governance structures for exactly this problem.

The full landscape of stakeholder relationships, including how to read their motivations, navigate conflicting interests, and build the kind of organizational trust that gives architects real influence, is covered in depth in Chapter 23. This section is intended as an orientation map. The territory is considerably richer.

1.4 The Spectrum of Architectural Roles

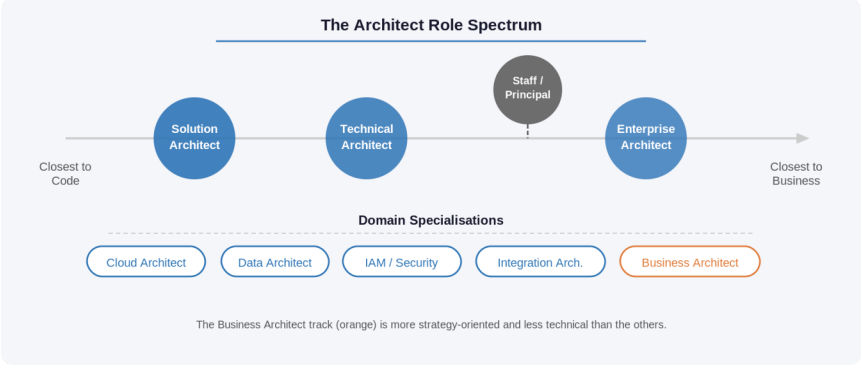


Figure 1.1: The Architect Role Spectrum

Before we go further, I want to name something that creates endless confusion in the industry: "architect" is not a single job. It's a family of related roles with meaningfully different scopes, responsibilities, and failure modes. Understanding where you are (or where you're aiming) on this spectrum matters a great deal for everything else in this book.

1.4.1 Solution Architect

A solution architect designs solutions to specific business problems. The role sits between the business and the engineering team: translating requirements into architecturally sound designs, choosing the right tools, platforms, and patterns for a given initiative, and guiding delivery teams through implementation without being in the code itself. A solution architect is seldom hands-on with the codebase. The accountability is for the shape and soundness of the solution, not for writing it. If you are a senior developer considering the transition, the solution architect path will feel like a significant step away from the

keyboard and toward whiteboards, stakeholder conversations, and design reviews.

1.4.2 Technical Architect

A technical architect is focused on the engineering standards, practices, and patterns within an organization or a large product. Where a solution architect asks "how should we build this thing," a technical architect asks "how should we build things, generally." This role is heavily invested in things like API design standards, testing philosophy, platform choices, and the practices that make a codebase maintainable at scale. Technical architects often have deep expertise in a specific domain (data, security, integration) and serve as the authority in that area across multiple teams.

1.4.3 Enterprise Architect

Enterprise architects operate at the furthest remove from the code. Their concern is the portfolio of systems across an organization: how they relate to each other, how they align with business strategy, where there is duplication or fragmentation, and how the technology landscape should evolve over time. This role requires as much business acumen as technical depth, and the audience for your work is as likely to be the CFO as the engineering team. It is also, frankly, the role most vulnerable to the ivory tower trap, where the abstraction can become so great that contact with engineering reality is lost entirely.

1.4.4 Staff / Principal Engineer

Worth naming here because many of you will encounter this question: what is the difference between a Staff Engineer and an Architect? The honest answer is that it varies enormously by organization. In many companies, Staff and Principal Engineers do architectural work without carrying the architect title; they own technical direction for a domain, make significant design decisions, and mentor other engineers in architectural thinking. In others, Staff Engineers are extremely senior individual contributors who are expected to stay closer to the implementation than a titled architect would. If your company has this track, understand which flavor it is before you decide which path to pursue.

1.4.5 Domain-Specific Architect Roles

Beyond these four tracks, there is a wide range of domain-specific architect roles. These are real career paths, and if one of them calls to you, the foundation this book builds is exactly what you need. The domain expertise you can grow into; the architectural mindset, the communication skills, and the career transition judgment are what we are here to develop.

Cloud Architect. Focuses on the design, migration, and operation of systems running on cloud platforms. The role requires deep familiarity with the service models and pricing structures of one or more major providers, as well as the architectural patterns specific to cloud-native workloads: managed services, serverless, multi-region resilience, and cost optimization. Cloud architects sit at the intersection of infrastructure, security, and application architecture, and are in strong demand as organizations move workloads off premises.

Data Architect. Owns the design of the data layer: how data is stored, structured, moved, and made available for analysis and product use. This includes data warehouse design, data lake architecture, streaming data pipelines, and the governance structures that ensure data quality and regulatory compliance. As organizations become more data-driven, the data architect's decisions have direct business impact, and the role requires close collaboration with data engineering, analytics, and product teams.

Security Architect / IAM Architect. Designs the security controls embedded in systems rather than applied to them after the fact. Where a security engineer implements and operates tooling, the security architect makes upstream decisions about trust models, authentication and authorization patterns, data classification, encryption strategies, and threat models. An IAM (Identity and Access Management) architect specializes further, focusing on the design of identity systems and authorization models across enterprise environments. This role is in strong demand as regulatory requirements and threat sophistication both increase.

Integration Architect. Specializes in the design of connections between systems: APIs, messaging infrastructure, data pipelines, and the contracts that govern how disparate applications exchange

information. In large enterprise environments, the integration architect is often the person who understands how the system of systems is wired together when no single team has that view. The role requires deep knowledge of messaging patterns, API design, and data transformation, combined with the organizational skills to negotiate contracts between teams that own different systems.

Database Architect. Owns the design and governance of database systems: schema design, indexing strategies, replication models, and the selection of database technologies appropriate to each workload. As the variety of database options has expanded (relational, document, graph, time-series, vector), the database architect's decisions have become more consequential. The role often overlaps with the data architect but focuses more narrowly on the operational characteristics of database systems rather than the broader data lifecycle.

Business Architect. Unlike the other roles in this list, the Business Architect sits closer to strategy than to technology. The work involves modeling business capabilities, aligning organizational structure with operating models, and translating business intent into the shape of systems and processes. It draws less on deep technical expertise and more on an understanding of how organizations work and how strategy becomes execution. If that direction interests you, pay particular attention to Parts IV and V of this book, which cover organizational architecture practice, communication, stakeholder dynamics, and organizational influence.

Throughout this book, I'll be speaking primarily to the experience of the Solution and Technical Architect: the roles that are most common, most accessible as first transitions for developers, and most tightly coupled to the act of building real systems. But the principles travel. Wherever you end up on the spectrum, you'll need what we cover here.

Developer	Architect
Pick up a ticket	Design reviews and RFCs
Write and run code	Write proposals and ADRs
See tests pass (or fail)	Translate for non-tech stakeholders

Developer	Architect
Open a pull request	Mentor engineers on design
Review others' PRs	Read code, stay technical
Mark the ticket done	End day uncertain of impact
Tight feedback loops. Progress is visible.	*Long feedback loops. Impact is indirect.*

Figure 1.2: How an architect's day differs from a developer's

1.5 Why Great Architects Still Write Code

I want to make a claim that some architects would dispute: the best architects I have worked with over the years have all maintained genuine engagement with code. Not necessarily writing production features, since their time is too expensive for that in most organizations, but deeply enough that they understand what is hard, what is fast, what is fragile, and what the current developer experience actually feels like.

There are several reasons this matters.

The first is credibility. Teams can tell when an architect understands the work and when they don't. They can tell within the first five minutes of a design review. When you propose a solution that is naive about the operational complexity of what you're suggesting, or when you use terminology in a way that reveals you haven't worked with a technology since version two and you're now on version nine, the credibility you need to have your ideas taken seriously erodes faster than you'd like to think.

The second is decision quality. There is a kind of knowledge you get from building things that you cannot get from reading about building things. You understand, in your hands, why certain things are harder than they look. You understand the difference between a clean abstraction and a leaky one. You understand why the thing that seemed simple in the architecture meeting turned into three weeks of unexpected work.

Architects who don't periodically get their hands dirty lose access to this knowledge, and their decisions degrade accordingly.

The third is trust. When you sit down with an engineering team and you can look at their code and understand what they're doing, when you can contribute to a design not just at the level of components but at the level of how those components get implemented, the relationship is different. The advice is different. The trust is different.

Now, I want to be equally honest about the other side of this. Knowing when not to write code is just as important as staying technically engaged. One of the most common failure modes for newly promoted architects is staying in the code because it's comfortable, because the feedback loops are rewarding, because the hard new parts of the job (the meetings, the ambiguity, the writing) are unfamiliar and uncomfortable. I have watched talented people get promoted into architect roles and spend the next six months doing what they did before, just with a fancier title. They were the best developer on the team. They were not doing the architect's job.

The transition requires you to accept a kind of deliberate incompetence. You are moving from a domain where you are highly skilled to one where you are a beginner, and there is no shortcut through that. Writing code can be a way of avoiding the discomfort of that transition. Be honest with yourself about which one you're doing.

> ***The transition requires you to accept a kind of deliberate incompetence. You are moving from a domain where you are highly skilled to one where you are a beginner, and there is no shortcut through that.***

1.5.1 A Practical Rule for When to Code

The balanced approach that works in practice is to be hands-on for problems that are architectural in nature and hands-off for everything else. Architectural problems where your involvement makes the most difference include: establishing the core framework or platform choices the team will build on, diagnosing structural issues that development is blocked on, validating that an approach is actually buildable before

asking others to build it, and identifying security vulnerabilities that require deep code-level understanding.

Routine feature development, incremental bug fixes, and day-to-day implementation work are better left to the team. Not because those tasks are beneath the role, but because every hour you spend on them is an hour not spent on the architectural decisions that only you are positioned to make. The cost of an architect spending a day writing a feature is not just that day's salary. It is the accumulated cost of the design questions that went unanswered while you were in the code.

The tell that you have the balance wrong is usually a feeling of discomfort in meetings: the design reviews where you have not prepared, the stakeholder conversations where you do not know what to say, the strategy discussions where you feel like a passenger rather than a contributor. That discomfort is not a sign that the meetings are bad. It is a sign that you are avoiding them by staying in the code. The meetings are where the architect's job actually happens.

1.6 A Story: The First Week

Let me go back to that first week with the new title.

I spent most of it doing what I had always done: writing code. There was a new reporting module to finish, a PR to review, some tests to fix. My manager, a patient and perceptive person, let me do this for a few days before he scheduled a coffee.

"How's it going?" he asked.

"Good," I said. "Finished the aggregation module yesterday. Tests are green. I think we can ship Friday."

He nodded. Then he said: "What happens to the report data when the backend service and the aggregation job write to the same tables at the same time?"

I opened my mouth. I closed it. I had not thought about this. The module worked, I had tested it thoroughly. I had not thought about what

happened when two different processes touched the same rows simultaneously under production load.

"That's the architect question," he said. "Not 'does it work in isolation.' 'What happens when the whole system does what we want it to do.'"

It was a small moment. But it was the first time I felt the shape of what I had walked into. I had been so focused on building the module correctly that I hadn't thought about the system around it, the other processes, the concurrency, the interactions I wasn't responsible for. I had been thinking like a developer: solve the problem in front of me. The job was starting to require me to think like an architect: think about the problem at the next scale up, the next failure mode, the next year.

That gap, between solving the problem in front of you and designing for the world the problem lives in, is what this book is about closing.

1.7 What to Take Away

Architecture is not a level above development. It is a different orientation to the same work. The best architects stay deeply technical; the difference is in the altitude at which they apply that technical knowledge.

The title of architect does not confer the skills of an architect. Those skills are built through deliberate practice, uncomfortable new experiences, and a willingness to sit with ambiguity and slow feedback loops.

And the most important thing I can tell you going in: the transformation you're undertaking is real, and it is hard, and it is worth it. The systems you will design will outlast the code you would have written. The judgment you will develop will compound across your career in ways that individual skills cannot. The impact you will have on the engineers around you will be real, even when you can't measure it.

You've spent years learning to build things well. This is the part where you learn to decide what to build.

Let's get into it.

2

C H A P T E R

2 From Solving Problems to Defining Them

A client representative from a large bank once joined a planning call and said: "We need a bulk export feature. Our collections team is losing time every month pulling account data manually. Can you build it by next quarter?"

