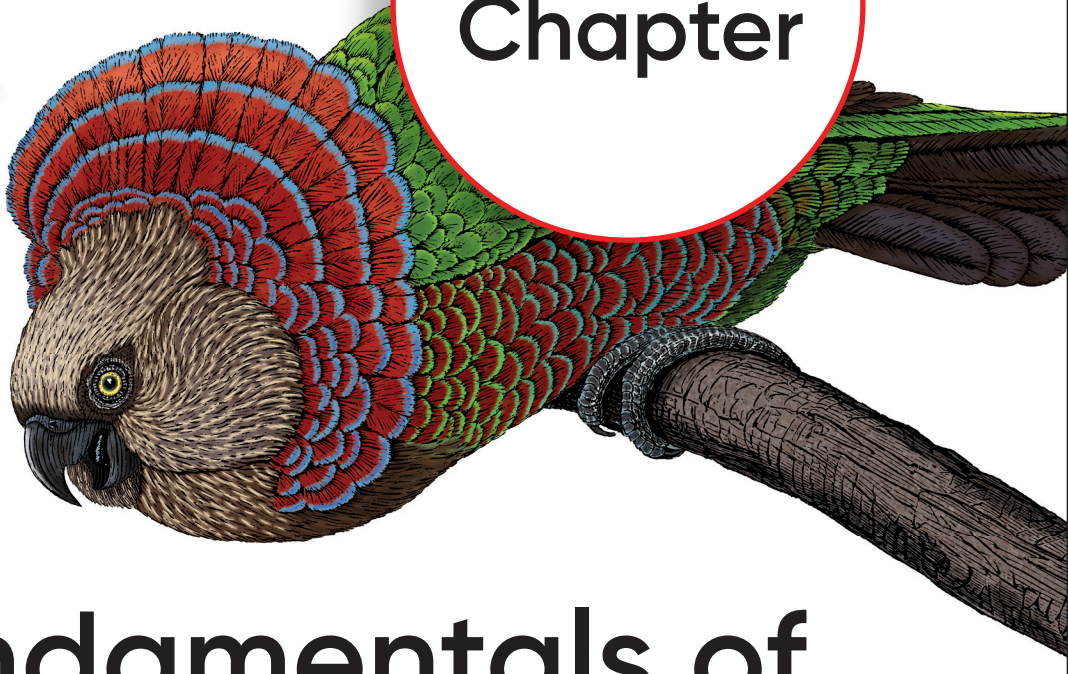
O'REILLY®

Free Chapter



Fundamentals of Software Architecture

An Engineering Approach

Mark Richards & Neal Ford

Fundamentals of Software Architecture

An Engineering Approach

This excerpt contains Chapter 1 of the book *Fundamentals of Software Architecture*. The complete book will be available on the O'Reilly Online Learning Platform and through other retailers in February 2020.

Mark Richards and Neal Ford

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Fundamentals of Software Architecture
by Mark Richards and Neal Ford

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Chris Guzikowski
Development Editors: Alicia Young and Virginia Wilson
Production Editor: Christopher Faucher
Copyeditor: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

February 2020: First Edition

Revision History for the First Edition
2020-01-22: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492043454* for release details.

978-1-492-04345-4

Table of Contents

1. Introduction. 1
Defining Software Architecture 3
Expectations of an Architect 7
Make Architecture Decisions 7
Continually Analyze the Architecture 8
Keep Current With Latest Trends 8
Ensure Compliance With Decisions 9
Diverse Exposure and Experience 9
Have Business Domain Knowledge 10
Possess Interpersonal Skills 10
Understand and Navigate Politics 11
Intersection of Architecture and … 12
Engineering Practices 13
Operations/DevOps 16
Process 17
Data 18
Laws of Software Architecture 18

CHAPTER 1
Introduction

The job *software architect* appears near the top of numerous "best jobs" lists across the world. Yet, when readers look at the *other* jobs on those lists (like nurse practitioner or finance manager), a clear career path exists from not being one of those things to becoming one. Why is this path absent for software architects?

First, the industry doesn't have a good definition of software architecture itself. When we teach foundational classes, we have often been asked for a concise, succinct definition of what a software architect does, and we have adamantly refused. And we're not the only ones. In his famous white paper Who Needs an Architect?, Martin Fowler famously refused to try to define it, instead falling back on the famous quote:

> Architecture is about the important stuff…whatever that is.
>
> —Ralph Johnson

When pressed, we created the mindmap shown Figure 1-1, which is woefully incomplete but indicative of the scope of software architecture.

