O'REILLY®

2nd Edition

Building Evolutionary Architectures

Automated Software Governance

Free Chapter

Neal Ford, Rebecca Parsons, Patrick Kua & Pramod Sadalage

Forewords by Mark Richards & Martin Fowler

2ND EDITION

Building Evolutionary Architectures

Automated Software Governance

Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage

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

Building Evolutionary Architectures
by Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage

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: Melissa Duffield
Development Editor: Virginia Wilson
Production Editor: Christopher Faucher
Copyeditor: Audrey Doyle
Proofreader: Piper Editorial Consulting, LLC
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: O'Reilly Media, Inc.

October 2017:	First Edition
December 2022:	Second Edition

Revision History for the Second Edition

2022-11-22: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492097549* for release details.

978-1-492-09754-9

[LSI]

Table of Contents

CHAPTER 1
Evolving Software Architecture

Building systems that age gracefully and effectively is one of the enduring challenges of software development generally and software architecture specifically. This book covers two fundamental aspects of how to build evolvable software: utilizing effective engineering practices derived from the agile software movement and structuring architecture to facilitate change and governance.

Readers will grow to understand the state of the art in how to manage change in architecture in a deterministic way, unifying previous attempts at providing protection for architecture characteristics and actionable techniques to improve the ability to change architecture without breaking it.

The Challenges of Evolving Software

bit rot: also known as *software rot*, *code rot*, *software erosion*, *software decay*, or *software entropy*, is either a slow deterioration of software quality over time or its diminishing responsiveness that will eventually lead to software becoming faulty.

Teams have long struggled with building high-quality software that *remains* high quality over time, including adages that reflect this difficulty, such as the varied definitions of *bit rot* shown above. At least two factors drive this struggle: the problems of policing all the various moving parts in complex software, and the dynamic nature of the software development ecosystem.

Modern software consists of thousands or millions of individual parts, each of which may be changed along some set of dimensions. Each of those changes has predictable and sometimes unpredictable effects. Teams that attempt manual governance eventually become overwhelmed by the sheer volume of parts and combinatorial side effects.

Managing the myriad interactions of software would be bad enough against a static backdrop, but that doesn't exist. The software development ecosystem consists of all the tools, frameworks, libraries, and best practices—the accumulated state of the art in software development at any given snapshot in time. This ecosystem forms an equilibrium—much like a biological system—that developers can understand and build things within. However, that equilibrium is *dynamic*—new things come along constantly, initially upsetting the balance until a new equilibrium emerges. Visualize a unicyclist carrying boxes: *dynamic* because the unicyclist continues to adjust to stay upright, and *equilibrium* because they continue to maintain balance. In the software development ecosystem, each new innovation or practice may disrupt the status quo, forcing the establishment of a new equilibrium. Metaphorically, we keep tossing more boxes onto the unicyclist's load, forcing them to reestablish balance.

In many ways, architects resemble our hapless unicyclist, constantly both balancing and adapting to changing conditions. The engineering practices of Continuous Delivery represent such a tectonic shift in the equilibrium: incorporating formerly siloed functions such as operations into the software development lifecycle enabled new perspectives on what *change* means. Enterprise architects can no longer rely on static, five-year plans because the entire software development universe will evolve in that time frame, rendering every long-term decision potentially moot.

Disruptive change is hard to predict even for savvy practitioners. The rise of containers via tools like Docker is an example of an unknowable industry shift. However, we can trace the rise of containerization via a series of small, incremental steps. Once upon a time, operating systems, application servers, and other infrastructure were commercial entities, requiring licensing and great expense. Many of the architectures designed in that era focused on efficient use of shared resources. Gradually, Linux became good enough for many enterprises, reducing the *monetary* cost of operating systems to zero. Next, DevOps practices like automatic machine provisioning via tools like Puppet and Chef made Linux *operationally* free. Once the ecosystem became free and widely used, consolidation around common portable formats was inevitable: thus, Docker. But containerization couldn't have happened without all the evolutionary steps leading to that end.

The software development ecosystem constantly evolves, which leads to new architectural approaches. While many developers suspect that a cabal of architects retreat to an ivory tower to decide what the *Next Big Thing* will be, the process is much more organic. New capabilities constantly arise within our ecosystem, providing new ways to combine with existing and other new features to enable new capabilities. For example, consider the recent rise of microservices architectures. As open source operating systems became popular, combined with Continuous Delivery–driven engineering practices, enough clever architects figured out how to build more scalable systems that they eventually needed a name: thus, microservices.

Why We Didn't Have Microservices in the Year 2000

Consider an architect with a time machine who travels back in time to the year 2000 and approaches the head of operations with a new idea.

"I have a great new concept for an architecture that allows fantastic isolation between each of the capabilities—it's called *microservices*; we'll design each service around business capabilities and keep things highly decoupled."