The developers in the room were already moving. I could see it: tabs opening, file format decisions being made, S3 bucket configurations forming. Someone asked whether we should use CSV or fixed-width. Someone else mentioned they had built a streaming export library before and it was solid.

I asked a different question. "What does the collections team do with the data after they export it?"

The client representative said: "They send it to their predictive dialer."

"And when you say they're losing time," I said, "is the time in the export itself, or in what happens to the data after?"

Silence. The representative did not know. She had the symptom (manual work) and a hypothesis (missing export feature) but not the actual workflow. We had been sixty seconds away from building a bulk export pipeline for a problem we had not actually diagnosed.

We spent the next hour with two collections managers instead. What we found was simpler and more interesting: the export already existed. The problem was that the exported data was missing three fields their dialer required, so the collections team was manually enriching each file in a spreadsheet before uploading it. The solution was not a new export system. It was adding three fields to the existing one and automating the dialer upload.

We shipped a fix in two weeks instead of a platform in three months.

That is the difference between solving a problem and defining it.

2.1 The Developer's Instinct

Developers are trained to solve. It is the core loop of the job: a problem comes in, you understand it well enough to implement a solution, you

ship it, you move on. The faster and more reliably you can do that, the better the developer you are considered to be.

This instinct is good. It is what makes developers productive. A team of people who spent all their time questioning whether they were solving the right problem and never actually built anything would not last long. The bias toward action, toward implementation, toward getting something working and in front of users is enormously valuable.

But it has a shadow side. When you are trained to solve, you start to see every incoming request as a problem to be solved rather than a situation to be understood. The notification system request sounded like a problem. It had the shape of a problem. A developer's brain pattern-matches quickly: notification system, got it, I know what that is, here are the components. The cognitive work of building the solution begins before the cognitive work of understanding the need is finished.

This is not a character flaw. It is a trained reflex. And as an architect, it is one of the first reflexes you need to retrain.

2.2 The Architect's Instinct

The architect's instinct, at its core, is to slow down the moment between receiving a request and responding to it. Not indefinitely. Not paralytically. Just long enough to ask: what is actually being asked for here, and is that what is actually needed?

This sounds simple. It is not. It requires you to push back on the social pressure of a room full of people who are ready to build. It requires you to ask questions that might make you seem like you are not being helpful. It requires comfort with a kind of productive ambiguity that most developers have been trained out of.

The questions that matter most at this stage are not technical. They are diagnostic.

What outcome are we actually trying to achieve? Not what feature are we building: what business or user outcome does this feature serve? If you cannot state the outcome clearly, you do not understand the problem yet.

What evidence do we have that this solution addresses that outcome? The notification system request came with a churn number. That number was real. But the connection between churn and notifications was a hypothesis, not a fact. Architects need to be able to distinguish between data and inference, and to push on the inference before committing to a direction.

What does success look like? If we build this, how will we know it worked? A surprising number of features get shipped without anyone having agreed on what "working" means. That is a failure of problem definition, not implementation.

What happens if we do nothing? This is an underused question. Sometimes the right answer to a problem is not to build something new but to stop building something old, or to accept the current situation and invest the capacity elsewhere. The cost of the solution should always be weighed against the cost of the problem.

> ***The questions that matter most at the start of any architectural engagement are not technical. They are diagnostic.***

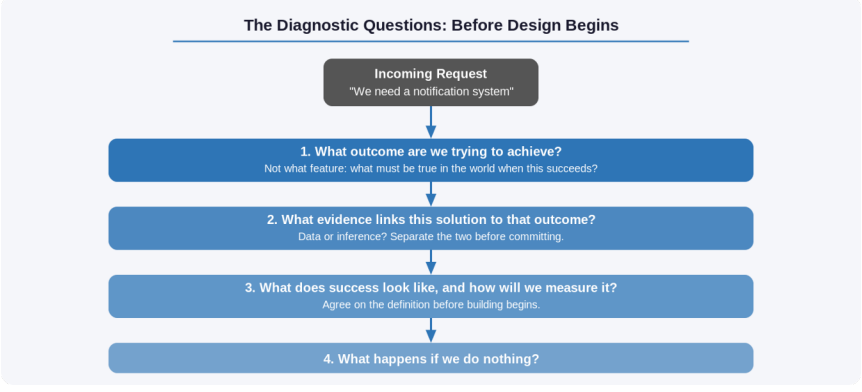


Figure 2.1: Four diagnostic questions before any design begins

2.3 Thinking in Constraints

Once you have a clearer picture of what problem you are actually solving, the next step is to understand the space you are solving it in. Every architectural problem exists inside a set of constraints, and those constraints shape what good solutions look like more than almost anything else.

There are five categories of constraint that architects need to map before making significant design decisions.

The first is time. How long do we have? Is there a fixed deadline, and if so, what is driving it? A hard regulatory deadline is different from a target date someone put on a roadmap six months ago and has since become gospel. Time constraints determine whether you can afford a complete solution or need to design for staged delivery.

The second is money. What is the budget, both to build and to operate? Architectural decisions have ongoing cost implications that are easy to underestimate at design time. A solution that costs twice as much to run as expected is not a good solution, even if it works technically. Always think about cost as a dimension of design, not an afterthought.

The third is team. Who is going to build this, and what can they actually do? A technically elegant solution that requires skills your team does not have is not a practical solution. Architects who design for an idealized team rather than the real one create delivery risk and resentment. Know your people.

The fourth is operations. Who is going to run this once it is built, and what do they need from you to do it well? A system that is hard to monitor, hard to debug, and hard to recover from when things go wrong creates ongoing pain regardless of how clean the architecture is on paper. Operational reality is an architectural constraint.

The fifth is risk. What can we not afford to get wrong? Every system has parts where failure is acceptable and parts where it is not. Identifying those early shapes where you invest in reliability, redundancy, and careful design versus where you can move fast and patch later.

These five constraints rarely pull in the same direction. Time pressure pushes toward simplicity; scale ambitions push toward more complexity. Budget limits push toward managed services; risk concerns push toward control. The architect's job is not to make the constraints disappear. It is to make the trade-offs visible and help the team make an informed choice about which ones to accept.

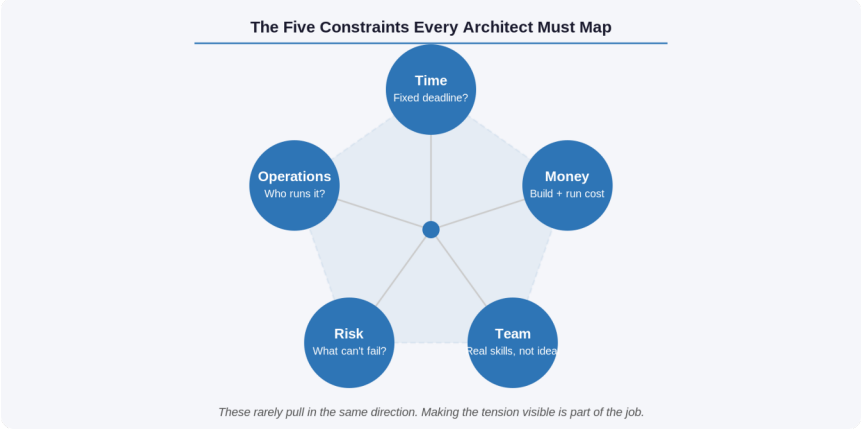


Figure 2.2: The five constraints every architect must map

2.4 The Most Useful Thing You Can Learn to Say

There is a phrase that new architects resist and experienced architects rely on. That phrase is: "It depends."

New architects resist it because it sounds evasive. When someone asks you whether they should use a relational database or a document store, "it depends" feels like a non-answer. The person asking wants a recommendation. They want you to know the answer.

But "it depends" is not evasion. It is precision. The right database depends on the read/write pattern, the consistency requirements, the team's familiarity, the operational complexity they can absorb, the size of the data, and a dozen other factors. A confident recommendation made without knowing those factors is not expertise. It is guessing with authority.

The key is what comes after "it depends." An architect who says "it depends" and stops there has said nothing useful. An architect who says "it depends, and here is what it depends on, and here is how those factors point toward this decision in your specific situation" has done exactly the right job.

Learning to say "it depends" properly is a form of intellectual honesty that takes practice, because it requires you to resist the pressure to perform certainty. Stakeholders often prefer a confident wrong answer to a careful right one, at least in the short term. Architects who can hold their ground on this, who can say "I will give you a clear recommendation once I understand these three things," build more trust over time than those who give quick answers that later have to be walked back.

This is also connected to something deeper about how architects think about problems. Good architects are comfortable with incomplete information. They know how to make decisions when they do not have all the facts, and they know when to wait for more information before deciding. That calibration is a skill. It does not come from intelligence alone. It comes from having been wrong confidently enough times that you learn to check your confidence before you express it.

2.5 The Difference Between a Requirement and a Solution

One of the most common traps in problem definition is receiving a solution disguised as a requirement. This happens constantly, and it is not usually anyone's fault. Non-technical stakeholders think in terms of what they want built, not what they want achieved. Developers think in terms of features. Even technically sophisticated people fall into this pattern because it is cognitively easier to specify a solution than to articulate an underlying need.

"We need a notification system" is a solution. The underlying requirement is something like: "Users need to take action on time-sensitive events, and currently too many of them are missing the window

to do so." Those are different problems, and they can have very different solutions.

"We need a microservices architecture" is almost always a solution in search of a problem. The underlying requirement might be: "We need to be able to deploy changes to our payment logic without coordinating with six other teams." That is a real requirement. Microservices are one possible response to it. They are not the only one, and for many teams they are not the right one.

Architects need to develop the habit of translating incoming requests back to their underlying requirement before evaluating solutions. The translation is not always dramatic. Sometimes the solution someone has proposed is actually the right one, and the underlying requirement confirms it. But the act of making that translation explicit ensures that the choice of solution is deliberate and defensible, not just the first thing that came to mind.

A useful test: can you state why this solution serves the requirement better than at least two alternatives? If you cannot, you have not finished defining the problem.

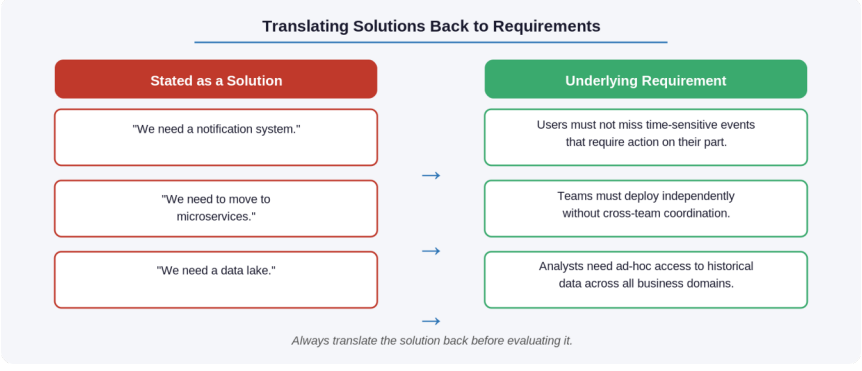


Figure 2.3: Translating solutions back to underlying requirements

2.6 Design Thinking: A Structured Approach to Problem Definition

The habits described in this chapter (slowing down, asking diagnostic questions, distinguishing requirements from solutions) are also the foundation of a formal discipline called **design thinking**. It is worth knowing by name, because you will encounter it in product organizations, consulting engagements, and innovation programs, and because it offers a structured vocabulary for the problem-definition work architects already need to do.

Design thinking was developed at IDEO and formalized at the Stanford d.school. Its central premise is that the best solutions emerge from a deep understanding of the people who have the problem, not from the expertise of the people proposing solutions. The process is organized into five phases: **Empathize**, **Define**, **Ideate**, **Prototype**, and **Test**. They are described as sequential, but in practice they are iterative: testing frequently reveals gaps in problem definition that send you back to Empathize, and prototyping often exposes constraints that reshape the Define phase.

2.6.1 The Five Phases and What Architects Do In Them

Empathize is about understanding the people who have the problem, not through intuition or assumption, but through direct observation and conversation. In a design thinking workshop, this means user interviews, ethnographic observation, and journey mapping. For an architect, it means resisting the impulse to propose a solution during stakeholder conversations and instead spending the first interaction simply listening: what does the user actually do today, what slows them down, what workarounds have they built, and what does failure look like from their perspective? The bulk export story at the start of this chapter succeeded because of an empathy insight: once we understood what the collections team actually did with the exported data, the problem became obvious.

