Chapter 1. Introduction

So you're interested in software architecture. Perhaps you're a developer who wants to move to the next career step, or perhaps you are a project manager who wants to understand what happens when software architectures work. You may also be an "accidental architect": someone who makes architecture decisions (defined below) but doesn't have the title of "software architect"…yet.

Why delve into the realm of software architecture? Perhaps you have experience with lots of projects and want to understand more deeply how the larger parts of systems fit together, along with the numerous trade-offs. If so, software architecture is an obvious next career step.

This book is designed for all of you. It provides an overview of the extremely multifaceted job of "software architect."

Software architects must understand and analyze software systems deeply, in all their complexity, and must make important trade-off decisions, sometimes with incomplete information. Many software developers worried that generative AI might slowly replace them are considering moving to software architecture, a role much harder to replace. Software architects make exactly the kinds of decisions that AI cannot, evaluating trade-offs within complex, changing contexts.

Architecture, like much art, can only be understood in context. Architects base their decisions on the realities of their environment. For example, one of the major goals of late-20th-century software architecture was to use shared infrastructure and resources as efficiently as possible, because operating systems, application servers, database servers, and so on were all commercial and very expensive.

In 2002, trying to build an architecture like microservices would have been inconceivably expensive. Imagine strolling into a 2002 data center and telling the head of operations, "Hey, I have a great idea for a revolutionary style of architecture, where each service runs on its own isolated machinery with its own dedicated database. I'll need 50 Windows licenses, another 30 application-server licenses, and at least 50 database server licenses." We can only build such architectures today because of the advent of open source and the updated engineering practices of the DevOps revolution. All architectures are products of their context—keep that in mind as you read this book.

Defining Software Architecture

So what is software architecture? Figure 1-1 illustrates how we like to think about software architecture. This definition has four dimensions. The software architecture of a system consists of an *architecture style* as the starting point, combined with the *architecture characteristics* it must support, the *logical components* to implement its behavior, and finally the *architecture decisions* justifying it all. The system's structure is denoted by the heavy black lines supporting the architecture. We'll walk quickly through these dimensions in the order in which architects analyze them, and subsequent chapters will provide more details.