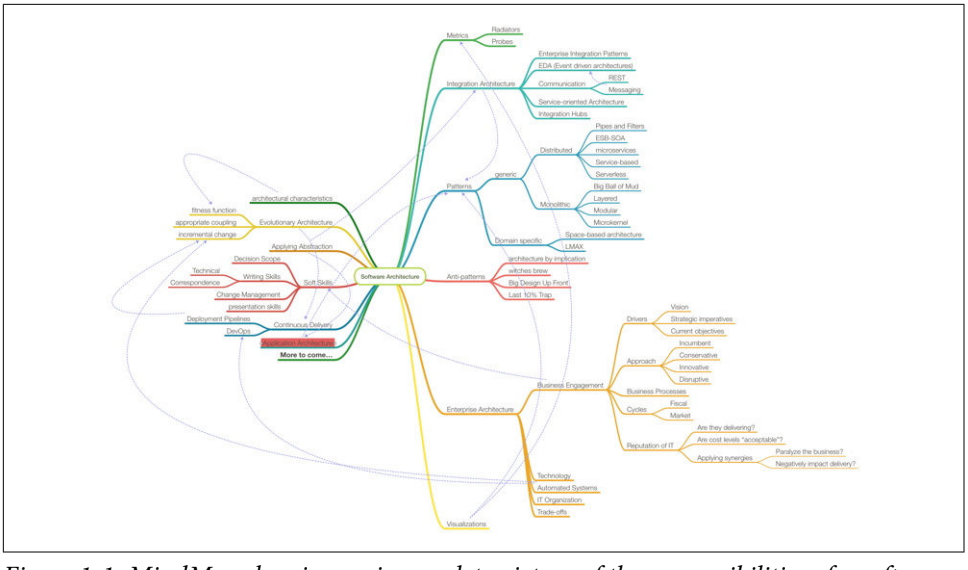


Figure 1-1. MindMap showing an incomplete picture of the responsibilities of a software architect.

We will in fact offer our definition of software architecture shortly.

Second, as illustrated by Figure 1-1, the role of software architect embodies a massive amount and scope of responsibility that continues to expand. A decade ago, software architects dealt only with the purely technical aspects of architecture, like modularity, components, and patterns. However, because of new architectural styles that leverage a wider swath of capabilities (like microservices), the role of software architect has expanded. We cover the many intersections of architecture and the remainder of the organization in the upcoming section "Intersection of Architecture and …" on page 12.

Third, software architecture is a constantly moving target because of the rapidly evolving software development ecosystem. Any definition cast today will be hopelessly outdated in a few years. The WikiPedia definition of software architecture provides a reasonable overview, but many statements are outdated (such as "Software architecture is about making fundamental structural choices which are costly to change once implemented." Yet, architects designed modern architectural styles like microservices with the idea of incremental built in—it is no longer expensive to make structural changes in microservices. Of course, that capability means *tradeoffs* with other concerns, such as coupling. Many books on software architecture treat it as a static problem; once solved, we can safely ignore it. However, we recognize the inherent dynamic nature of software architecture, including the definition itself, throughout the book.

Fourth, much of the material extant about software architecture has only historical relevance. Readers of the Wikipedia page won't fail to notice the bewildering array of acronyms and cross references to an entire universe of knowledge, with the typical WikiPedia lack of consistent level of exposition. Yet, many of these acronyms represent outdated or failed attempts. Even solutions that were perfectly valid a few years ago cannot work not because the context has changed. The history of software architecture is littered with things architects tried, only to realize damaging side effects. We cover many of those lessons in this book.

Why a book on software architecture fundamentals now? The scope of software architecture isn't the only part of the development world that constantly changes. New technologies, techniques, capabilities…in fact, it's easier to find things that haven't changed over the last decade than list all the changes. Software architects must make decisions against this constantly changing ecosystem. Because everything changes, including foundations upon which we make decisions, architects should reexamine some core axioms that informed earlier writing about software architecture. For example, earlier books about software architecture don't consider the impact of DevOps because it didn't exist when the book was written.

When studying architecture, readers must keep in mind that, like much art, it can only be understood in context. Many of the decisions architects made were based on realities of the environment they found themselves in. For example, one of the major goals of late 20th century architecture included making most efficient use of shared resources, because all the infrastructure at the time was expensive and commercial: operating systems application servers, database servers, etc. Imagine strolling into a 2002 data center and telling the head of operations "Hey, I have a great idea for a revolutionary style of architect, where each service runs on its own isolated machinery, with its own dedicated database (describing what we now know as microservices). So, that means I'll need 50 licenses for Windows, another 30 application server licenses, and at least 50 database server licenses." In 2002, trying to build an architecture like microservices would be inconceivably expensive. Yet, with the advent of open source during the intervening years, coupled with updated engineering practices via the DevOps revolution, we can reasonably build an architecture as described. Readers should keep in mind that all architectures are a product of their context.

Defining Software Architecture

The industry as a whole has struggled to precisely define "software architecture". Some architects refer to software architecture as the *blueprint* of the system, while others define it as the *roadmap* for developing a system. The issue with these common definitions is understanding what the blueprint or roadmap actually contains. For example, what is analyzed when an architect *analyzes* an architecture?

Figure 1-2 illustrates a way to think about software architecture. In this definition, software architecture consists of the *structure* of the system (denoted as the heavy black lines supporting the architecture), combined with *architecture characteristics* ("-ilities") the system must support, *architecture decisions*, and finally *design principles*.