Define is where the observations from Empathize are synthesized into a clear, actionable problem statement. The most useful tool in this phase

is the "How Might We" (HMW) reframe. Instead of accepting the solution-as-requirement ("we need a bulk export feature"), you restate the underlying need as an open question: "How might we get the enriched data to the dialer without manual spreadsheet work?" That reframe opens the solution space rather than closing it. A good problem statement specifies whose problem it is, what they are trying to achieve, and what is currently standing in the way, without proposing any solution.

Ideate is structured divergence. The discipline is to generate a wide range of possible solutions before evaluating any of them. This is harder than it sounds, because experienced architects and developers have strong pattern-matching instincts that jump to familiar solutions quickly. Ideation asks you to hold that instinct back: generate ten options before filtering to one. Bad ideas belong in the ideation phase. They often trigger the good ones. The evaluation and narrowing comes afterward, once the full landscape has been explored.

Prototype is about making ideas tangible cheaply enough to test them. For UX designers this typically means mockups and paper interfaces. For architects, prototyping means something different: an architecture sketch that makes the structure of a proposed solution visible, a spike that proves a technical assumption before committing to an approach, or a simple working model of a data flow. The purpose is the same as in UX: spend thirty minutes making an idea concrete rather than three months building something that turns out to be wrong. Architecture prototypes should be cheap enough to throw away.

Test closes the loop by validating your prototype, and more importantly, by validating your problem statement. A test that shows the prototype does not solve the user's problem is only a failure if you expected the prototype to be correct. Treated as a learning mechanism, a failed test is valuable: it tells you which assumption was wrong and sends you back to Define or Empathize with better information. Architects often skip this phase because they conflate it with quality assurance or load testing. It is neither. It is the question: "Does this approach address what we said the problem was?"

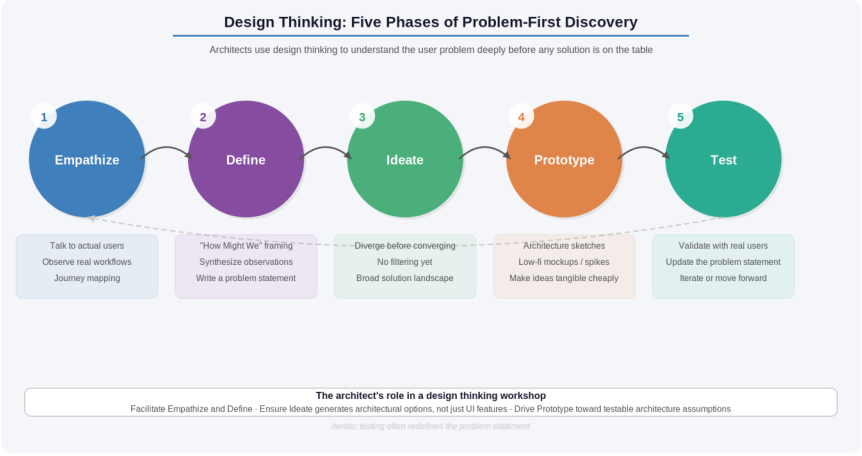


Figure 2.4: The five phases of design thinking and what architects do in each. The cycle is iterative, not linear

2.6.2 When to Use It and When to Skip It

Design thinking is most valuable when the problem is genuinely complex and user-centered: greenfield products where the user need is unclear, major re-architectures where the current system has embedded a misunderstanding of the domain, cross-functional initiatives where multiple stakeholders have conflicting mental models of the problem, or any context where you are not confident the stated requirement reflects the actual need.

It is overkill for well-understood infrastructure work, internal tooling with a small, accessible user base, or any situation where the problem is already clearly defined and validated. Not every architectural engagement is a design problem. When the requirement is specific, credible, and well-evidenced, skipping directly to design is the right call.

The architect's role in a design thinking engagement is frequently facilitation rather than execution. You are unlikely to be running user interviews yourself, but you should know how to structure the problem statement session that follows them, how to run an ideation exercise that generates architecturally meaningful options rather than purely UI-level ideas, and how to design a prototype that tests an architectural assumption rather than just a visual layout. These facilitation skills make you substantially more effective in cross-functional product

conversations, and they signal to product and business stakeholders that you understand their domain of thinking as well as your own.

2.7 Exercise: Reframe a Problem You Have Already Solved

This exercise is most powerful when you apply it to work you have already done, because hindsight removes the pressure of an active decision and lets you see the problem definition process more clearly.

Pick a significant piece of work from the past six to twelve months. A feature you built, a system you designed, a refactor you led. Something substantial enough that there were real choices involved.

Now answer these questions about it in writing. Do not rush. The value is in the friction.

First: what was the stated request? Write it down as it was given to you, as closely as you can remember.

Second: what was the underlying outcome the request was serving? Not what was asked for but what the person asking actually needed to be true in the world.

Third: what were the real constraints? Time, money, team, operations, risk. Which ones did you know about at the start? Which ones did you discover partway through? Which ones were you not aware of until they caused problems?

Fourth: what solution did you implement? And what were the alternatives you considered or dismissed?

Fifth, and most importantly: if you had spent one more hour understanding the problem before you started, what would you have learned? And would that have changed what you built?

Most architects, when they do this exercise honestly, find at least one place where a sharper problem definition would have changed the solution. Sometimes the change is minor. Sometimes it is significant. The point is not to feel bad about past decisions. It is to build the habit

of asking the defining questions earlier, so that future decisions are made on better foundations.

Do this exercise with a colleague if you can. Explaining your answers out loud almost always reveals assumptions you did not know you were making. A printable worksheet for this exercise is at devtoarch.com/resources/problem-reframing.

2.8 A Story: The Requirement That Was Not a Requirement

Partway through a platform consolidation engagement, the project sponsor informed me that the new system needed to support a minimum of one thousand concurrent users. The number had come from the technical lead on the client side, and everyone had accepted it without scrutiny. It went into the requirements document. It shaped the infrastructure design. It became a load-testing target.

Three weeks before go-live, I asked the sponsor a simple question: where did the thousand users figure come from? She checked with the technical lead, who had derived it from peak usage figures for the old system, multiplied by a growth factor of four. The growth factor was aspirational, not forecast. The actual peak on the old system had been two hundred and twelve concurrent users. The projected peak for the next twelve months, based on the business plan, was three hundred and forty.

We had designed and load-tested for a capacity target nearly three times what the system would realistically need. The infrastructure cost was significant. More importantly, the target had driven architectural decisions about the caching layer and the database connection pool that we now had to re-examine.

The number had never been a requirement. It had been an assumption that nobody questioned because it arrived wearing the costume of a requirement. The architect who asks "where did this constraint come from?" before accepting it is doing problem definition work. It is less comfortable than accepting the brief. It is considerably cheaper than building to the wrong specification.

2.9 Commitments and Their Consequences

There is a distinction between developer commitments and architect commitments that most people making the transition do not understand until it costs them something.

As a developer, when you commit to a deadline and miss it, the consequence is usually internal. A conversation with your manager, a sprint retrospective item, a revised estimate. The breach is noted and the work continues. Developer-to-developer SLAs, the internal deadlines for delivering a feature or a service to another team, carry a similar weight. Missing them is uncomfortable. It is rarely catastrophic.

As an architect, you will be involved in setting commitments that are external: service level agreements with clients, availability guarantees embedded in contracts, performance commitments that have financial penalties for breach. These are not internal targets. They are business obligations. Missing them can trigger penalty clauses, damage client relationships, and create liability for the organization. The shift from internal to external commitments is a shift in the weight of the word "commitment" itself.

Understanding this changes how you approach the design of systems. An internal target of 99% availability is an aspiration. An external SLA of 99% availability is a number your architecture needs to reliably deliver, with monitoring to prove it and incident response to protect it when it is at risk. The architect who treats these equivalently will, at some point, discover the difference the hard way.

The practical implication: when you are asked to sign off on or contribute to an external commitment, treat that moment as a design review. Ask whether the architecture supports the number. Ask what the failure modes are. Ask what the incident response plan is if the commitment is at risk. These are architect questions. They belong in the conversation before the commitment is made, not after it is missed.

2.10 What to Take Away

The most expensive architectural mistakes are not made in the design phase. They are made in the five minutes before the design phase begins, when a poorly understood problem gets locked in as a constraint.

Your job as an architect is not just to design good solutions. It is to make sure you and your team are solving the right problem in the first place. That requires slowing down the moment between receiving a request and responding to it. It requires asking diagnostic questions before technical ones. It requires translating solutions back to requirements, and requirements back to outcomes.

None of this means you should be slow. It means you should be deliberate. The time you spend sharpening a problem definition before designing a solution almost always comes back to you in reduced rework, fewer surprises, and systems that actually do what they were meant to do.

The developer in you wants to start building. The architect in you needs to first make sure you know what you are building and why.

3

C H A P T E R

3 Zoom Out: Thinking in Systems

Early in my time on the enterprise telephony platform, I owned a .NET REST API service that processed customer scheduling requests, coordinating with the platform's routing system to queue and fulfill customer interactions. It was a well-built service. Clean code, good test coverage, fast response times, monitored carefully. I was proud of it. I knew everything about it.

One evening, our platform started dropping scheduling requests at a rate that set off every alert we had. The on-call team paged me. I looked at my service first, because that was what I knew. Everything looked fine. Latency normal. Error rate normal. Queue depth normal. My service was healthy.

I spent forty minutes looking in the wrong place before someone upstream finally traced the issue to an upstream system, which had started responding slowly under an unexpected load spike. My service was healthy because it was rejecting new requests early rather than queuing them. The rejections looked like normal backpressure from my vantage point. From the caller's vantage point, their requests were never being fulfilled.

My service was fine. The system was broken. And I had no model for the difference.

That incident taught me something I have thought about ever since: you can understand every component in a system perfectly and still not understand the system. The system is not the sum of its parts. It is the sum of its parts plus the interactions between them, and those interactions are where the real behavior lives.

3.1 The Shift from Components to Systems

Developers naturally think in components. A component is a bounded, understandable unit: a service, a function, a module, a database. It has inputs and outputs. It has a contract. You can test it in isolation. You can reason about it completely if you study it long enough.

This is one of the great achievements of software engineering: the ability to decompose complex problems into components that can be

understood and built independently. Without it, large systems would be impossible to build.

But decomposition has a cost. When you break a system into components and hand each component to a team or a developer to own, you create a natural pull toward local optimization. Each owner makes their component as good as it can be. They monitor it, tune it, and improve it. The components individually get better and better.

And the system can still fail in ways none of them anticipated.

This happens because systems have properties that no individual component has. A network of services under load behaves differently than any single service under load. A database that performs perfectly in isolation can become a bottleneck when twelve services are hitting it simultaneously with patterns that were never coordinated. A message queue that is well-sized for normal traffic becomes a dangerous buffer when an upstream spike creates a backlog that takes hours to drain, during which downstream consumers are processing stale data.

Architects think in systems. Not instead of components, but in addition to them. They hold both levels simultaneously: what each part does, and what the whole does. This dual vision is not automatic. It is a skill, and like all skills it is built through practice and the right mental models.

> ***You can understand every component in a system perfectly and still not understand the system. The behavior of a system lives in the interactions between its parts, not in the parts themselves.***

3.2 Emergence: When the Whole Surprises You

Systems thinking has a concept called **emergence**: the phenomenon where a system exhibits properties and behaviors that none of its individual components exhibit. These properties emerge from the interactions between components, not from the components themselves.

You have seen this, even if you did not have a name for it. A traffic jam is an emergent property of individual cars following simple rules. No single car is a traffic jam. A market crash is an emergent property of individual investors making individually rational decisions. No single investor is a crash. The behavior arises from the interactions, not the actors.

In software systems, emergence shows up constantly. **Thundering herd** problems emerge when many clients retry simultaneously after a shared dependency recovers. **Cascading failures** emerge when a slow service causes its callers to accumulate threads, which causes their callers to time out, which propagates up the call chain in ways no individual service owner predicted. Data inconsistencies emerge when multiple services write to shared state under assumptions that are each individually valid but collectively contradictory.