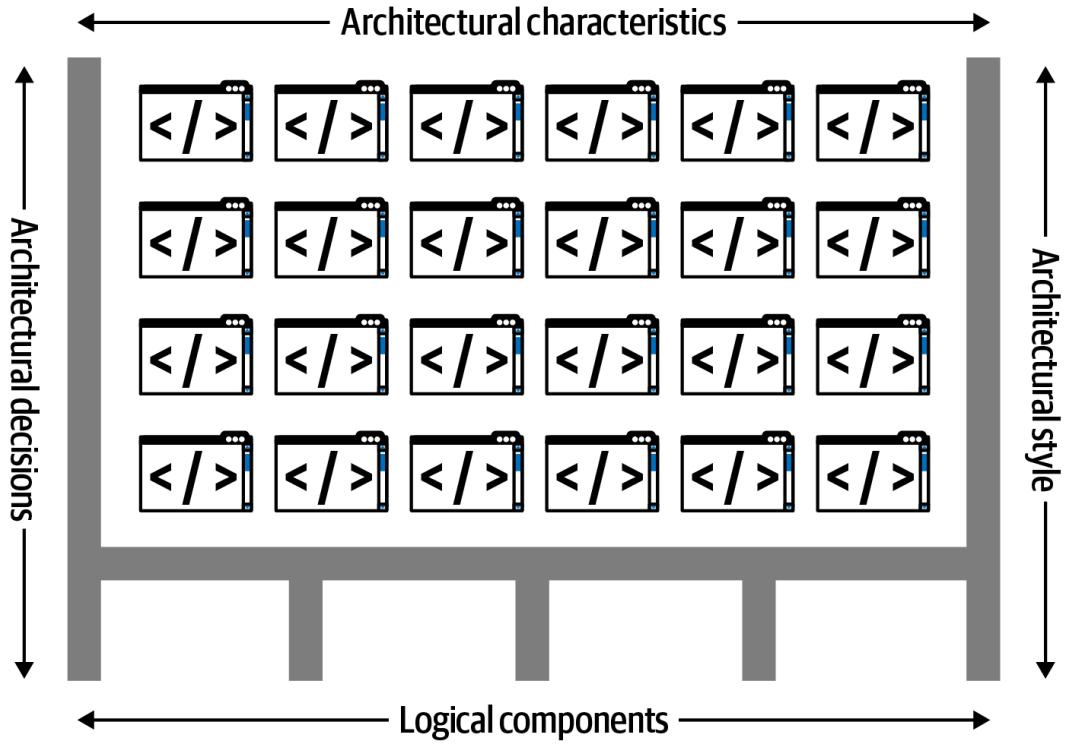

Figure 1-1. Architecture consists of the system's structure, combined with architecture characteristics ("-ilities"), logical components, architecture styles, and decisions

Architecture characteristics (see Figure 1-2) define the *capabilities* of a system (commonly abbreviated as "-ilities") and the criteria for its success: in short, what the system should *do*. Architecture characteristics are so important that we've devoted several chapters in this book to understanding and defining them.

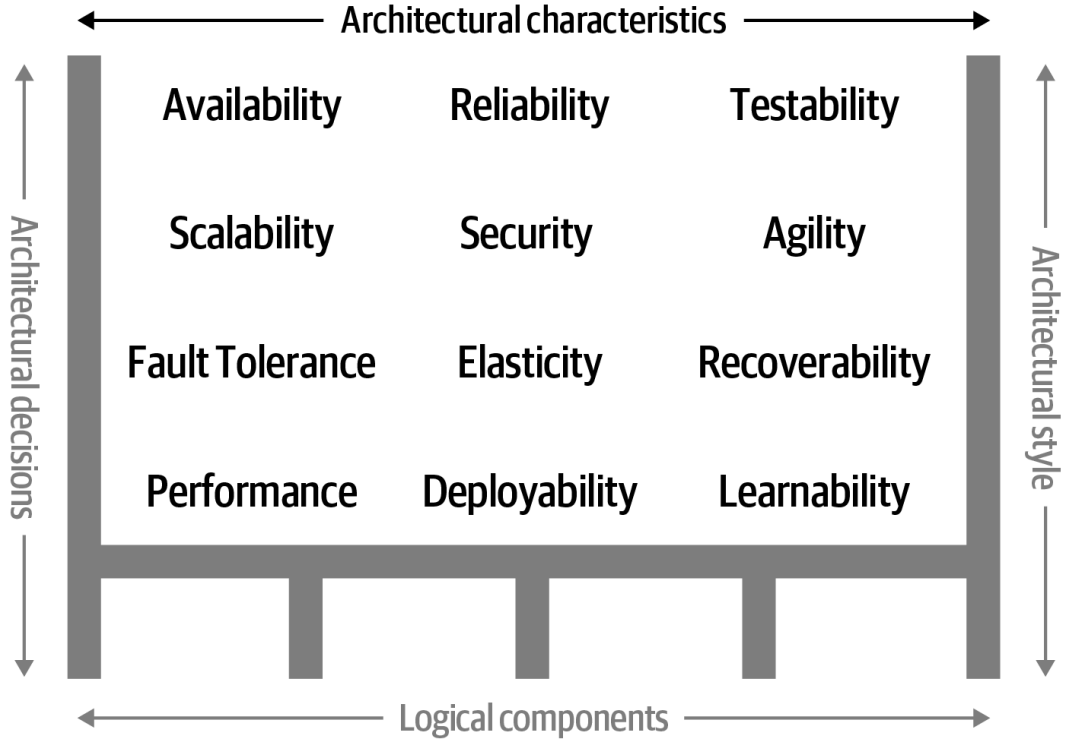

Figure 1-2. "Architecture characteristics" refer to the "-ilities" that the system must support

While architectural characteristics define a system's capabilities, *logical components* define its *behavior*. Designing logical components is one of the key structural activities for architects. In Figure 1-3, the logical components form the domains, entities, and workflows of the application.