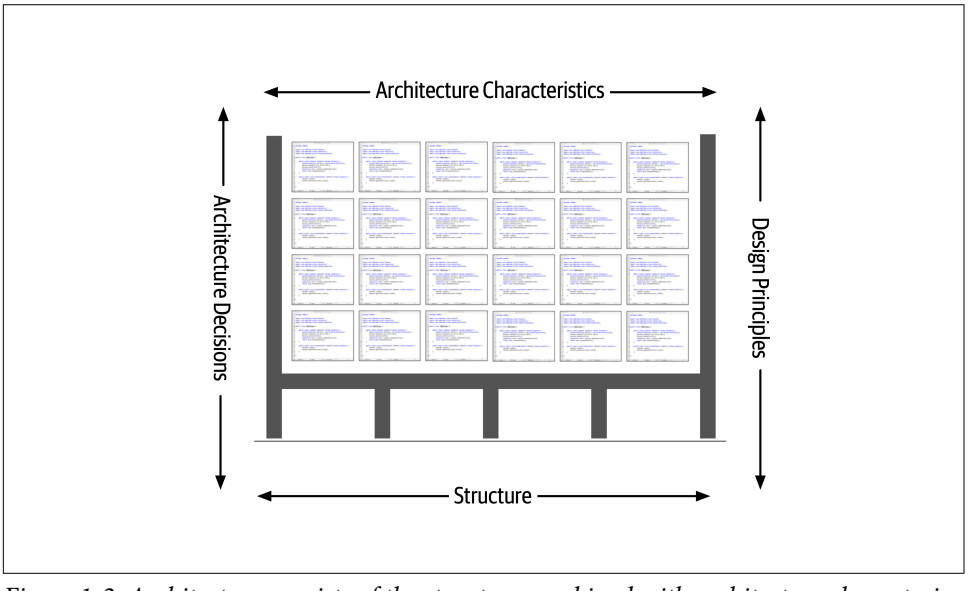


Figure 1-2. Architecture consists of the structure combined with architecture characteristics ("-ilities"), architecture decisions, and design principles.

The *structure* of the system, as illustrated in Figure 1-3, refers to the type of architecture style (or styles) the system is implement in (such as microservices, layered, microkernel, and so on). Describing an architecture solely by the structure does not wholly elucidate an architecture. For example, suppose an architect is asked to describe an architecture and that architect responds "it's a microservices architecture". Here, the architect is only talking about the *structure* of the system, but not the *architecture* of the system. Knowledge of the architecture characteristics, architecture decisions, and design principles are also needed to fully understand the architecture of the system.

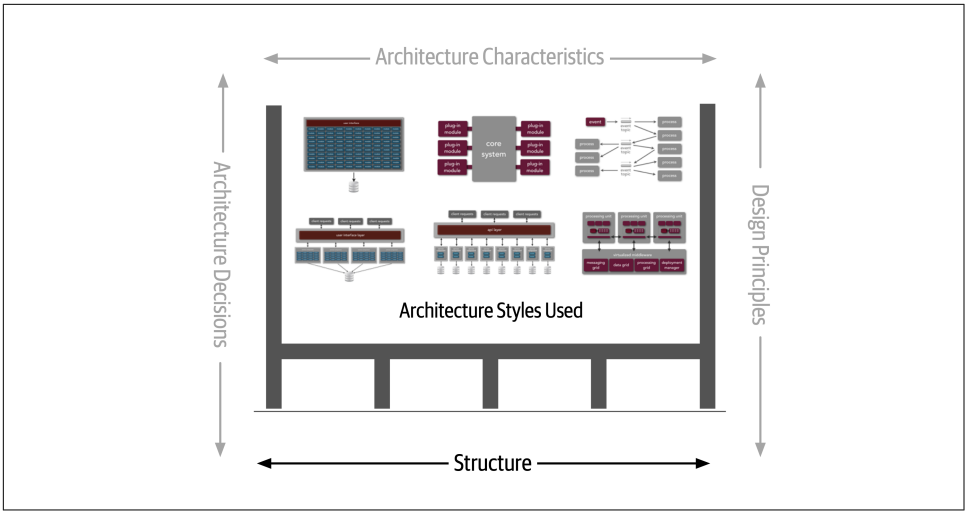


Figure 1-3. Structure refers to the type of architecture styles used in the system.

Architecture characteristics is another dimension of defining software architecture (see Figure 1-4). The architecture characteristics define the success criteria of a system, which is generally orthogonal to the functionality of the system. Notice in Figure 1-4 that all of the characteristics listed do not require knowledge of the functionality of the system, yet they are required in order for the system to function properly. Architecture characteristics are so important that we've devoted several chapters in this book to understanding and defining them.

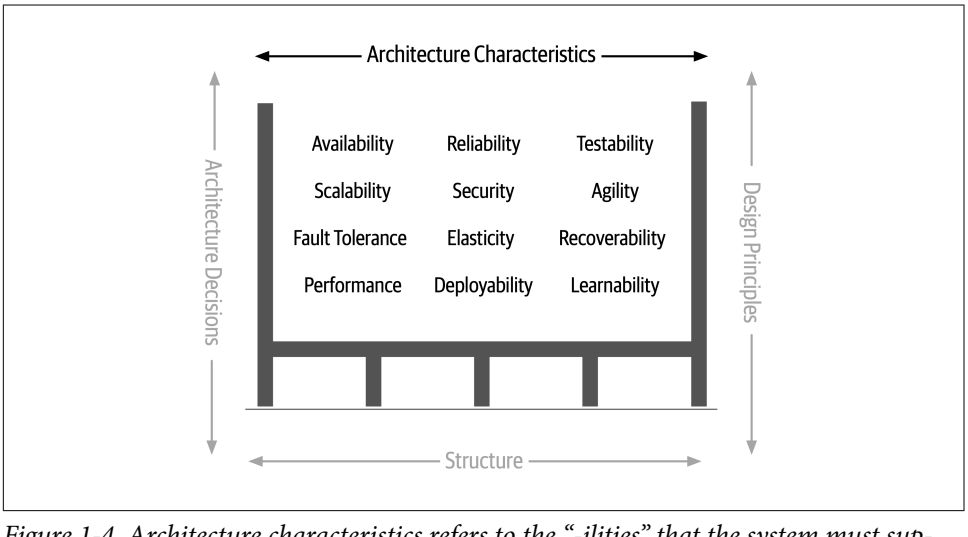


Figure 1-4. Architecture characteristics refers to the "-ilities" that the system must support

The next factor that defines software architecture is architecture decisions. Architecture decisions define the "rules" for how a system should be constructed. For example, an architect might make an architecture decision that only the business and services layer within a layered architecture may access the database (see Figure 1-5), therefore 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.