The practical implication for architects is this: you cannot reason about system behavior purely by reasoning about components. You have to reason about the interactions. What happens when this service calls that one and the response is slow? What happens when this queue fills up? What happens when two services both decide to refresh their cache at the same time?

These questions do not have obvious answers from reading the component code. They require you to build a mental model of the system as a whole, including the spaces between the components where emergence happens.

3.3 Mental Models Every Architect Needs

A mental model is a simplified representation of how something works, good enough to reason about without being complete. Architects carry a set of mental models for system behavior that they apply to new situations quickly and reliably. Here are the ones I reach for most often.

3.3.1 Queues and Buffers

Any place where work accumulates before it is processed is a queue. Message queues are obvious examples, but queues are everywhere:

HTTP request queues, database connection pools, thread pools, disk write buffers, retry mechanisms. Understanding how queues behave under load is one of the most practically useful things an architect can know.

The core behavior is this: a queue grows when the arrival rate exceeds the processing rate. Once a queue starts growing, it continues to grow until either the arrival rate drops or the processing rate increases. A queue that grows without bound will eventually exhaust whatever resource it is consuming: memory, disk, processing time. This seems obvious, but the failure mode is surprisingly easy to miss in production because queues often grow slowly at first, during normal operation, before load spikes expose them catastrophically.

When you see a queue in a system design, ask: what is the maximum arrival rate? What is the processing rate? What happens when the queue fills? Who is waiting, and for how long? What is the back-pressure mechanism? These questions will surface more problems than almost any other line of inquiry.

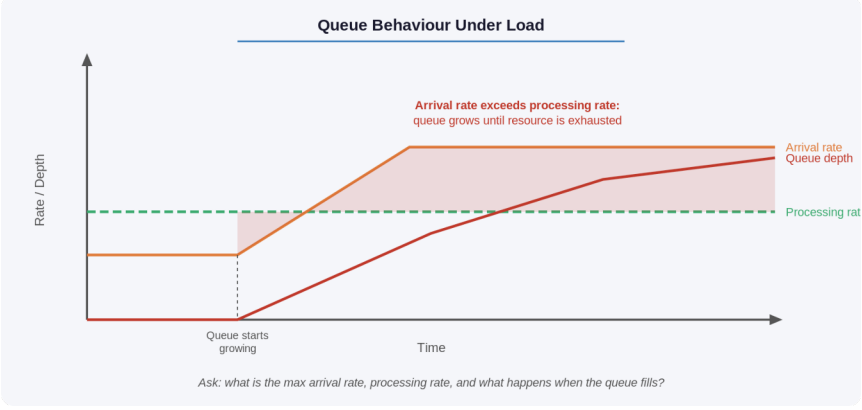


Figure 3.1: Queue depth grows when arrival rate exceeds processing rate

3.3.2 Feedback Loops

A **feedback loop** exists when the output of a process influences its future input. Feedback loops are everywhere in systems, and they come in two flavors: stabilizing and amplifying.

A **stabilizing feedback loop** pushes a system back toward equilibrium when it drifts. A thermostat is the classic example: when temperature rises above the target, the cooling turns on, which lowers the temperature, which turns off the cooling. Many well-designed systems have stabilizing feedback built in: circuit breakers that stop sending traffic to a struggling service, auto-scaling that adds capacity when load rises, rate limiters that slow incoming requests when downstream systems are struggling.

An **amplifying feedback loop**, sometimes called a **positive feedback loop**, pushes a system further from equilibrium when it is disturbed. These are the dangerous ones in software. **Retry storms** are a classic example: a service starts failing, clients retry, the retries increase the load, which causes more failures, which causes more retries. Each cycle amplifies the problem. Amplifying feedback loops are behind most of the catastrophic failures in distributed systems, and recognizing them in a design before they manifest in production is one of the highest-value things an architect can do.

When you review a system design, look for the feedback loops. Ask: where could a disturbance get amplified rather than dampened? Where is the back-pressure? Where are the circuit breakers? What prevents a slow degradation from becoming a runaway cascade?

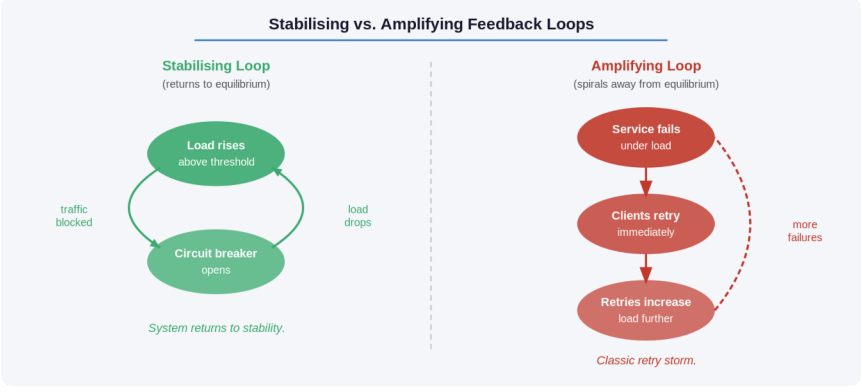


Figure 3.2: Stabilizing loops return to equilibrium; amplifying loops spiral away from it

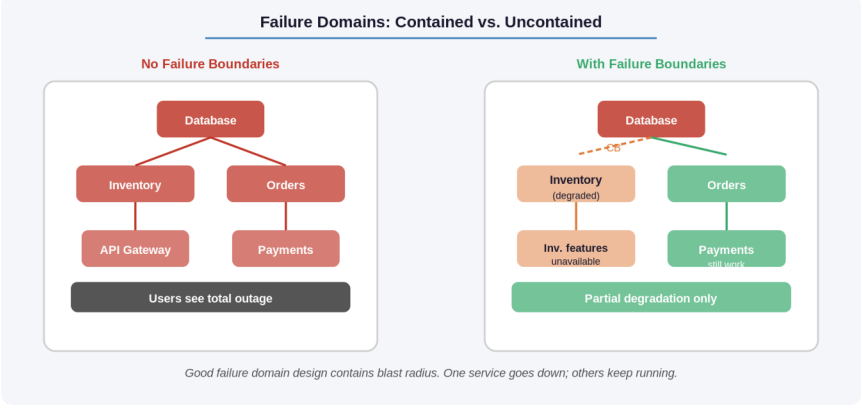
3.3.3 Failure Domains

A **failure domain** is a boundary within which a failure can propagate. Designing good failure domains is one of the core jobs of a systems architect.

The principle is simple: failures are inevitable, so design your system so that when a failure occurs, it stays contained. A database outage should not take down your entire platform. A bad deployment to one service should not prevent users from using unrelated features. A third-party API going down should degrade one workflow, not corrupt the state of the whole system.

Failure domains are defined by decisions about isolation: separate deployments, separate databases, separate network segments, circuit breakers between services, bulkhead patterns that limit how many resources any one dependency can consume. Every boundary you draw between parts of the system is, in effect, a potential failure domain boundary. The question is whether you have drawn those boundaries in places that actually contain failures, or whether you have drawn them based on other concerns (team ownership, feature grouping, technical convenience) without considering the failure propagation properties.

When you look at an architecture diagram, trace the failure paths. If this database goes down, what stops working? If this service starts returning errors, what calls it, and what calls those services? How far does the failure travel before something contains it? If the answer is "all the way to the user, across all features," you have a failure domain problem.



Good failure domain design contains blast radius. One service goes down; others keep running.

Figure 3.3: Failure domain boundaries determine how far a failure travels

3.3.4 Blast Radius

Blast radius is a way of quantifying the impact of a failure. It asks: if this thing breaks, how many users are affected, how many features stop working, and how much data is at risk?

Blast radius is a useful concept because it gives you a language for prioritizing reliability investments. A component with a large blast radius deserves more investment in redundancy, failover, testing, and monitoring than one with a small blast radius. Not everything needs five nines of availability. But the things whose failure would affect your entire user base, corrupt critical data, or create regulatory exposure absolutely do.

Blast radius analysis also shapes deployment strategy. A change that touches a component with a large blast radius should be deployed with more caution: canary releases, feature flags, staged rollouts, the ability to roll back quickly. A change to a low-blast-radius component can be shipped with less ceremony. Treating all deployments the same regardless of blast radius is a risk management failure dressed up as process consistency.

3.4 Reading a System Before You Touch It

One of the most undervalued skills in architecture is the ability to read an existing system: to build an accurate mental model of how it actually behaves before you make any changes to it.

This matters because most architectural work happens in the context of existing systems, not greenfield builds. You will inherit systems you did not design, built by teams you never worked with, solving problems that were framed differently than you would frame them today. Before you can improve those systems, you need to understand them. Not just how they were designed, but how they actually behave in production.

The documentation, if it exists, tells you what someone intended. The code tells you what was implemented. The logs, metrics, and traces tell

you what is actually happening. All three sources matter, and they rarely agree completely. The most important source is usually the last one.

When I approach an unfamiliar system, I follow a rough sequence. I start with the data flows: where does data enter the system, how does it move through it, and where does it exit? Data flows expose the critical paths, the integration points, and the places where failures have the most impact.

Then I look at the dependencies: what does this system rely on, what relies on it, and what are the latency and reliability characteristics of those dependencies? Dependencies are where surprises live. A system that looks simple in isolation often becomes complicated when you see everything it is calling and everything that is calling it.

Then I look at the failure history. Incident logs, post-mortems, known issues in the backlog. How a system has failed in the past is one of the best predictors of how it will fail in the future, and the historical failures usually reveal the failure domains and feedback loops that are not obvious from the design.

Finally, I try to find the people who know the system's secrets: the engineers who have been on-call for it longest, who have responded to its worst incidents, who know where the bodies are buried. Every complex system has institutional knowledge that lives in people's heads and nowhere else. Finding those people and learning from them before you make changes is not just good manners. It is sound risk management.

3.5 A Story: The Cascade I Did Not See Coming

Let me return to that incident from the beginning of this chapter, because there is more to it than I told you.

After we resolved the immediate crisis, we did a post-mortem. The proximate cause was clear: the upstream system slowed down, my service started rejecting requests, scheduling requests were dropped. Simple enough.

But as we traced the causal chain further back, we found something more interesting. The upstream system had slowed because it was hitting its database connection pool limit. It was hitting its connection pool limit because a recently deployed reporting component had started making synchronous calls to the upstream system as part of its own response path. The reporting team had not told us this was coming, and we had no visibility into our incoming dependency graph, only our outgoing one.

The reporting component was new and traffic to it was growing. Every new query to the reporting component generated a synchronous lookup against the upstream system. As reporting traffic grew, so did the load on the upstream system, but gradually enough that no single alert threshold was breached until the connection pool suddenly maxed out under a moderate traffic spike.

No single team had done anything wrong. My service was behaving correctly. The upstream system was behaving correctly. The reporting component was behaving correctly. The system as a whole had an emergent failure mode that none of the individual component owners could see, because none of them had a model of how the components interacted at scale.

After this incident, we built a dependency map. We instrumented inter-service calls. We established a process for notifying teams when a new dependency on a shared platform was being introduced. These were not technically complex changes. They were visibility changes. We gave the system's operators a way to see the system as a system, rather than as a collection of independent services.

The cascade stopped happening. Not because we fixed any one component, but because we could now see the interactions in time to manage them.

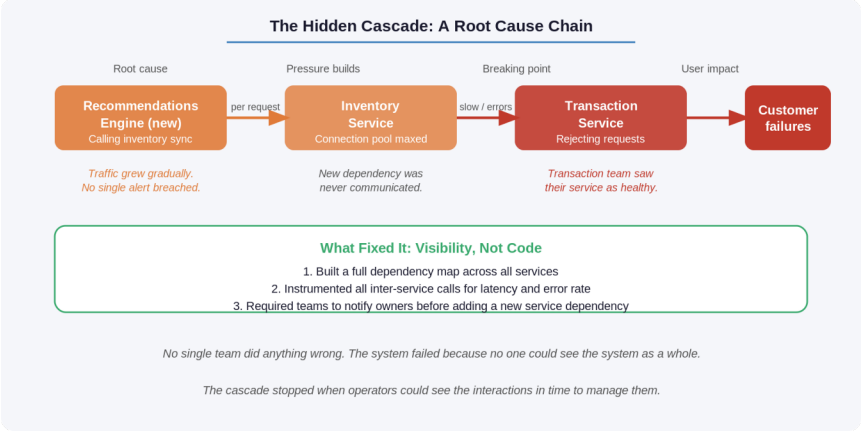


Figure 3.4: The hidden dependency chain behind the cascade failure

3.6 What to Take Away

Systems thinking is not an abstract concept. It is a practical discipline with concrete tools: queues, feedback loops, failure domains, blast radius. These mental models let you reason about system behavior before you observe it failing in production.