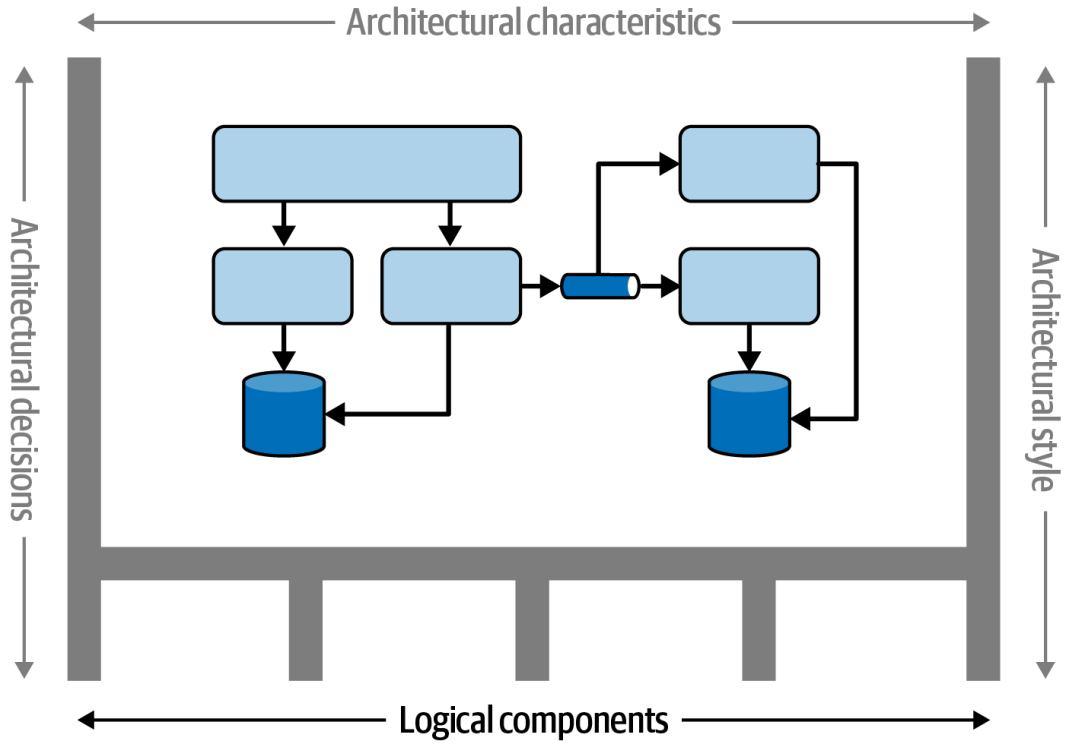

Figure 1-3. Logical components structure the behavior of the system

Once an architect has analyzed the architectural characteristics and logical components the system needs (both described in detail later), they know enough to choose an appropriate architecture style as a starting point for implementing their solution Figure 1-4.