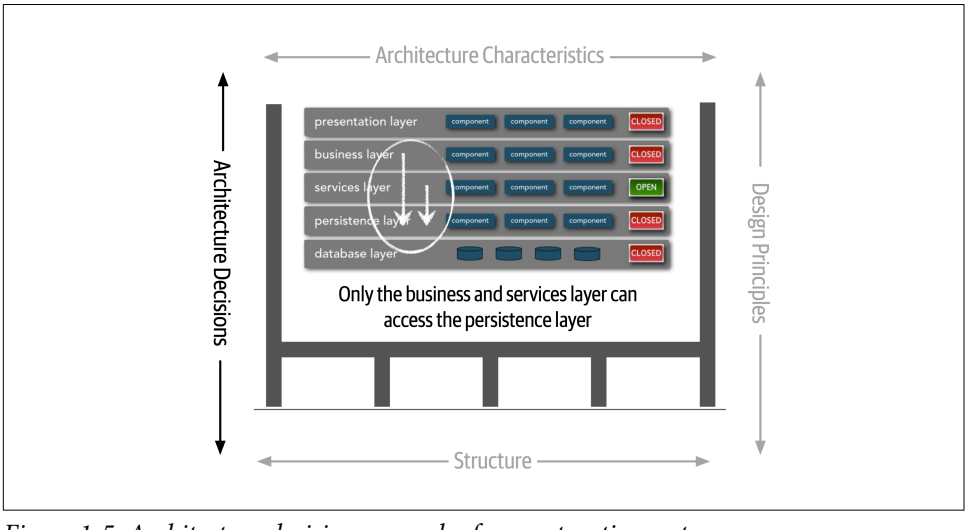


Figure 1-5. Architecture decisions are rules for constructing systems.

If a particular architecture decision cannot be implemented in one part of the system due to some condition or other constraint, that decision (or rule) can be "broken" through something called a *variance*. Most organizations have variance models that are used by an architecture review board (ARB) or chief architect that formalize the process for seeking a variance to a particular standard or architecture decision. An exception to a particular architecture decision is analyzed by the ARB (or chief architect if no ARB exists) and is either approved or denied based on justifications and tradeoffs.

The last factor in the definition of architecture is *design principles*. A design principle differs from an architecture decision in that a design principle is a *guideline* rather than a hard-and-fast *rule*. For example, the design principle as illustrated in Figure 1-6 states that the development teams should leverage asynchronous messaging between services within a microservices architecture to increase performance. An architecture decision (rule) could never cover every condition and option for communication between services in an architecture decision, so a design principle can be used to provide guidance for the preferred method (in this case asynchronous mes-

saging) to allow the developer to choose a more appropriate communication protocol (such as REST or gRPC) given a specific circumstance.

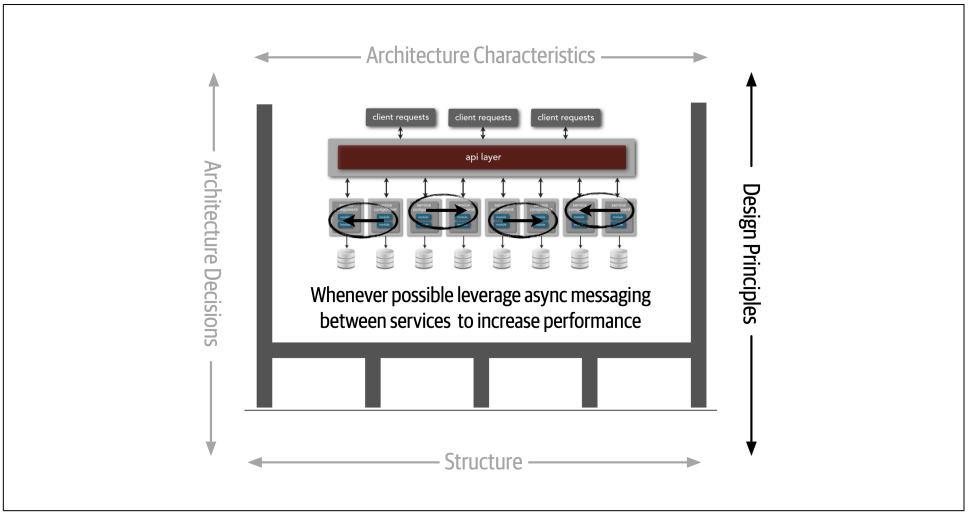


Figure 1-6. Design principles are guidelines for constructing systems.

Expectations of an Architect

Defining the role of a software architect presents as much difficultly as defining software architecture. It can range from expert programmer up to defining the strategic technical direction for the company. Rather than waste time on the fool's errand of defining the role of a software architect, we recommend focusing on the *expectations* of an architect.