The most important habit to build is the habit of zooming out. Every time you are looking at a component, practice asking what it connects to, what depends on it, and what happens to those connections when the component behaves unexpectedly. Over time this becomes automatic, and your design decisions will start to reflect it without you having to consciously remind yourself to think this way.

And when you join a new system or inherit one you did not build, invest the time to read it before you change it. Talk to the people who know it. Look at the failure history. Build your mental model of the whole before you start optimizing the parts.

Developers who think in components build excellent services. Architects who think in systems build systems worth building.

14

C H A P T E R

14 How to Make Architectural Decisions

During the convergence of two separate platforms into a single system, one of the key decisions was how to handle the message queuing layer. Both platforms had grown their own queuing implementations. We needed to pick one approach for the converged system. The decision to use a distributed broker rather than a database-backed queue seemed obvious. I documented it in a team wiki page, got verbal agreement in a planning meeting, and started building.

Two months later, I discovered that a parallel workstream, a team building the integration layer with the routing system, had evaluated the same options independently and chosen a different broker for their side of the integration. Nobody had known the two workstreams were solving overlapping problems simultaneously. And nobody had been aware that a third team had evaluated both options six months earlier for a different project and had documented concerns about the distributed broker's operational overhead.

We had made the same decision three times, in three slightly different ways, with no coordination and no shared understanding of why each team had chosen what they had.

The decision itself was reasonable. The process was not. We had no record of what we had evaluated, why we had chosen our approach over the alternatives, what constraints had shaped the choice, or what we had consciously traded away. When a senior stakeholder asked me to justify the choice, I could not reconstruct the full reasoning three months after the fact. The meeting notes had been overwritten. The wiki page had been moved during a reorganization.

This is the problem that **Architectural Decision Records (ADRs)** were invented to solve. Not the decision itself, but the record of the decision: who made it, what they knew at the time, what they chose, what they considered and rejected, and what they knew they were trading away. That record is what allows a team to learn from its own history, to avoid making the same decision three times, and to revisit a decision when the circumstances that produced it no longer apply.

14.1 The Anatomy of an Architectural Decision

Every significant architectural decision has five components, whether you document them or not. Making them explicit is the core practice.

The first component is the context: the situation that makes a decision necessary. What is the problem? What are the forces at play? What constraints, either technical, operational, financial, or organizational, limit the options? Context is the most important part of an ADR because it is the part that becomes invisible over time. The decision itself is visible in the codebase. The context that produced it is not. Without the context, a future engineer looking at an architectural choice cannot tell whether it was wise, whether it was forced, or whether the conditions that made it appropriate still apply.

The second component is the decision itself: what was chosen and, crucially, why that option over the alternatives. This is not a list of options that were considered. It is a statement of what was decided, expressed clearly enough that a new team member can understand it without asking anyone. It should answer the question: if I were defending this decision to a skeptical colleague, what would I say?

The third component is the alternatives that were considered and why they were rejected. This is the part most teams skip, and it is the part that saves the most time. When a team considers reversing a decision, the first question is always "did anyone think about option X?" If the ADR documents that option X was considered and rejected for specific reasons, the conversation is short. If there is no record, the team relitigates the original decision from scratch.

The fourth component is the consequences: what becomes easier and what becomes harder as a result of the decision. Good architectural decisions are not free. They trade one set of costs for another. Documenting the trade-offs explicitly is an act of intellectual honesty that forces the decision-maker to acknowledge what they are giving up, and that creates an audit trail for later when the costs materialize.

The fifth component is the status: proposed, accepted, deprecated, or superseded. A decision that has been reversed or replaced should not be silently deleted. It should be marked as superseded, with a reference to

what replaced it and why. The history of reversals is as informative as the history of decisions.

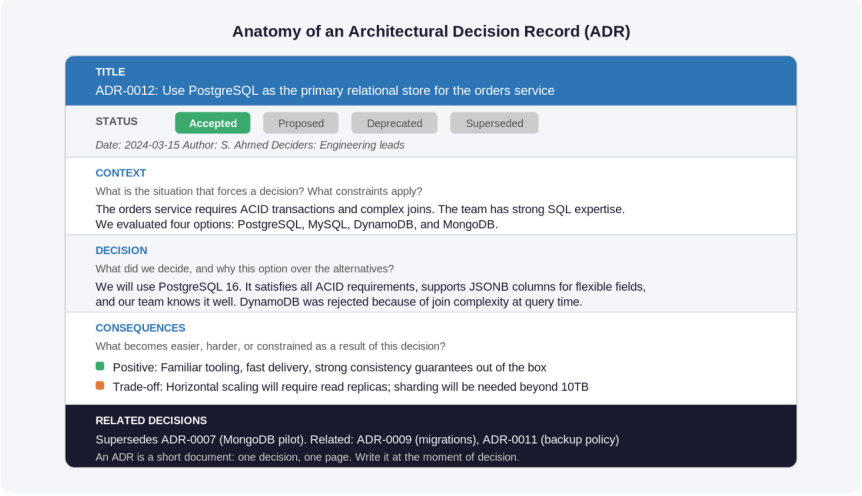


Figure 14.1: The five components of an Architectural Decision Record

14.2 Reversible vs Irreversible Decisions

Not all decisions deserve the same amount of deliberation. One of the most useful mental models for architectural decision-making is the distinction between **two-way doors** and **one-way doors**, a distinction popularized by Jeff Bezos in his 2015 Amazon shareholder letter.

A two-way door decision is one that can be reversed at acceptable cost. If you make it and it turns out to be wrong, you can go back through the door. The right posture for two-way door decisions is to make them quickly, with good-enough information, and course-correct if needed. Excessive deliberation on two-way door decisions is a form of organizational waste. It slows delivery without meaningfully improving outcomes.

A one-way door decision is one that cannot be reversed without prohibitive cost. Once you walk through it, you are committed. The right posture for one-way door decisions is to slow down, gather more

information, involve more stakeholders, explore the consequences more deeply, and document the reasoning thoroughly. The cost of deliberation is far lower than the cost of getting it wrong.

The judgment that matters is knowing which kind of decision you are facing. Most developers, when they become architects, default to treating every decision like a one-way door: lengthy analysis, broad stakeholder consultation, extensive documentation. This approach is appropriate for a small fraction of decisions and paralyzes the team for the rest. The skill is not thoroughness applied uniformly. It is calibration: spending deliberation where it is warranted and moving quickly where it is not.

A useful heuristic: a decision is effectively irreversible when reversing it would require rebuilding or migrating something significant, breaking a contract with an external party, or coordinating a change across many teams simultaneously. Database engine choices, public API contracts, core data model design, cloud provider selection for a greenfield system: these are one-way doors. Logging library choices, internal naming conventions, the specific queue implementation for a new service: these are two-way doors.

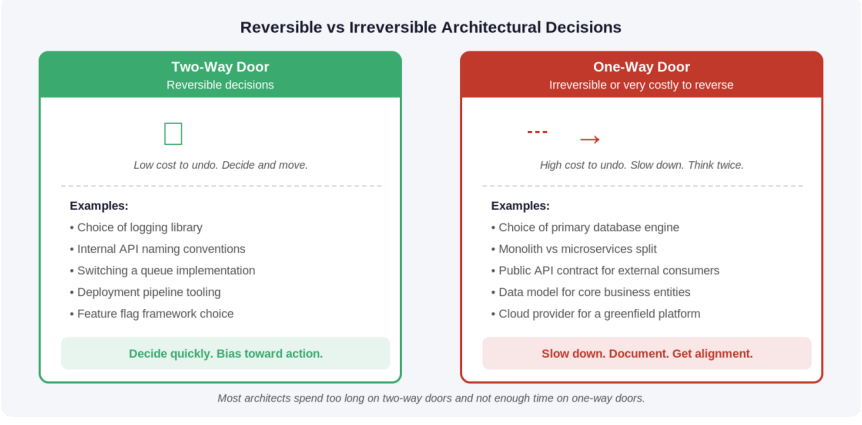


Figure 14.2: Two-way and one-way doors: calibrate deliberation to the cost of reversal

14.3 Architectural Decision Records

An Architectural Decision Record is a short document that captures a single architectural decision. Short is the key word. An ADR should fit on one page. If it cannot, the decision is probably two decisions pretending to be one, or the document has drifted from the decision into a design document.

The format that has worked best across most of the teams I have seen use ADRs is simple: a numbered identifier and descriptive title, a status, a context section, a decision section, a consequences section, and optionally a section noting related or superseded decisions. That is it. The value is not in the format. The value is in writing it down at all.

The timing matters as much as the format. An ADR written at the moment of decision captures the actual reasoning: what the team knew, what they were uncertain about, what they weighed. An ADR written six months after the fact is a rationalization. It will describe the decision that was made in the best possible light, because the alternative, acknowledging that the team did not think through the consequences, is uncomfortable. Write ADRs when the decision is made.

ADRs should live where the code lives: in the repository, in a directory called docs/decisions or similar, committed alongside the code they govern. This keeps the decision record close to what it describes, makes it visible in code review, and ensures it is versioned alongside the code. A decision record that lives in a wiki will be edited, moved, and eventually lost. A decision record in the repository survives team turnover, company acquisitions, and wiki migrations. A ready-to-use template is at devtoarch.com/templates/adr.

14.3.1 Getting Teams to Actually Write ADRs

The practical obstacle is not format or tooling. It is habit. Engineers are trained to focus on code, not documentation. Getting a team to write ADRs consistently requires two things: making it part of the definition of done for significant decisions, and making the template trivially easy to fill in.

The lightest-weight approach that works is a short template in the repository with placeholder text for each section. When a significant decision is being discussed, whoever is driving it opens a new ADR file, fills in the context and proposed decision before the decision meeting, and updates the status and consequences after. The PR for the work that implements the decision includes the ADR. Reviewers check the ADR as part of code review.

Over time, the ADR log becomes one of the most valuable artifacts in the repository. New team members read through the last thirty or forty ADRs and understand the architectural history of the system faster than any amount of wiki reading. When a system is being handed to a new team, the ADR log transfers institutional knowledge that would otherwise leave with the departing engineers.

14.4 When to Decide, When to Delay, When to Delegate

One of the underappreciated skills in architectural decision-making is knowing when not to decide yet. Premature decisions, made before the information that would make them obvious is available, create unnecessary constraints. The discipline is distinguishing between a decision that needs to be made now to keep moving, and one that can be deferred until the picture is clearer without blocking anything important.

The principle to internalize is: delay irreversible decisions until the **last responsible moment**. The last responsible moment is the latest point at which a decision can be made without foreclosing options that will still be needed. This is not the same as the latest possible moment. It is the moment beyond which delay becomes more costly than deciding with incomplete information.

For many early-stage architectural decisions, that moment is further away than it feels. It feels urgent to decide on a messaging system before the first message is sent. But the choice of messaging system often has no bearing on the first six months of development, and deciding now forecloses the option of learning from the system's actual usage patterns

before committing. The last responsible moment may be when the first message needs to go into production, not when the first service is being built.

Delegation is the third option, alongside deciding and delaying. Some decisions should not be made by the architect at all. A decision about how a particular team structures their internal service boundaries, where the teams have more context than anyone else, should be delegated to that team with guidelines rather than mandated from above. A decision about which testing framework to use within a service is not an architectural decision at all and should not be on an architect's agenda.

Knowing what to delegate requires a clear sense of what architectural decisions actually are. An architectural decision is one that affects multiple teams, crosses service boundaries, constrains future choices in significant ways, or involves non-trivial trade-offs between competing quality attributes. Decisions that affect only one team, are easily reversible, and have limited cross-system consequences should be made by the people closest to the problem.