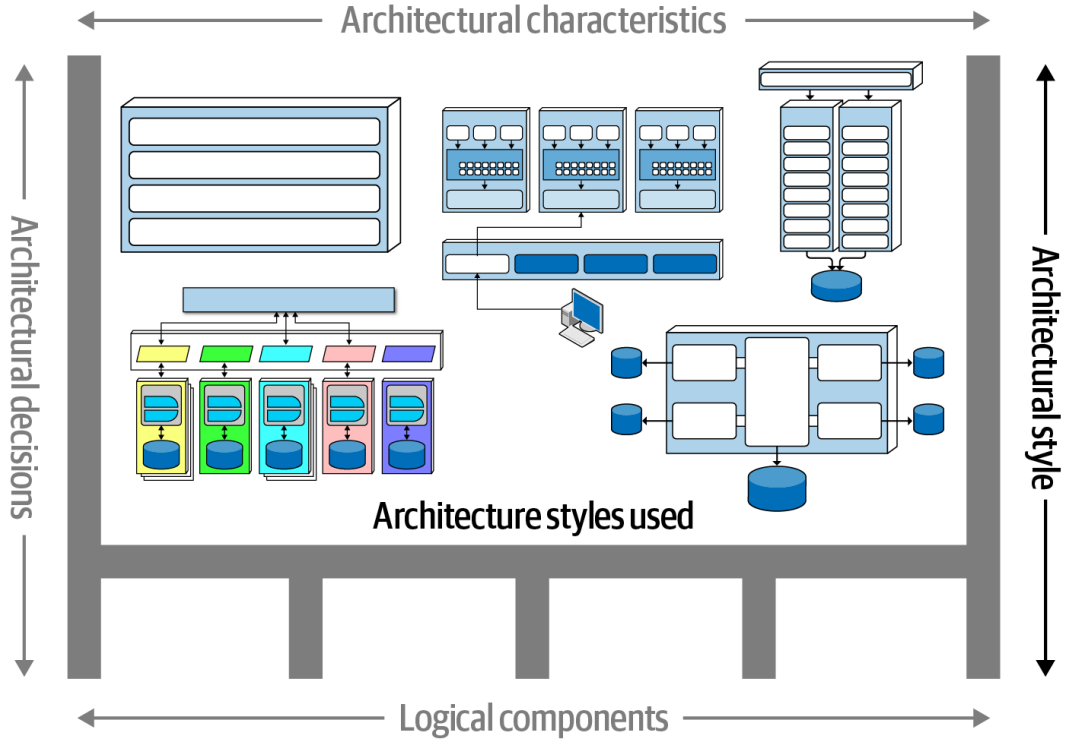

Figure 1-4. Choosing an architectural style involves finding the easiest implementation path for a given set of requirements

The fourth dimension that defines software architecture is *architecture decisions*, which define the rules for how a system should be constructed. For example, an architect might make a decision that only the Business and Services layers within a layered architecture can access the database (see Figure 1-5), restricting the

Presentation layer from making direct database calls. Architecture decisions form the constraints of the system and direct the development teams on what is and what isn't allowed.

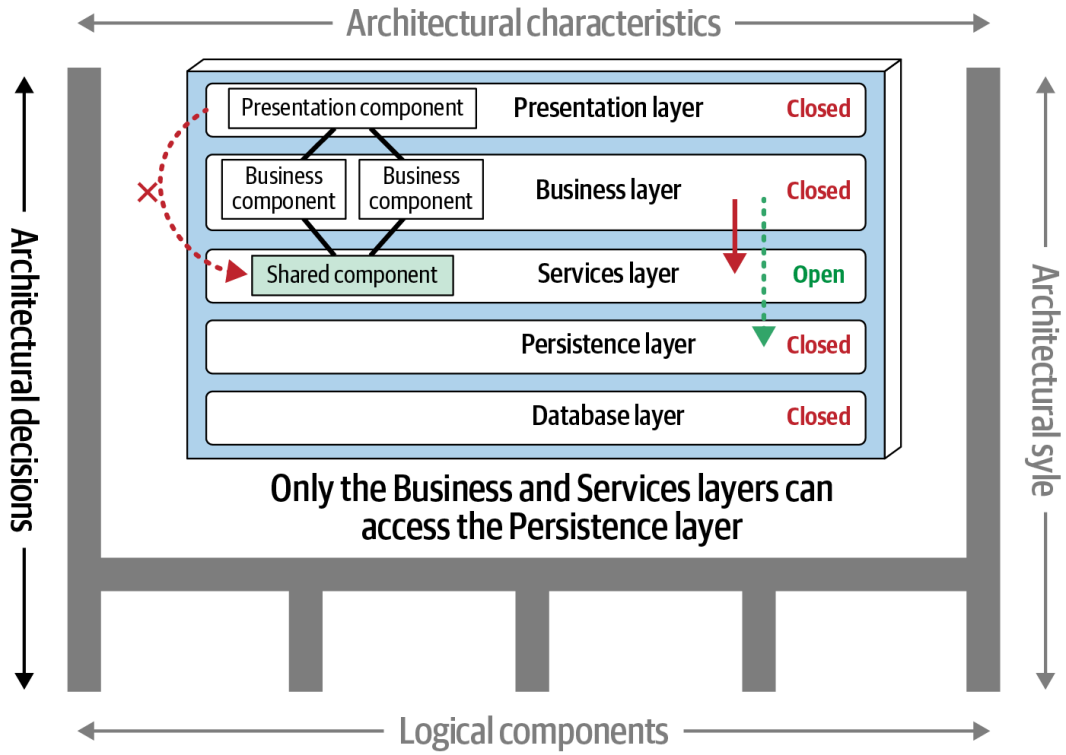

Figure 1-5. Architecture decisions are rules for constructing systems

We discuss architecture decisions and how to document them concisely in Chapter 21.