There are eight core expectations placed on a software architect, irrespective of any given role, title, or job description (e.g., application architect, solution architect, network architect, data architect, integration architect, and so on). The first key to effectiveness and success in the software architect role depends on understanding and practicing each of these expectations.

Make Architecture Decisions

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

Guide is the key operative word in this first expectation. An architect should *guide* rather than *specify* technology choices. For example, an architect might make a decision to use React.js for front-end development. In this case, the architect is making a technical decision rather than an architectural decision or design principle that will help guide development teams choices. An architect should instead instruct develop-

ment teams to use a *reactive-based framework for front-end web development*, hence guiding the development team in making the choice between Angular, Elm, React.js, Vue, or any of the other host of reactive-based web frameworks.

Guiding technology choices through architecture decisions and design principles is difficult. The key to making effective architectural decisions is asking whether the architecture decision is helping to *guide* teams in making the right technical choice or whether the architecture decision *makes* the technical choice for them. That said, an architect on occasion might need to make specific technology decisions in order to preserve a particular architectural characteristic such as scalability, performance, availability, and so on. In this case it would be still considered an architectural decision, even though it specifies a particular technology. Architects often struggle finding the correct line, so we devote an entire chapter to architecture decisions (see Chapter 19).

Continually Analyze the Architecture

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

This expectation of an architect refers to *architecture vitality*, which assesses how viable the 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, scalability, and so on.

Another easily forgotten aspect of the analysis expectation is the inclusion of the testing and release environments into projects. Agility for code modification code has obvious benefits, but if it takes teams weeks to test changes and months for releases, then architects cannot achieve agility in the overall architecture.

An architect must holistically analyze changes in technology and problem domain to determine the soundness of the architecture. While this kind of consideration rarely appears in a job posting, architects must meet this expectation to keep applications relevant.

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 in the latest technologies they use on a daily basis remain relevant (and to also retain a job!). However, an architect has an even more critical requirement to keep current on the latest technical and industry trends. The decisions an architect makes tend to be long lasting and generally difficult to change.

Understanding and following key trends helps the architect prepare for the future and also helps the architect make the correct decision.

Tracking trends and keeping current with those trends is hard, particularly for a software architect. We discuss various techniques and resources on how to keep current with the latest trends in Chapter 24 of the book.

Ensure Compliance With Decisions

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

Ensuring compliance means that the architect is continually verifying that development teams are following the architecture decisions and design principles defined, documented, and communicated by the architect. Consider the scenario where an architect makes a decision to restrict access to the database in a layered architecture to only the business and services layers (and not the presentation layer). This means that the presentation layer must go through all layers of the architecture to make even the simplest of database calls. A user interface developer might disagree with this decision and access the database (or the persistence layer) directly for performance reasons. However, the architect made that architecture decision for a specific reason - to control change. By closing the layers, database changes can be made without impacting the presentation layer. By not ensuring compliance with architecture decisions, violations like this can occur and the architecture will not meet the required architectural characteristics ("-ilities") and hence the application or system will not work as expected.

We talk more about measuring compliance using automated fitness functions and automated tools in Chapter 6.

Diverse Exposure and Experience

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

This expectation does not mean an architect must be an expert in every framework, platform, and language, but rather that an architect must at least be familiar with a variety of varying technologies. Most environments these days are heterogeneous, and at a minimum an architect should know how to interface with multiple systems and services, irrespective of the language, platform, and technology those systems or services are written in.

One of the best ways of mastering this expectation is for the architect to stretch her comfort zone. Focusing only on a single technology or platform is a safe haven. An effective software architect should be aggressive in seeking out opportunities to stretch that comfort zone and gain experience in multiple languages, platforms, and

technologies. A good way of mastering this expectation is to 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 is far more valuable for an architect to be familiar with ten different caching products and the associated pros and cons of each rather than to be an expert in only one of them.

Have Business Domain Knowledge

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

Effective software architects understand not only technology but also the business domain of a problem space. Without business domain knowledge, it is difficult to understand the business problem, goals, and requirements, making it difficult to design an effective architecture to meet the requirements of the business. Imagine being an architect at a large financial institution and not understanding common financial terms such as an average directional index, aleatory contracts, rates rally, or even non-priority debt. Without this knowledge an architect cannot communicate with stakeholders and business users, and will quickly lose credibility.

The most successful architects we know are those who have broad hands-on technical knowledge coupled with a strong knowledge of a particular domain. These software architects are able to effectively communicate with C-level executives and business users using the domain knowledge and language that these stakeholders know and understand. This in turn creates a strong level of confidence that the software architect knows what they are doing and is competent to create an effective and correct architecture. Knowing the business domain allows the software architect to better understand problems, issues, goals, data, and business processes, all of which are key factors when designing an effective software architecture.

Possess Interpersonal Skills

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

Having exceptional leadership and interpersonal skills is a difficult expectation for most developers and architects. As technologists, developers and architects like to solve technical problems, not people problems. However, as Gerald Weinberg is famous for saying, *"no matter what they tell you, its always a people problem"*. An architect is not only expected to provide technical guidance on the team, but also expected to lead the development teams through the implementation of the architecture. Leadership skills are at least half what it takes to become effective software architect, regardless of the role or title the architect has.

The industry is flooded with software architects, all competing for a limited number of architecture positions. Having strong leadership and interpersonal skills is a good way for an architect to differentiate themselves from other architects and stand out from the crowd. We've known many software architects who are excellent technologists, but ineffective architects due to the inability to lead teams, coach and mentor developers, and effectively communicate ideas and architecture decisions and principles. Needless to say, those architects had difficulties holding a position or job. Leadership and negotiation skills are so important that we've dedicated an entire chapter to the book on the software architect as a leader (see Chapter 23).