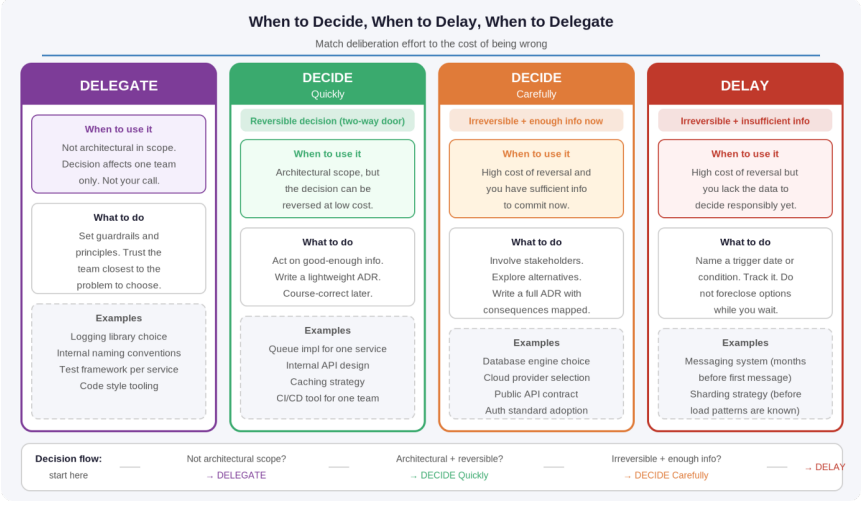


Figure 14.3: The three decision postures (decide, delay, or delegate), keyed to reversibility, urgency, and architectural scope

14.5 A Story: The Decision Nobody Wrote Down

The platform convergence situation was not the last time I encountered the problem of undocumented decisions. It was the one that convinced me to change my approach permanently.

After the duplicate decision incident, I introduced a lightweight ADR process into the team. The template was four sections, fit on one page, and lived in the repository. We agreed that any decision affecting another team, any decision involving a trade-off between quality attributes, and any decision that would be difficult to reverse required an ADR.

The first few ADRs were awkward. Engineers wrote them as if they were justifying decisions to a hostile auditor rather than informing a future colleague. The context sections were defensive rather than descriptive. The consequences sections listed only benefits.

I introduced one additional practice: I wrote the first draft of the consequences section for each ADR myself, based on what I thought the real trade-offs were, and asked the engineer to correct it if I had gotten anything wrong. This had two effects. It modeled the level of honesty I wanted to see, and it created a genuine conversation about what the team was actually giving up with each choice.

Two years later, when the platform was handed to a new team during a reorganization, the engineering lead walked them through the repository, opened the docs/decisions directory, and said: "This is how the team made decisions." There were thirty-one ADRs spanning two and a half years. The new team read them all. They were up to speed in a week rather than a month.

14.6 Build vs. Buy vs. Open Source

One of the decisions architects face most frequently, and often underestimate, is whether to build a capability in-house, purchase it from a vendor, or adopt an open-source solution. The question sounds

straightforward. In practice, it involves trade-offs that play out over years and that are very difficult to reverse once made.

The instinct to build is strong among engineers. Building feels like control. It produces something tailored precisely to the need, with no licensing cost and no dependency on a vendor's roadmap. The instinct is also frequently wrong. Building a capability means owning it forever: maintaining it, extending it, documenting it, onboarding new team members onto it, and fixing it at 3am when it breaks. Every hour spent maintaining custom infrastructure is an hour not spent on the business problem the company exists to solve. The question is never "can we build it?" It is "should we build it, given what it will cost to own it?""

Buying a SaaS product or vendor solution shifts the maintenance burden to the vendor and typically compresses time to value. The trade-offs are vendor lock-in, recurring cost that scales with usage, and a ceiling on customization. Vendor lock-in is the one that architects underestimate most consistently. A capability that is core to your product's differentiation should not be owned by a vendor who can reprice it, deprecate it, or be acquired and absorbed. A capability that is genuinely commodity, authentication, email delivery, payment processing, observability tooling, is a candidate for buying because the cost of undifferentiated ownership is high and the vendor's specialization produces a better outcome than you would build in the same time.

Open source occupies a middle position. The software is free to use, inspect, and modify. The maintenance is yours to perform, or you can buy commercial support from the vendor or community that maintains it. Open source is most attractive when the capability is complex enough that building it from scratch is prohibitive, when you need the ability to inspect and modify the internals, and when a mature ecosystem exists around the project. The risks are adoption of a project that is abandoned, the cost of upgrading across breaking versions, and the organizational overhead of managing open-source dependencies at scale.

The decision framework I use has three questions. First: is this capability part of our core product differentiation? If yes, build. The competitive advantage lives in how this capability works, and that advantage should not be delegated to a vendor or a community. Second: is this a commodity capability with mature, well-supported options? If yes, buy or adopt open source. The cost of custom ownership is high and the

differentiation is low. Third: do we need the ability to inspect internals, modify behavior, or avoid per-seat pricing at scale? Open source is typically the answer when two of those three are true.

The most expensive mistake in this decision is building what should be bought. The second most expensive is buying something that should be core to the product. The third, and most insidious, is adopting an open-source project without honestly assessing the total cost of ownership: upgrades, security patches, operational support, and the organizational knowledge required to run it well. All three mistakes are common. The way to avoid them is to make this decision explicitly, document it in an ADR, and revisit it when circumstances change, when the vendor reprices, when the open-source project slows down, or when the custom-built solution has become a maintenance burden nobody wants to own.

14.7 What to Take Away

The quality of your architectural decisions matters less than you think. The quality of your decision-making process, including how you document, communicate, and revisit decisions, matters more. A mediocre decision that is documented, shared, and revisited when circumstances change does less long-term damage than a brilliant decision that nobody can reconstruct twelve months later.

Write Architectural Decision Records at the moment of decision. Keep them short: one decision, one page. Store them in the repository. Mark them superseded rather than deleting them when they are reversed.

Calibrate your deliberation to the reversibility of the decision. Two-way doors deserve quick action. One-way doors deserve slow thinking. Most architects default to the wrong setting for each category.

And when you are not sure whether to decide now or wait: ask whether deferring will keep options open or just create more uncertainty. If it keeps options open, defer. If it just delays the inevitable, decide.

27

CHAPTER

27 The First 90 Days as an Architect

I spent the first three weeks of my first architect role drawing diagrams. The systems were large and unfamiliar, there were six teams touching different parts of the platform, and I had been brought in, at least partly, because the engineering leadership felt the architecture had drifted and needed attention. I could see the drift from the outside. I wanted to understand it from the inside before I said anything about it.

The CTO stopped by my desk after two weeks and asked what I was working on. I showed him the maps I had been making, system boundaries, data flows, dependency graphs, the places where the dependencies made no sense given what I now understood of the business domains. He looked at them for a while and said: "Nobody has ever drawn this before."That was not a compliment about my drawing ability. It was an indication of how little anyone had looked at the system whole.

I spent another week in conversations, one-on-ones with every engineering manager, a long lunch with the principal engineer who had been there longest, a session with the head of product who had been trying to articulate why certain features took so long. By the end of week four I had a picture of the system, the organization, and the problems. I had not yet proposed a solution to anything.

That restraint was the most important thing I did in those first four weeks. What I found in week five, when I did start proposing, was that every proposal landed because it was grounded in understanding rather than assumption. People could see I had done the work of listening. They trusted the conclusions because they could see where they had come from.

This chapter is about those first ninety days, what to do, what to avoid, and why the instinct to change things quickly is the instinct most likely to undermine everything that follows.

27.1 What to Do Before You Redesign Anything

The new architect's most common mistake is also the most understandable one: they arrive with ideas and start proposing them.

They have read the situation from the outside, formed a view, and are eager to demonstrate that the view is correct and that they are capable of acting on it. This eagerness is natural. It is also damaging.

The problem is not that the ideas are wrong. Sometimes they are exactly right. The problem is that they are formed without the context that only comes from being inside the system, inside the organization, and inside the relationships that shape how technical decisions get made. A proposal formed without that context, even a technically correct one, will be resisted, not because the people resisting it are obstructive, but because the proposal does not reflect an understanding of the constraints that are shaping their world.

The first ninety days are a listening and learning exercise. The goal is not to solve problems. It is to understand them accurately enough that the solutions you eventually propose have a chance of being adopted. That requires patience that the role's urgency makes genuinely difficult to maintain. Maintain it anyway.

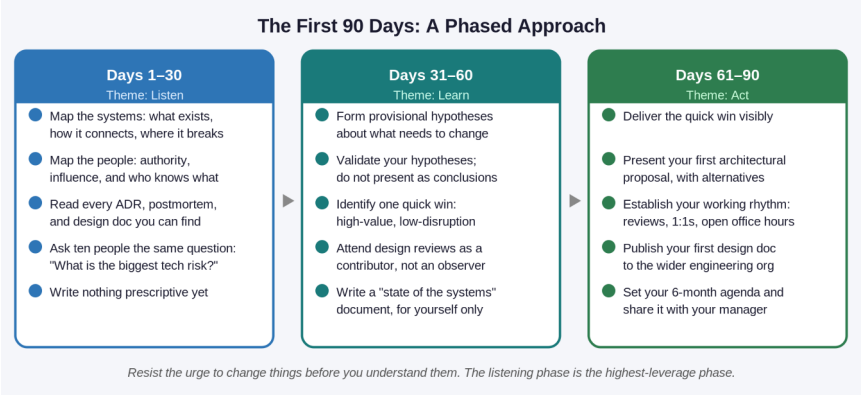


Figure 27.1: The first 90 days: three phases of listening, learning, and acting

27.2 Listening Tours: Understanding the System and Its History

A **listening tour** is a structured series of conversations designed to build a picture of the technical system, the organizational dynamics, and

the problems that matter most. It is the architect's equivalent of the ethnographer's fieldwork: immersive, systematic, and deliberately non-prescriptive.

The structure is simple. Identify the key people, engineering managers, senior developers, on-call engineers, product managers, recent hires. Schedule thirty to forty-five minutes with each of them. Prepare three or four open questions and ask the same ones across all the conversations. Listen more than you speak. Take notes. Look for the patterns in the answers, where multiple people independently identify the same problem, that problem is real.

27.2.1 The Questions That Matter

The most useful single question in a listening tour is: "What is the biggest technical risk you see right now?" Ask it of everyone. The answers are almost always revealing, and the variation across roles is often more informative than any individual answer. An engineering manager who says "we have too much technical debt to move fast" and a product manager who says "I can never get a reliable estimate because the codebase surprises us constantly" and a senior developer who says "the data model is wrong and we know it" are all describing the same underlying problem from different vantage points. The architect who can see all three perspectives simultaneously begins to understand the real shape of what needs to change.

Other useful questions: "What would make your team meaningfully faster?" "What is the one part of the system you would not touch if you could avoid it, and why?" "If you could change one architectural decision that was made before you joined, what would it be?" These questions surface the institutional memory of pain, the things that everyone knows are wrong but nobody has had the mandate or the energy to fix.

Figure 27.2: The listening tour: six constituencies, tailored questions, consistent core inquiry

27.2.2 Reading the Archaeological Record

Before the listening tour is complete, read everything written. Every **architecture decision record** in the repository. Every postmortem. Every design document, however old. Every RFC, including the ones that were rejected.

This archaeological reading is not primarily about understanding the current state of the system. It is about understanding the decisions that produced it, the constraints that existed at the time, the options that were considered and rejected, the assumptions that turned out to be wrong. An architect who understands why the system is the way it is will make much better decisions about how to change it than one who only understands what it currently does.

The rejected RFCs are especially valuable. A proposal that was discussed, debated, and ultimately not adopted tells you something about the organization's technical values, its risk tolerance, and the constraints that were operative at the time. Some of those constraints will have changed. The fact that a proposal was rejected in the past does not mean

it is wrong now. But you need to understand the rejection before you can assess whether the conditions have changed.

27.3 Quick Wins That Build Credibility

A quick win in the first ninety days is not about proving that you have good ideas. It is about demonstrating that you can translate understanding into action, that the listening was not an academic exercise but a foundation for change that actually ships.

The criteria for a first quick win are specific: it should be visible enough that the engineering organization notices it, contained enough that it can be completed in under four weeks, grounded in something you learned during the listening tour rather than something you brought with you, and low enough in risk that getting it slightly wrong does not matter. The goal is not the win itself. The goal is establishing the pattern of how you work: listen, understand, propose, deliver.