Laws of Software Architecture

As your two authors set out to write the first edition of this book, we had an ambitious goal: we hoped to find things that seemed universally true about software architecture and codify them as "laws" of software architecture. As we wrote, we kept our eyes peeled for things we could capture; we had hoped to find maybe 10 or 15. To our surprise, we ended up identifying just two laws in the first edition, and uncovered one more while writing the second edition. True to our original intent, these three laws seem pretty universal and inform many important perspectives for working software architects.

We learned the First Law of Software Architecture by constantly stumbling across it, and we think it gets to the heart of why these universal truths seem so elusive:

> Everything in software architecture is a trade-off.
>
> First Law of Software Architecture

Nothing exists on a nice, clean spectrum. Every decision a software architect makes must account for a large number of variables that take on different values depending on the circumstances. Trade-offs are the essence of software architecture decisions.

> If you think you've discovered something that *isn't* a trade-off, more likely you just haven't *identified* the trade-off…yet.
>
> Corollary 1
>
> You can't just do trade-off analysis once and be done with it.
>
> Corollary 2

Teams love standards, and it would be nice if architects could just do One Big Trade-off Jamboree to decide on our defaults for which architecture styles to use, how parts of the architecture should communicate, how to manage shared functionality, and a host of other sticky decisions. But we can't, because every situation requires us to re-evaluate all those trade-offs. (We've seen teams try to do this—for example, attempting to default to using only choreography in distributed workflows, only to discover that it works sometimes and is a spectacular disaster at other times. See "Choreography and Orchestration" for our discussion.)

Architecture is broader than just the combination of structural elements, which is why our dimensional definition incorporates principles, characteristics, and so on. This is reflected in our Second Law of Software Architecture:

> *Why* is more important than *how*.
>
> Second Law of Software Architecture

As an experienced architect, if someone shows me an architecture I've never seen before, I can understand *how* it works, but I might struggle with *why* the previous architect or team made certain decisions. Architects make decisions within extremely specific contexts, making sweeping and generic decisions difficult. *Why* an architect made a particular decision includes the trade-offs they considered, adding to the context of why this decision versus another. It turns out that one of the defining characteristics of software architecture decisions is they are rarely binary. This brings us to the Third Law of Software Architecture:

> Most architecture decisions aren't binary but rather exist on a spectrum between extremes.
>
> Third Law of Software Architecture

Throughout the book, we highlight *why* architects make certain decisions along with trade-offs. We also highlight good techniques for capturing important decisions in Chapter 21.

Readers will recognize these laws at work throughout the book, and we recommend keeping them in mind when evaluating software architecture decisions. We come back to these laws in Chapter 27 with some additional examples.

With a working definition of software architecture in place, let's move into the role itself.

Expectations of an Architect

The role of a software architect can range from acting as an expert programmer to defining an entire company's strategic technical direction. This makes trying to define it a fool's errand, but what we can tell you more readily is what's *expected*

of a software architect. We've identified eight core expectations for any software architect, irrespective of their role, title, or job description:

- Make architecture decisions
- Continually analyze the architecture
- Keep current with latest trends
- Ensure compliance with decisions
- Understand diverse technologies, frameworks, platforms, and environments
- Know the business domain
- Lead a team and possess interpersonal skills
- Understand and navigate organizational politics

Succeeding as a software architect depends on understanding and living up to each of these expectations. This section looks at all eight in turn.

Make Architecture Decisions

An architect is expected to define the architecture decisions and design principles used to guide technology decisions within the team, within the department, or across the enterprise.