Understand and Navigate Politics

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

It might seem rather strange talk about negotiation and navigating office politics in a book about software architecture. To illustrate how important and necessary negotiation skills are, consider the scenario where a developer makes the decision to leverage the Strategy Pattern to reduce the overall cyclomatic complexity of a particular piece of complex code. Who really cares? One might applaud the developer for using such a pattern, but in almost all cases the developer does not need to seek approval for such a decision. Now consider the scenario where an architect, responsible for a large customer relationship management system, is having issues controlling database access from other systems, securing certain customer data, and making any database schema change because too many other systems are using the CRM database. The architect therefore makes the decision to create what are called *application silos*, whereas each application database is only accessible from the application owning that database. Making this decision will give the architect better control over the customer data, security, and also change control. However, unlike the previous developer scenario, this decision will also be challenged by almost everyone in the company (with the possible exception of the CRM application team of course). In effect this is perhaps a million dollar decision. Other applications need the customer management data. If those applications are no longer able to access the database directly, they must now ask the CRM system for the data, requiring remote access calls through either REST, SOAP, or some other remote access protocol.

The main point is that *almost every decision an architect makes will be challenged*. Architectural decisions will be challenged by product owners, project managers, and business stakeholders due to increased costs or increased effort (time) involved. Architectural decisions will also be challenged by developers who feel their approach is better. In either case, the architect must navigate the politics of the company and apply basic negotiation skills to get most decisions approved. This fact can be very frustrating to a software architect, because most decisions made as a developer did not require approval or even a review. Programming aspects such as code structure,

class design, design pattern selection, and sometimes even language choice are all part of the art of programming. However, an architect, now able to finally be able to make broad and important decisions, must justify and fight for almost every one of those decisions. Negotiation skills are so critical and necessary that, like leadership skills, we've dedicated an entire chapter in the book to understanding them (see Chapter 23).

Intersection of Architecture and . . .

The scope of software architecture has grown over the last decade to encompass more and more responsibility and perspective. A decade ago, the typical relationship between architecture and operations was contractual and formal, with lots of bureaucracy. Most companies, trying to avoid the complexity of hosting their own operations, frequently outsourced operations to a third-party company, with contractual obligations for service level agreements such as uptime, scale, responsiveness, and a host of other important architectural characteristics. However, tectonic shifts in recent years include architectures such as microservices that freely leverage former solely operational concerns. In the past, a firm and artificial barrier existed between operations and architecture, leading to overly complex solutions to problems. For example, elastic scale was once painfully built into architectures (see Chapter 15) while microservices handle it less painfully via a liaison between architects and DevOps.

History: *Pets.com* and Why We Have Elastic Scale

The history of software development contains rich lessons, both good and bad. We assume that current capabilities (like elastic scale) just appeared one day because of some clever developer, but those ideas were often born of hard lessons. *Pets.com* represents an early example of hard lessons learned. They appeared in the early day so of the Internet, hoping to become the *Amazon.com* of pet supplies. Fortunately, they had a brilliant marketing department, which invented a compelling mascot: a sock puppet with a microphone that said irreverent things. The mascot became a super star, appearing in public at parades, national sporting events, and anywhere else that defines **saturation**.

Unfortunately, management at *Pets.com* apparently spent all their money on the mascot, not on infrastructure. Once orders started pouring in, they weren't prepared. The web site was slow, transactions were lost, deliveries delayed, and so on…pretty much the worse case scenario. So bad, in fact, that they closed the business shortly after their disastrous Christmas rush, selling the only remaining valuable asset (the mascot) to a competitor.

What they needed was elastic scale: the ability to spin up more instances of resources as needed. Cloud providers offer this feature as a commodity, but in the early days of

the Internet, companies had to manage their own infrastructure, and many companies fell victim to a previously unheard of phenomenon: too much success can kill the business. *Pets.com* and other similar horror stories lead engineers to develop the frameworks that architects enjoy now.

The following sections delve into some of the newer intersections of the role of architect and other parts of the organization, highlighting new capabilities and responsibilities for architects.

Engineering Practices

Traditionally, software architecture was separate from the development process used to create software. Dozens of popular methodologies exist to build software, including Waterfall, many flavors of agile (such as Scrum, eXtreme Programming, Lean, Crystal, and others), which mostly don't impact software architecture.

However, over the last few years, engineering advances have thrust process concerns upon software architecture. It is useful to separate software development *process* from *engineering practices*. By *process*, we mean how teams are formed and managed, how meetings are conducted, and workflow organization; it refers to the mechanics of how people organize and interact. Software *engineering* practices, on the other hand, refer to process-agnostic practices that have illustrated repeatable benefit. For example, continuous integration is a proven engineering practice that doesn't rely on a particular process.

The Path from eXtreme Programming to Continuous Delivery

The origins of eXtreme Programming nicely illustrate the difference between *process* and *engineering*. In the early 1990's, a group of experienced software developers, led by Kent Beck, started questioning the dozens of different development processes popular at the time. In their experience, it seemed that none of them created repeatably good outcomes. One of the XP founders said that choosing one of the extant processes was "no more guarantee of project success than flipping a coin." They decided to rethink how to build software, and started the XP project in March of 1996. To inform their process, they rejected the conventional wisdom and focused on the *practices* that lead to project success in the past, pushed to the extreme. Their reasoning follows: we have seen a correlation on previous projects between more tests and higher quality. Thus, the XP approach to testing took the practice to the extreme: do test-first development, ensuring that all code is tested before it enters the codebase.