"Great," says the head of operations. "What do you need?"

"Well, I'm going to need about 50 new computers, and of course 50 new operating system licenses, and another 20 computers to act as isolated databases, along with licenses for those. When do you think I can get all that?"

"Please leave my office."

While microservices might have seemed like a good idea even back then, the ecosystem wasn't available to support it.

A portion of an architect's job is structural design to solve particular problems—you have a problem, and you've decided that software will solve it. When considering structural design, we can partition it into two areas: the *domain (or requirements)* and *architecture characteristics*, as illustrated in Figure 1-1.

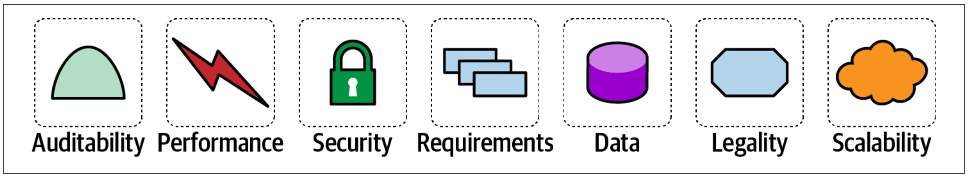

Figure 1-1. The entire scope of software architecture encompasses requirements plus architecture characteristics: the "-ilities" of software

The requirements shown in Figure 1-1 represent whatever problem domain the software solution addresses. The other parts are variously known as *architecture characteristics* (our preferred term), *nonfunctional requirements*, *system quality attributes*, *cross-cutting requirements*, and a host of other names. Regardless of the name, they represent critical capabilities required for project success, both for initial release and long-term maintainability. For example, architecture characteristics such as *scale* and *performance* may form success criteria for a market, whereas others such as *modularity* contribute to *maintainability* and *evolvability*.

The Many Names of Architecture Characteristics

We use the term *architecture characteristics* throughout the book to refer to nondomain design considerations. However, many organizations use other terms for this concept, among them nonfunctional requirements, cross-cutting requirements, and system quality attributes. We don't have a strong preference for one term over another —feel free to translate our term to yours anywhere you see it in the book. These are not distinct concepts.

Software is rarely static; it continues to evolve as teams add new features, integration points, and a host of other common changes. What architects need are protection mechanisms for architecture characteristics, similar to unit tests but focused on architecture characteristics, which change at a different rate and are sometimes subject to forces that are different from the domain. For example, technical decisions within a company may drive a database change that is independent of the domain solution.

This book describes the mechanisms and design techniques for adding the same kind of continual assurance about architectural governance that high-performing teams now have about other aspects of the software development process.

Architectural decisions are ones in which each choice offers significant trade-offs. Throughout this book, when we refer to the role of *architect*, we encompass anyone who makes architectural decisions, regardless of their title in an organization. Additionally, important architecture decisions virtually always require collaboration with other roles.

Do Agile Projects Need Architecture?

This is a common question asked of those who have utilized agile engineering practices for a while. The goal of agility is to remove *useless* overhead, not necessary steps such as design. As in many things in architecture, the scale dictates the level of architecture. We use the analogy of building—if we want to build a dog house, we don't need an elaborate architecture; we just need materials. On the other hand, if we need to build a 50-story office building, design must occur. Similarly, if we need a website to track a simple database, we don't need an architecture; we can find materials that enable us to piece it together. However, we must carefully consider many trade-offs to design a highly scalable and available website, such as a high-volume concert ticketing website.

Rather than the question *Do Agile projects need architecture?*, the question for architects lies in how little unnecessary design they can afford, while building the ability to iterate on early designs to work toward more suitable solutions.

Evolutionary Architecture

Both the mechanisms for evolution and the decisions architects make when designing software derive from the following definition:

> An evolutionary software architecture supports *guided*, *incremental* change across *multiple dimensions*.

The definition consists of three parts, which we describe in more detail below.

Guided Change

Once teams have chosen important characteristics, they want to *guide* changes to the architecture to protect those characteristics. For that purpose, we borrow a concept from evolutionary computing called *fitness functions*. A fitness function is an objective function used to summarize how close a prospective design solution is to achieving the set aims. In evolutionary computing, the fitness function determines whether an algorithm has improved over time. In other words, as each variant of an algorithm is generated, the fitness functions determine how "fit" each variant is, based on how the designer of the algorithm defined "fit."

We have a similar goal in evolutionary architecture: as architecture evolves, we need mechanisms to evaluate how changes impact the important characteristics of the architecture and prevent degradation of those characteristics over time. The fitness function metaphor encompasses a variety of mechanisms we employ to ensure architecture doesn't change in undesirable ways, including metrics, tests, and other verification tools. When an architect identifies an architectural characteristic they want to protect as things evolve, they define one or more fitness functions to protect that feature.