Guide is the key word in this first expectation: architects should *guide* technology choices rather than *specify* them. For example, deciding to use React.js for frontend development is a technical decision rather than an architectural decision. Instead of making this decision, the architect should instruct the development teams to use a reactive-based framework for frontend web development, guiding them to choose Angular, Elm, React.js, Vue, or any of the other reactive-based web frameworks. The key is asking whether the architecture decision will *guide* teams in making the right technical choices or make the choice for them. That said, there are some occasions where architects need to make specific technology decisions in order to preserve a particular architectural characteristic, such as scalability, performance, or availability. Architects often struggle with finding this balance, so Chapter 21 is entirely about architecture decisions.

Continually Analyze the Architecture

An architect is expected to continually analyze the architecture and the current technology environment and recommend solutions for improvement.

Architecture vitality assesses how viable an architecture that was defined three or more years ago is *today*, given changes in both business and technology. In our experience, not enough architects focus their energies on continually analyzing existing architectures. As a result, most architectures experience elements of structural decay, which occurs when developers make coding or design changes that impact the required architectural characteristics, such as performance, availability, and scalability.

Other frequently forgotten aspects of this expectation are testing and release environments. Being able to modify code quickly is a kind of agility with obvious benefits, but if it takes weeks to test the changes and months to release, the overall architecture cannot achieve agility.

Architects must holistically analyze changes in the technology and the problem domain to determine the ongoing soundness of the architecture. This kind of consideration is what keeps applications relevant, even if it rarely appears in job postings.

Keep Current with Latest Trends

An architect is expected to keep current with the latest technology and industry trends.

Developers must keep up to date on the latest technologies, as well as those they use on a daily basis, to remain relevant (and to retain a job!). It's even more critical for architects to keep current on the latest technical and industry trends. Architects' decisions tend to be long-lasting and difficult to change. Understanding trends helps architects make decisions that will remain relevant into the future. For example, architects in recent years have needed to learn about cloud-based storage and deployment, and as we write this second edition, generative AI is having a massive impact on many parts of the development ecosystem.

Tracking trends and keeping current with those trends is hard, particularly for a software architect. In Chapter 2, we discuss some techniques and resources for how to do this.

Ensure Compliance with Decisions

An architect is expected to ensure compliance with architecture decisions and design principles.

Ensuring compliance means continually verifying that the development teams are following the decisions and design principles defined, documented, and communicated by the architect.

Imagine that you, as an architect, make a decision to restrict access to the database in a layered architecture to only the Business and Services layers (not the Presentation layer). This (as you'll see in Chapter 10) means that the Presentation layer must go through all layers of the architecture to make even the simplest of database calls. However, a user interface (UI) developer disagrees with this decision and gives the Presentation layer direct access to the database for performance reasons. You made that architecture decision for a specific reason: so that changes to the database changes would not affect the Presentation layer. If you aren't ensuring compliance with your architecture decisions, violations like this can occur. The result can be that the architecture fails to provide the required characteristics, and the application or system does not work as expected. In Chapter 6, we talk about measuring compliance using automated fitness functions and other tools.

Understand Diverse Technologies

An architect is expected to have exposure to multiple and diverse technologies, frameworks, platforms, and environments.

Every architect isn't expected to be an expert in every framework, platform, and language, but they should at least be familiar with a variety of technologies. Most environments these days are heterogeneous, so at a minimum, architects should

know how to interface with multiple systems and services, whatever their language, platform, or technology.

One of the best ways to meet this expectation is to stretch outside your comfort zone, aggressively seeking out opportunities to gain experience in multiple languages, platforms, and technologies. Architects should focus on technical *breadth* rather than technical depth. Technical breadth includes the stuff you know about, but not at a detailed level, combined with the stuff you know a lot about. For example, it's far more valuable for an architect to be familiar with the pros and cons of 10 different caching products than to be an expert in only one of them.

Know the Business Domain

An architect is expected to have a certain level of business-domain expertise.