XP was lumped into other popular agile processes that shared similar perspectives, but it was one of the few methodologies that included engineering practices such as automation, testing, continuous integration, and other concrete, experienced-based techniques. The efforts to continue advancing the engineering side of software devel-

opment continued with the Continuous Delivery book (an updated version of many XP practices) and saw fruition in the DevOps movement. In many ways, the DevOps revolution occurred when operations adopted many of the engineering practices originally espoused by XP: automation, testing, declarative single-sources of truth, and many others.

We strongly support these advances, which form the incremental steps that will eventually graduate software development into a proper engineering discipline.

Focusing on engineering practices is important. First, software development lacks many of the features of more mature engineering disciplines. For example, civil engineers can predict structural change with much more accuracy than similarly important aspects of software structure. Second, one of the Achilles heal's of software development is estimation—how much time, how many resources, how much money? Part of this difficulty lies with antiquated accounting practices that cannot accommodate the exploratory nature of software development, but another part is because we're traditionally bad at estimation, at least in part because of *unknown unknowns*.

> …because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know.
>
> —former United States Secretary of Defense Donald Rumsfeld

Unknown unknowns are the nemesis of software systems. Many projects start with a list of *known unknowns*: things developers must learn about the domain and technology they know is upcoming. However, projects also fall victim to *unknown unknowns*: things no one knew was going to crop up yet has appeared unexpectedly. This is why all Big Design Up Front software efforts suffer—architects cannot design for unknown unknowns.

> All architectures become iterative because of *unknown unknowns*, agile just recognizes this and does it sooner.
>
> —Mark Richards

Thus, while process is mostly separate from architecture, an iterative process fits the nature of software architecture better. Teams trying to build a modern system such as microservices using an antiquated process like Waterfall will find a great deal of friction from an antiquated process that ignores the reality of how software comes together.

Often, the architect is also the technical leader on projects and therefore determines the engineering practices the team uses. Just as architects must carefully consider the problem domain before choosing an architecture, they must also ensure that the

architectural style and engineering practices form a symbiotic mesh. For example, a microservices architecture assumes automated machine provisioning, automated testing and deployment, and a raft of other assumptions. Trying to build one of these architectures with an antiquated operations group, with manual processes and little testing, creates tremendous friction and challenges to success. Just as different problem domains lend themselves towards certain architectural styles, engineering practices have the same kind of symbiotic relationship.

The evolution of thought leading from eXtreme Programming to Continuous Delivery continues. Recent advances in engineering practices allow new capabilities within architecture. Neal's most recent book, Building Evolutionary Architectures, highlights new ways to think about the intersection of engineering practices and architecture, allowing better automation of architectural governance. While we won't summarize that book here, it gives an important new nomenclature and way of thinking about architectural characteristics that will infuse much of the remainder of this book.

The book covers techniques for building architectures that change gracefully over time. In Chapter 4, we described architecture as the combination of requirements and additional concerns, as illustrated in Figure 1-7.

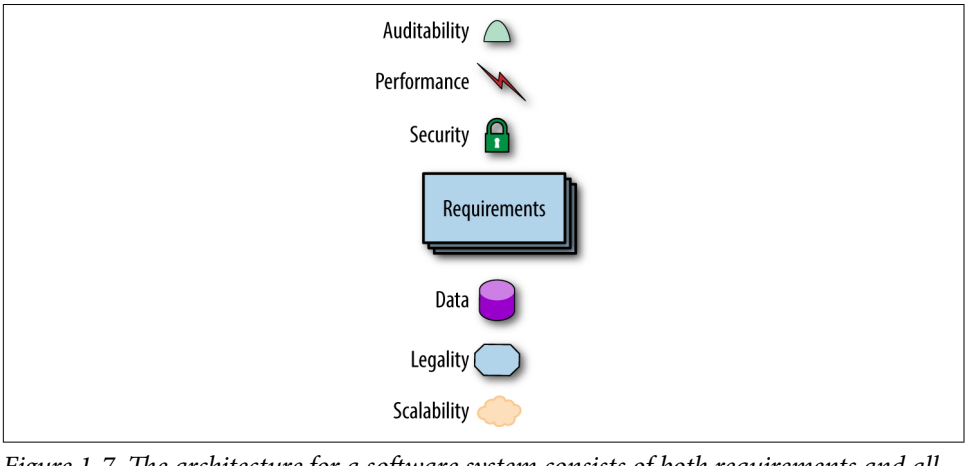


Figure 1-7. The architecture for a software system consists of both requirements and all the other architectural characteristics.

As any experience in the software development world illustrates, nothing remains static. Thus, architects may design a system to meet certain criteria, but that design must survive both implementation (how can architects make sure that their design is implemented correctly) and the inevitable change driven by the software development ecosystem. What we need is an *evolutionary architecture*.