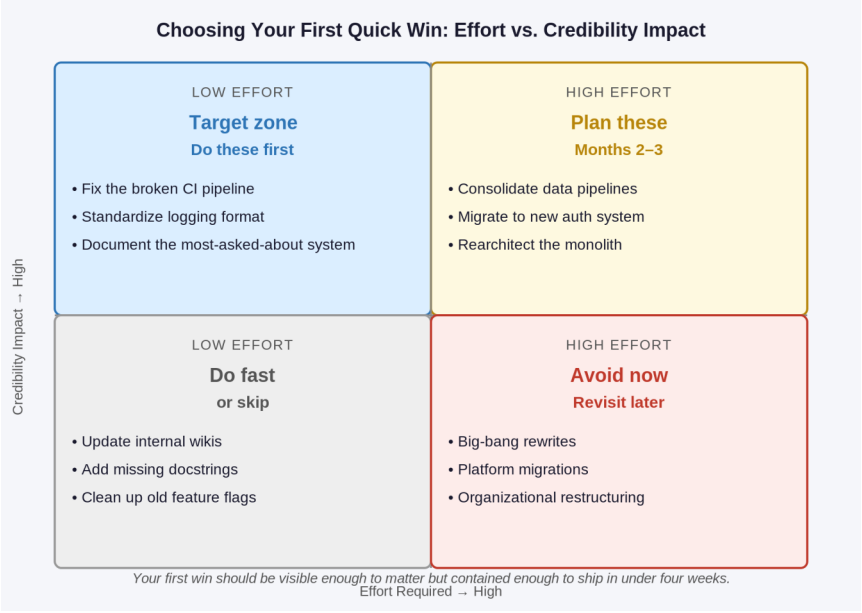


Figure 27.3: Choosing your first quick win: the effort-versus-credibility matrix

27.3.1 What Good Quick Wins Look Like

The best first wins tend to be in one of a few categories. **Observability improvements**, adding structured logging to a component that was previously opaque, creating a dashboard for a metric everyone cared about but nobody had assembled, are high-value and low-disruption. They make the system more legible without changing how it behaves. **Documentation that closes a known gap**, writing the architecture overview that everyone knew was missing, creating the onboarding guide that new engineers said they wished existed, is similarly valuable. It demonstrates understanding and produces something concretely useful.

Process improvements that reduce friction, standardizing a CI pipeline configuration that had diverged across teams, proposing a lightweight ADR template that makes decision documentation easier, are quick wins that benefit multiple teams simultaneously. They are the kind of change that is easy to say yes to because the cost is low and the benefit is visible.

What good first wins do not look like: migrations, redesigns, consolidations, or anything that requires more than one team to change their behavior simultaneously. Those are second or third quarter work. Do the contained thing first.

27.4 The Trap of the New Architect

The trap is specific and nearly universal. The new architect arrives, spends a few weeks looking around, forms a view about what needs to change, and then begins changing things, simultaneously, enthusiastically, and faster than the organization can absorb. Within six months, they have alienated three teams, created two half-finished initiatives that nobody is maintaining, and established a reputation for disruption rather than for judgment.

The pattern is so common it has a name in engineering organizations: the **new architect effect**. It is not caused by incompetence. It is caused by the combination of genuine insight, the new architect often does see things clearly, and insufficient appreciation of the organizational cost of

change. Systems can only absorb so much change at once. Teams can only participate in so many initiatives simultaneously. An architect who launches five things at once ensures that none of them will be done well.

27.4.1 The Principle of Minimum Simultaneous Change

The antidote is a principle that experienced architects apply automatically and new architects have to learn deliberately: change one significant thing at a time. Not because the other things are not important, they may be more important, but because the organization's capacity to absorb architectural change is a real constraint, and working within it is the difference between an agenda that gets adopted and one that gets ignored.

This does not mean moving slowly. It means sequencing deliberately. Identify the highest-priority change. Deliver it. Then identify the next. The sequence is faster in practice than the parallel approach, because each completed initiative builds the credibility and organizational alignment that makes the next one easier to adopt.

27.4.2 Changing Too Much, Too Soon

The specific failure mode worth naming is the proposal that is technically correct but organizationally premature. An architect who has spent four weeks in the codebase and identified a genuine structural problem may be entirely right about the problem and entirely wrong about the timing. The organization that is two weeks from a major product launch is not available to absorb a platform migration, regardless of how well-argued the case for it is. The team that has just shipped a painful re-organization is not in a position to take on significant additional technical change.

Reading the organizational moment, understanding what the teams around you are absorbing, what they are committed to, what capacity they have for change beyond their current commitments, is a skill that develops with time and attention. In the first ninety days, when that skill is still developing, the safe default is to do less than you think you could and to do it better than anyone expected.

27.5 Establishing Your Working Rhythm

By the end of the first ninety days, the working rhythm should be established and visible: the recurring touchpoints, the channels for architectural input, the places where decisions are made and where you need to be present to influence them.

For most architect roles this includes a set of regular one-on-ones, with engineering managers whose teams you support, with the principal engineers whose judgment you respect and who need to respect yours, and with your own manager about the strategic agenda and how it is developing. It includes a regular presence in design reviews and architecture forums. And it includes a pattern of written output: design documents, ADRs, technical proposals, and the occasional broader communication that keeps the engineering organization informed about the architectural agenda.

27.5.1 The 90-Day Review

At the end of the ninety days, do a structured retrospective, with yourself and with your manager. What did you learn? What did you get right in the assessment? What turned out to be different from what you expected? What have you delivered? What is the agenda for the next six months?

Write it down. The written review serves two purposes. It creates accountability, a record of what you committed to and what you delivered against it. And it creates a baseline that makes it possible to demonstrate progress over time. The architecture work that is hardest to make visible is the work of preventing problems that never happen. The written agenda and the periodic review are how you make that work legible. A 90-day plan template for tracking this work is at devtoarch.com/templates/90-day-plan.

27.6 A Story: The Redesign I Did Not Ship

In the second month of a previous architect role, I had identified what I was certain was the central structural problem in the platform: a

data model that had been designed for a product that no longer existed, and that was now contorting every new feature into shapes it had not been designed for. The evidence was everywhere. The design documents said it. The postmortems said it. Every senior developer I spoke to said it.

I wrote a proposal. It was thorough. It was technically correct. It addressed the migration path in detail, acknowledged the risk, and proposed a phased approach. I was proud of it. I shared it at the architecture forum in week nine.

The response was not hostile. It was exhausted. There were four teams currently in the middle of other significant work. Two of them had just finished a painful migration the previous quarter. The principal engineer who knew the data model best had just come back from sick leave. The timing was wrong in every direction.

I withdrew the proposal, not because it was wrong, but because it could not succeed in those conditions. I filed it away. Six months later, when two of the migrations were complete and the team had more capacity, I brought it back. It shipped in the following quarter with unanimous support.

The proposal had not changed. The conditions had. Knowing the difference is the work.

27.7 What to Take Away

The first ninety days are not about demonstrating what you know. They are about demonstrating how you work. The architect who spends thirty days listening before proposing anything establishes a pattern of judgment that compounds through the rest of their time in the role.

Do the listening tour. Read the archaeological record. Find the quick win that is grounded in understanding rather than assumption. Resist the urge to change multiple things at once. Establish the working rhythm. Then write the ninety-day retrospective and set the agenda for what comes next.

The trap, changing too much too soon, is not a failure of ambition. It is a failure of pacing. The architects who have the most impact over time are the ones who understand that the organization's capacity to absorb change is a real constraint, and who design their agenda to work within it rather than against it.

APPENDIX E

Recommended Diagramming Tools

The right tool for an architecture diagram depends on who will use it, how often it will change, and where it will live. The summary below covers the tools most commonly used in practice, with honest assessments of where each one works best and where it falls short. Current links and pricing are maintained at devtoarch.com/resources/diagramming-tools.

E.1 For Collaborative Design Sessions

Miro is the closest digital equivalent to a room with a physical whiteboard. It is optimized for real-time collaboration, which makes it ideal for architecture workshops, event storming sessions, and any situation where a group of people need to think through a design together. It is not a good tool for canonical reference diagrams, because the freeform format makes it hard to enforce consistent notation and harder still to keep current over time.

FigJam (Figma) offers similar capabilities with a cleaner interface and tighter integration with Figma for teams that use both. Mural is an alternative with strong facilitation features for larger workshops.

E.2 For Reference Diagrams (Maintained Alongside Code)

Structurizr is the tool designed specifically for C4 model diagrams. You define your architecture in a domain-specific language and Structurizr renders it into context, container, and component views from the same underlying model. Changes to the model propagate to all views automatically. The diagram-as-code approach enables version control, code review of diagram changes, and consistent notation across the entire system. For teams that take architecture documentation seriously, Structurizr or Structurizr Lite (the self-hosted open-source version) is the strongest option.

PlantUML and Mermaid are text-based diagramming tools that integrate well with repositories and documentation sites. Mermaid in

particular renders inside GitHub markdown, making it a low-friction choice for sequence diagrams, flowcharts, and simple architecture sketches that live in the repository. Neither enforces C4 model conventions, but both work well for technical documentation that needs to stay close to the code.

E.3 For General-Purpose Architecture Diagrams

Draw.io (also available as diagrams.net) is free, integrates with Google Drive, Confluence, and VS Code, and exports diagrams as XML that can be stored in a repository. It is the most widely accessible option and covers the full range of diagram types. The notation is entirely manual, which means discipline is required to maintain consistency. Lucidchart offers the same capabilities with better real-time collaboration and a more polished interface, at a subscription cost.

Microsoft Visio remains widely used in enterprise environments, particularly in organizations with existing Microsoft tooling investments. It has the most extensive library of shapes and integrates with SharePoint and Teams. For organizations not already using it, the cost and learning curve are rarely justified compared to the alternatives.

E.4 For Infrastructure and Cloud Diagrams

Cloudcraft specializes in AWS infrastructure diagrams and can import an existing AWS account configuration to generate a diagram automatically. Equivalent tools exist for Azure and GCP. These are valuable for operational documentation but should not replace the higher-level C4 diagrams that communicate architecture intent rather than infrastructure topology.

The cloud providers also offer their own diagram tools: AWS Architecture Center, Azure Architecture Center, and Google Cloud Architecture Framework all provide official diagram templates and icon sets. These are useful when producing architecture documentation for audiences inside those ecosystems.

E.5 When to Use a Whiteboard or Napkin

Any diagram drawn for thinking rather than communication should be drawn by hand. When you are working through a design alone, when you are explaining something quickly to a colleague in real time, or when you are in a meeting and need to make an idea visible, the overhead of opening a tool is a distraction. Draw it on paper, on a physical whiteboard, or on a tablet.

Napkin diagrams do not need to be archived. They serve the moment and can be discarded when that moment passes. If the conversation produces something worth keeping, transcribe the key ideas into a proper tool afterward, when you have the time to do it properly. The only mistake with a napkin diagram is spending more time making it look good than thinking about whether the design is right.

E.6 A Note on Tooling Investment

The best diagramming tool is the one your team will actually use and maintain. A team that consistently maintains accurate diagrams in Draw.io produces more architectural value than a team that has adopted Structurizr but finds it too much friction and lets the diagrams fall out of date. Start with the simplest tool that meets your needs. Add capability when the simpler tool becomes a genuine constraint.

APPENDIX F

Certifications Worth Pursuing

Certifications are not a substitute for experience, and no hiring panel has ever promoted someone to principal architect because they held a certificate that a developer without one did not. That said, certifications serve real purposes: they provide structured frameworks for self-assessment, they signal domain commitment to employers who do not yet have direct evidence of your capability, and the better ones teach material that is genuinely useful.

The value of a certification depends almost entirely on what it represents in the domain you are targeting. A certification in a framework nobody uses in your industry is a line on a CV that nobody will ask about. A certification in a framework that your organization's clients routinely audit against is meaningful professional currency. Know the difference before you invest the time.

What follows is a practical guide to the certifications most relevant to architects and developers making the transition, organized by category and ranked within each by the frequency with which they matter professionally. Current links and pricing for each certification are at devtoarch.com/certifications.