Effective software architects understand the business problem, goals, and requirements the architecture is intended to solve, collectively known as the *business domain* of a problem space. It's difficult to design an effective architecture if you don't understand the requirements of the business. Imagine being an architect at a major bank and not understanding common financial terms like *average directional index*, *aleatory contracts*, *rates rally*, or even *nonpriority debt*. You wouldn't be able to communicate with stakeholders and business users and would quickly lose credibility.

The most successful architects we know have broad, hands-on technical knowledge *and* a strong knowledge of a particular domain. They can communicate with C-level executives and business users in language these stakeholders know and understand, creating confidence that they know what they're doing and are competent to create an effective and correct architecture.

Possess Interpersonal Skills

An architect is expected to possess exceptional interpersonal skills, including teamwork, facilitation, and leadership.

As technologists, developers and architects tend to prefer solving technical problems, not people problems, so exceptional leadership and interpersonal skills are a difficult expectation. However, as Gerald Weinberg was famous for saying, "No matter what they tell you, it's always a people problem." The guidance that architects provide is not only technical—it includes leading the development teams through implementing the architecture. Leadership skills are *at least half* of what it takes to become an effective software architect, regardless of role or title.

The industry is flooded with software architects, all competing for a limited number of positions. Those with strong leadership and interpersonal skills stand out from the crowd. Conversely, we've known many software architects who are excellent technologists but struggle to lead teams, coach and mentor developers, or communicate ideas and architecture decisions and principles. Needless to say, those architects have difficulties holding down a job.

Understand and Navigate Politics

An architect is expected to understand the political climate of the enterprise and be able to navigate its politics.

It might seem strange to talk about office politics in a book about software architecture, but negotiation skills are crucial to the role. To illustrate, consider two scenarios.

In the first scenario, a developer decides to leverage a particular design pattern to reduce the complexity of a particular piece of convoluted code. That's a fine decision, and the developer does not need to seek approval for it. Programming aspects such as code structure, class design, design-pattern selection, and sometimes even language choice are all part of the art of programming.

In the second scenario, the architect responsible for a large customer relationship management (CRM) system is having difficulty controlling database access from other systems, securing certain customer data, and making changes to the database schema. All of these problems stem from too many other systems using the CRM database. The architect therefore decides to create *application silos*, where each application database is only accessible from the application that owns that database. This decision will give the architect better control over the customer data, security, and changes.

However, unlike the developer's decision in the first scenario, the architect can expect almost everyone in the company to challenge their decision (with the possible exception of the CRM application team). Other applications need the CRM data, and if they can no longer access the database directly, they must ask the CRM system for the data via remote access calls. Product owners, project managers, and business stakeholders may object to increases in their costs or effort, while developers may feel their approach is better. *Almost every decision an architect makes will be challenged*.

Whatever the objections, the architect must navigate organizational politics and apply negotiation skills to get most decisions approved. This can be very frustrating; most software architects started as developers and got used to making decisions without approval or even review. As architects, they're now finally able to make broad and important decisions, but they must justify and fight for almost every one. Negotiation skills, like leadership skills, are so necessary that we've dedicated an entire chapter to them (Chapter 25).

Roadmap

This book consists of three parts:

Part I: Foundations

> Part I defines the key components of software architecture, focusing on the two key elements of architectural structure: architectural characteristics and logical components. Analyzing each of these requires different techniques, which we cover in depth. The outputs of these activities provide the software architect with enough information to choose an appropriate architectural style to provide a scaffolding for the general philosophy of the implementation.

Part II: Architecture Styles

> Part II provides a catalog of *architectural styles*, the named topologies of software architecture. We show the structural and communication differences in each style, and provide a basis for comparison along a wide spectrum, including data topologies, teams, and physical architectures.

Part III: Techniques and Soft Skills

Parts I and II focus on technical aspects of the job, but a major part of the role of software architect involves what are traditionally called *soft skills*: skills that involve other people instead of technology. Ironically, soft skills are often the most difficult for burgeoning architects to acquire, because the main pathways to the software architect role typically involve more technical than interpersonal brilliance. However, once on the job, architects discover that these skills are vitally important. Thus, Part III of our book covers some critical soft skills to help you succeed in the role.