The book *Building Evolutionary Architectures* introduces the concept of using *fitness functions* to protect (and govern) architectural characteristics as change occurs over

time. The concept of *fitness function* comes from evolutionary computing. When designing a genetic algorithm, developers have a variety of techniques to mutate the solution, evolving new solutions iteratively. When designing such an algorithm for a specific goal, developers must measure the outcome to see if it is closer or further away from an optimal solution; that measure is called a fitness function. A fitness function is an objective measure of how close a given solution is to the desired goal. For example, if developers designed a genetic algorithm to solve the traveling salesperson problem (whose goal is the shortest route between various cities), the fitness function would look at the path length.

Building Evolutionary Architectures co-opts this idea to create *architectural fitness functions*: an objective integrity assessment of some architectural characteristic(s). This may include a variety of mechanisms: metrics, unit tests, monitors, chaos engineering, and so on. For example, an architect may identify page load time as an importance characteristic of the architecture. To allow the system to change without degrading performance, the architecture builds a fitness function as a test that measures page load time for each page, and runs the test as part of the continuous integration for the project. Thus, architects always know the status of critical parts of the architecture because they have a verification mechanism in the form of fitness functions for each part.

We won't go into the full details of fitness functions here. However, we will point out opportunities and examples of the approach where applicable. However, note the correlation between how often fitness functions execute and the feedback they provide. Thus, adopting agile engineering practices such as continuous integration, automated machine provisioning, and similar practices make building resilient architectures easier. It also illustrates how intertwined architecture has become with engineering practices.

Operations/DevOps

The most obvious recent intersection between architecture and related fields recently occurred with the advent of DevOps, driven by some rethinking of architectural axioms. For many years, many companies considered operations as a separate function from software development; they often outsource operations to another company as a cost-savings measure. Many architectures designed during the 1990s and 2000's assumed that architects couldn't control operations, and built architectures defensively around that restriction (for a good example of this, see Space-based Architecture in Chapter 15).

However, a few years ago, several companies started experimenting with new forms of architecture that combine many operational concerns with architecture. For example, in older-style architectures such as ESB-driven SOA, the architecture was designed to handle things like elastic scale, greatly complicating the architecture in the process.

Basically, architects were forced to defensively design around the limitations introduced because of the cost-savings measure of outsourcing operations. Thus, they built architects that could handle scale, performance, elasticity, and a host of other capabilities internally. The side effect of that design is vastly more complex architecture.

The builders of the microservices style of architecture realized that these operational concerns are better handled by operations. By creating a liaison between architecture and operations, the architects can simplify the design and rely on operations for the things they handle best. Thus, by realizing a misappropriation of resources led to accidental complexity, architects and operations teamed up to create microservices, the details of which we cover in Chapter 17.

Process

Software architecture is mostly orthogonal to the software development process; the way that you build software (*process*) has little impact on the software architecture (*structure*). Thus, while the software development process a team uses has some impact on software architecture (especially around engineering practices), they are mostly separate. Most books on software architecture ignore the software development process outside making specious assumptions about things like predictability. However, the process by which teams develop software has an impact on many facets of software architecture. For example, many companies over the last few decades have adopted agile development methodologies discussed above because of the nature of software. Architects in agile projects can assume iterative development and therefore a faster feedback loop for decisions. That in turn leads to the ability for architects to be more aggressive about experimentation and other knowledge that relies on feedback.

As Mark noted in a previous section, all architecture becomes iterative; it's only a matter of time. Towards that end, we're going assume a baseline of agile methodologies throughout and call out exceptions where appropriate. For example, it is still common for many monolithic architectures to use older processes because of their age, politics, or other mitigating factors unrelated to software.

One critical aspect of architecture where agile methodologies shine: restructuring. Teams often find that they need to migrate their architecture from one pattern to another. For example, the team started with a monolithic architecture because it was easy and fast to bootstrap, but now they need to move it to a more modern architecture. Agile methodologies support these kinds of changes better than planning-heavy processes because of the tight feedback loop and encouragement of techniques like the Strangler Pattern and feature toggles.

Data

A large percentage of serious application development includes external data storage, often in the form of a relational (or, increasingly, NoSQL) database. However, many books about software architecture include light treatment of this important aspect of architecture. Code and data have a symbiotic relationship: one isn't useful without the other.

Database administrators often work alongside architects to build data architecture for complex systems, analyzing how relationships and reuse will affect a portfolio of applications. We won't delve into that level of specialized detail in this book, but rather save that detailed discussion for our next book, *Architecture The Hard Parts*. However, we also don't ignore the existence and dependence on external storage. In particular, when we talk about the operational aspects of architecture and *architectural quantum* (see Chapter 3), we include important external concerns such as databases.

Laws of Software Architecture

While the scope of software architecture is almost impossibly broad, unifying elements do exist. The authors have first and foremost learned the first law by constantly stumbling across it:

> Everything in software architecture is a tradeoff.
>
> —1st Law of Software Architecture

Nothing exists on a nice clean spectrum for software architects—every decision must take into account many opposing factors.

> If an architect thinks they have discovered something that *isn't* a tradeoff, more likely they just haven't *identified* the tradeoff yet.
>
> —Corollary 1

About the Authors

Mark Richards is an experienced hands-on software architect involved in the architecture, design, and implementation of microservices architectures, service oriented architectures, and distributed systems in J2EE and other technologies.

Neal Ford is Director, Software Architect, and Meme Wrangler at ThoughtWorks, a global IT consultancy with an exclusive focus on end-to-end software development and delivery. Before joining ThoughtWorks, Neal was the Chief Technology Officer at The DSW Group, Ltd., a nationally recognized training and development firm.