H.1 Architecture Certifications

H.1.1 TOGAF (The Open Group Architecture Framework)

TOGAF is the most widely recognized enterprise architecture framework in the world. The certification comes in two levels: Foundation (Part 1) and Practitioner (Part 2). Foundation tests knowledge of the TOGAF framework, its terminology, and its core concepts. Practitioner tests the ability to apply those concepts to architectural scenarios.

When it matters: TOGAF certification is most relevant in large enterprise environments, consulting firms, and organizations that do formal EA governance. Financial services, government, and large-scale IT services firms are the most common contexts where TOGAF credentials are actively sought. If your target environment does none of this, TOGAF is unlikely to differentiate you.

When to pursue it: after you have a working understanding of the enterprise architect role and have been involved in at least one cross-organizational architecture initiative. The framework makes most sense when you have context to understand what problem it is solving. Pursuing it as the first thing you do after deciding to become an architect typically results in learning a vocabulary without understanding why it exists.

Study path: The Open Group publishes official study guides. Third-party courses from providers such as The Art of Service and Simplilearn are widely used. The exam itself is open-book for Part 2, which tests application of the framework under constraints rather than pure recall.

H.1.2 iSAQB CPSA (International Software Architecture Qualification Board)

The iSAQB Certified Professional for Software Architecture is structured in two levels: Foundation and Advanced. The Foundation level covers the vocabulary, concepts, and basic application of software architecture. The Advanced level consists of modules in specific domains, such as domain-driven design, software architecture documentation, and evolutionary architectures, each earning credits toward the CPSA-A designation.

When it matters: iSAQB credentials are most valued in German-speaking markets and in organizations with European engineering leadership. They are recognized across continental Europe more broadly, and the modular Advanced structure means you can pursue the specific technical domains most relevant to your work.

When to pursue it: the Foundation level is appropriate for developers with two or more years of architectural involvement who want structured validation of their foundational knowledge. The Advanced modules are appropriate once you have decided which architectural domains you want to develop specialization in.

H.1.3 AWS Certified Solutions Architect

The AWS Solutions Architect certification comes at two levels: Associate and Professional. The Associate level tests the ability to design resilient,

cost-optimized, and secure applications on AWS. The Professional level tests the ability to design and evaluate complex, multi-account, multi-region architectures for enterprise workloads.

When it matters: nearly universally. AWS is the dominant cloud provider across most industries, and the Solutions Architect credential is one of the most recognized technical certifications in the industry. For any architect working on cloud-native or cloud-migrated systems, this credential is close to a baseline expectation in many organizations.

When to pursue it: Associate level is appropriate once you have six months of hands-on AWS experience. Professional is appropriate after two or more years of AWS architectural work. Do not pursue Professional before Associate; the jump in complexity is significant and the Associate credential provides genuinely useful structured learning even for experienced practitioners.

Study path: Adrian Cantrill's courses are widely regarded as the most thorough preparation. Stephane Maarek's Udemy courses are a faster alternative. Supplement with hands-on labs; the exam tests application, not recall.

H.1.4 Microsoft Azure Solutions Architect Expert

The Azure Solutions Architect Expert (AZ-305) is the premier Azure architecture certification. It requires passing AZ-104 (Administrator) or AZ-900 plus demonstrated experience. It tests design of identity, governance, data storage, business continuity, and infrastructure solutions on Azure.

When it matters: in Microsoft-aligned organizations, financial services, and enterprise environments where Azure is the strategic cloud platform. Less universally in demand than AWS but strongly valued in its domain.

When to pursue it: after at least one year of Azure architectural work. The AZ-900 (Fundamentals) is worth doing first if you are new to Azure; it maps the service landscape clearly and is a fast study.

H.1.5 Google Cloud Professional Cloud Architect

The GCP Professional Cloud Architect tests the ability to design, develop, and manage robust, secure, scalable, highly available, and dynamic solutions on Google Cloud. It is widely regarded as one of the more technically rigorous cloud certifications.

When it matters: in data-heavy organizations, machine learning-adjacent workloads, and companies where GCP is the strategic platform. Less broadly in demand than AWS but highly valued in its specific contexts.

H.2 Security Certifications

H.2.1 CISSP (Certified Information Systems Security Professional)

The CISSP is the most widely recognized senior security certification globally, administered by (ISC)2. It covers eight domains including security and risk management, software development security, identity and access management, and security architecture and engineering. It requires five years of paid work experience in two or more domains.

When it matters: when you are working in environments with significant security governance requirements, financial services, healthcare, government, or critical infrastructure. For architects responsible for designing systems that handle regulated data or that are subject to formal security audit, the CISSP demonstrates a breadth of security knowledge that narrower credentials do not.

When to pursue it: it is a serious credential that requires significant study and verified experience. Pursue it when security architecture is a meaningful part of your role and you have the five years of experience required. It is not a certification for early-career architects.

H.2.2 CSSLP (Certified Secure Software Lifecycle Professional)

The CSSLP, also from (ISC)2, focuses specifically on security in the software development lifecycle: secure software concepts, requirements, design, implementation, testing, and supply chain and software acquisition security. It is more directly relevant to software architects than the CISSP's broader security management focus.

When it matters: in organizations that build software and are required to demonstrate that security is embedded in their development process. For architects who own the software design process and want to demonstrate that they integrate security at the design stage rather than as a post-delivery review.

H.3 Developer Certifications Worth Noting

Cloud provider developer certifications (AWS Certified Developer, Azure Developer Associate, GCP Associate Cloud Engineer) are useful stepping stones to the architect-level certifications and are worth pursuing first if you are moving from development into cloud-focused architectural work. They are faster to prepare for and provide the foundational cloud knowledge that the architect-level exams assume.

Kubernetes certifications, specifically the CKA (Certified Kubernetes Administrator) and CKAD (Certified Kubernetes Application Developer) from the Cloud Native Computing Foundation, are worth pursuing if your architectural work involves significant container orchestration responsibility. The CKA in particular is highly regarded in infrastructure and platform engineering contexts.

H.4 The Right Sequence

For a developer making the transition to architecture, the most productive sequencing is: cloud certifications first (AWS Associate or Azure Administrator, depending on your environment), followed by a cloud architect certification once you have accumulated the experience

to make the Professional-level exam meaningful, followed by a domain-specific architectural certification (TOGAF or iSAQB) if your organizational context rewards it.

Security certifications are a specialization layer. Pursue them when security architecture is a defined part of your role, not as general career advancement credentials.

The principle underlying all of this is that certifications work best when they validate and structure experience you are already developing, not when they substitute for it. The candidate who studies for the AWS Professional exam while actively architecting AWS systems learns far more and performs far better than the one who studies in the abstract. Build the experience first. Let the certification make it legible.

APPENDIX I

Companion Site Resources

All resources listed in this book are available free of charge at devtoarch.com. No registration is required. The table below lists each resource by its location in the book.

Location	Resource	URL
Introduction	All templates, tools, and resources	devtoarch.com/resources
Chapter 2	Problem-reframing worksheet	devtoarch.com/resources/problem-reframing
Chapter 5	Fallacies of distributed computing reference	devtoarch.com/resources/fallacies
Chapter 11	SLO and error budget worksheet	devtoarch.com/templates/slo
Chapter 9	STRIDE threat modeling worksheet	devtoarch.com/templates/threat-model
Chapter 14	Architectural Decision Record template	devtoarch.com/templates/adr
Chapter 18	Quality attribute prioritizer (interactive)	devtoarch.com/tools/quality-prioritizer
Chapter 18	Quality attributes worksheet	devtoarch.com/templates/quality-attributes
Chapter 19	RFC proposal template	devtoarch.com/templates/rfc
Chapter 19	Technology Radar template	devtoarch.com/templates/tech-radar
Chapter 20	Reference architecture template	devtoarch.com/templates/reference-architecture
Chapter 21	Architecture proposal template	devtoarch.com/templates/architecture-proposal
Chapter 23	Stakeholder mapping template	devtoarch.com/templates/stakeholder-map

Location	Resource	URL
Chapter 27	90-day architect plan template	devtoarch.com/templates/90-day-plan
Appendix B	ADR template (Markdown download)	devtoarch.com/templates/adr
Appendix C	Skills self-assessment (interactive)	devtoarch.com/tools/skills-assessment
Appendix D	Architecture review checklist	devtoarch.com/resources/review-checklist
Appendix E	Diagramming tools guide	devtoarch.com/resources/diagramming-tools
Appendix H	Certifications roadmap	devtoarch.com/certifications
Community	Reader community forum	devtoarch.com/community
Updates	Updates and errata	devtoarch.com/updates

Figure I.1: All companion site resources, by chapter

APPENDIX J

The Original Article Series

This book grew from a five-part article series published on the Red Hat Enable Architect platform between January and March 2023. The articles are freely available and provide a compact overview of the ideas developed at full length in these pages. Readers who want a refresher on any of the core themes, or who want to share a shorter introduction to the topic with a colleague, will find the original series useful.

All five articles are authored by Shameel Ahmed Z and are available at no cost without registration. The author profile page, which links to all five articles, is at:

redhat.com/en/authors/shameel-ahmed

Part 1 3 lessons for software developers pivoting into IT architecture

Red Hat Enable Architect • January 9, 2023

The mindset shift that makes or breaks the transition. Covers the breadth of the architect role, the landscape of architect specializations, and why resisting the urge to start coding is the first discipline to develop.

www.redhat.com/en/blog/it-career-developer-architect-lessons

Part 2 3 soft skills aspiring IT architects need to develop

Red Hat Enable Architect • January 16, 2023

Why technical skill alone is not enough. Covers understanding the business domain, managing stakeholder expectations, and the shift from internal development deadlines to external business commitments with real consequences.

www.redhat.com/en/blog/it-career-developer-architect-soft-skills

Part 3 5 technical skills aspiring IT architects need to learn

Red Hat Enable Architect • January 24, 2023

How to grow from depth in one stack to useful breadth across many. Covers the hands-on vs. hands-off decision, architecture strategies and frameworks, managing technical debt deliberately, and understanding architecture tradeoffs.

www.redhat.com/en/blog/architect-technical-skills

Part 4 4 people skills aspiring IT architects need to master

Red Hat Enable Architect • February 21, 2023

The human side of the role. Covers communicating architecture effectively to different audiences, tailoring views for engineers vs. executives, listening and collaborating on design, and negotiating architectural decisions with stakeholders.

www.redhat.com/en/blog/people-skills-architects

Part 5 5 ways helping the community can boost your IT architect career

Red Hat Enable Architect • March 27, 2023

How contributing to the wider community accelerates your own development. Covers open source contributions, maintaining a professional presence, mentoring junior engineers, writing books and blogs, and creating training courses.

www.redhat.com/en/blog/it-career-community-contribution

About the Author

Shameel Ahmed Z is a software architect whose career began writing desktop applications in Visual Basic, building ActiveX controls and component libraries before the web took over. He moved to .NET and ASP.NET, progressed through enterprise content management, case management, and enterprise telephony platforms, and spent a significant portion of his career working across a wide stack: WinForms clients, .NET and Java services, TSAPI integrations, and a large-scale data warehouse and reporting platform built on ASP.NET and SQL Server, culminating in a migration from multiple on-premises SQL Server instances to PostgreSQL on AWS Aurora. He has held principal and staff architect roles across multiple organizations, working primarily for US companies and a few based in Europe.

Over his career he has designed systems handling millions of daily transactions, led platform migrations from legacy monoliths to cloud-native architectures, and built engineering organizations from the ground up. A recurring theme in his work is developer productivity and reusability: building platforms, pipelines, and shared patterns that reduce the friction teams face when delivering software, and automating the repetitive work that accumulates in codebases and deployment pipelines when no one has stepped back to design for repeatability.

He writes and speaks on the craft of architecture, the developer-to-architect transition, and the organizational and human dimensions of engineering leadership. His particular interest is in the gap between how architecture is taught and how it is practiced: what the books say, what the job actually requires, and how to close the distance between them.

He can be reached at shameel@shameel.net.

Get the Full Book

You have read 6 of the 28 chapters.
The full book is available on LeanPub in PDF and ePub formats.

leanpub.com/devtoarch

Free templates and resources at devtoarch.com

ISBN 978-93-6068-417-4 | Copyright 2026 Shameel Ahmed Z. All rights reserved.