Historically, a portion of architecture has often been viewed as a governance activity, and architects have only recently accepted the notion of enabling change through architecture. Architectural fitness functions allow decisions in the context of the organization's needs and business functions, while making the basis for those decisions explicit and testable. Evolutionary architecture is not an unconstrained, irresponsible approach to software development. Rather, it is an approach that balances the need for rapid change and the need for rigor around systems and architectural characteristics. The fitness function drives architectural decision making, guiding the architecture while allowing the changes needed to support changing business and technology environments.

We use *fitness functions* to create evolutionary guidelines for architectures; we cover them in detail in Chapter 2.

Incremental Change

Incremental change describes two aspects of software architecture: how teams build software incrementally and how they deploy it.

During development, an architecture that allows small, incremental changes is easier to evolve because developers have a smaller scope of change. For deployment, incremental change refers to the level of modularity and decoupling for business features and how they map to architecture. An example is in order.

Let's say that PenultimateWidgets, a large seller of widgets, has a catalog page backed by a microservices architecture and modern engineering practices. One of the page's features enables users to rate different widgets with star ratings. Other services within PenultimateWidgets' business also need ratings (customer service representatives, shipping provider evaluation, etc.), so they all share the star rating service. One day, the star rating team releases a new version alongside the existing one that allows half-star ratings—a small but significant upgrade. The other services that require ratings aren't required to move to the new version but to gradually migrate as convenient. Part of PenultimateWidgets' DevOps practices include architectural monitoring of not only the services but also the routes between services. When the operations group observes that no one has routed to a particular service within a given time interval, they automatically disintegrate that service from the ecosystem.

This is an example of incremental change at the architectural level: the original service can run alongside the new one as long as other services need it. Teams can migrate to new behavior at their leisure (or as need dictates), and the old version is automatically garbage collected.

Making incremental change successful requires coordination of a handful of Continuous Delivery practices. Not all of these practices are required in all cases; rather, they commonly occur together in the wild. We discuss how to achieve incremental change in Chapter 3.

Multiple Architectural Dimensions

> There are no separate systems. The world is a continuum. Where to draw a boundary around a system depends on the purpose of the discussion.
>
> —Donella H. Meadows

Classical Greek physicists gradually learned to analyze the universe based on fixed points, culminating in classical mechanics. However, more precise instruments and more complex phenomena gradually refined that definition toward relativity in the early 20th century. Scientists realized that what they previously viewed as isolated phenomena in fact interact relative to one another. Since the 1990s, enlightened architects have increasingly viewed software architecture as multidimensional.

Continuous Delivery expanded that view to encompass operations. However, software architects often focus primarily on *technical* architecture—how the software components fit together—but that is only one dimension of a software project. If architects want to create an architecture that can evolve, they must consider *all* the interconnected parts of the system that change affects. Just like we know from physics that everything is relative to everything else, architects know there are many dimensions to a software project.

To build evolvable software systems, architects must think beyond just the technical architecture. For example, if the project includes a relational database, the structure and relationship between database entities will evolve over time as well. Similarly, architects don't want to build a system that evolves in a manner that exposes a security vulnerability. These are all examples of *dimensions* of architecture—the parts of architecture that fit together in often orthogonal ways. Some dimensions fit into what are often called *architectural concerns* (the list of "-ilities" referred to earlier), but *dimensions* are actually broader, encapsulating things traditionally outside the purview of technical architecture. Each project has dimensions the architect role must consider when thinking about evolution. Here are some common dimensions that affect evolvability in modern software architectures:

Technical
: The implementation parts of the architecture: the frameworks, dependent libraries, and implementation language(s).

Data
: Database schemas, table layouts, optimization planning, and so on. The database administrator generally handles this type of architecture.

Security
: Defines security policies and guidelines, and specifies tools to help uncover deficiencies.

Operational/System
: Concerns how the architecture maps to existing physical and/or virtual infrastructure: servers, machine clusters, switches, cloud resources, and so on.

Each of these perspectives forms a *dimension* of the architecture—an intentional partitioning of the parts supporting a particular perspective. Our concept of architectural dimensions encompasses traditional architectural characteristics ("-ilities") plus any other role that contributes to building software. Each of these forms a perspective on architecture that we want to preserve as our problem evolves and the world around us changes.

When architects think in terms of architectural dimensions, it provides a mechanism by which they can analyze the evolvability of different architectures by assessing how each important dimension reacts to change. As systems become more intertwined with competing concerns (scalability, security, distribution, transactions, etc.), architects must expand the dimensions they track on projects. To build an evolvable system, architects must think about how the system might evolve across all the important dimensions.

The entire architectural scope of a project consists of the software requirements plus the other dimensions. We can use fitness functions to protect those characteristics as the architecture and the ecosystem evolve together through time, as illustrated in Figure 1-2.

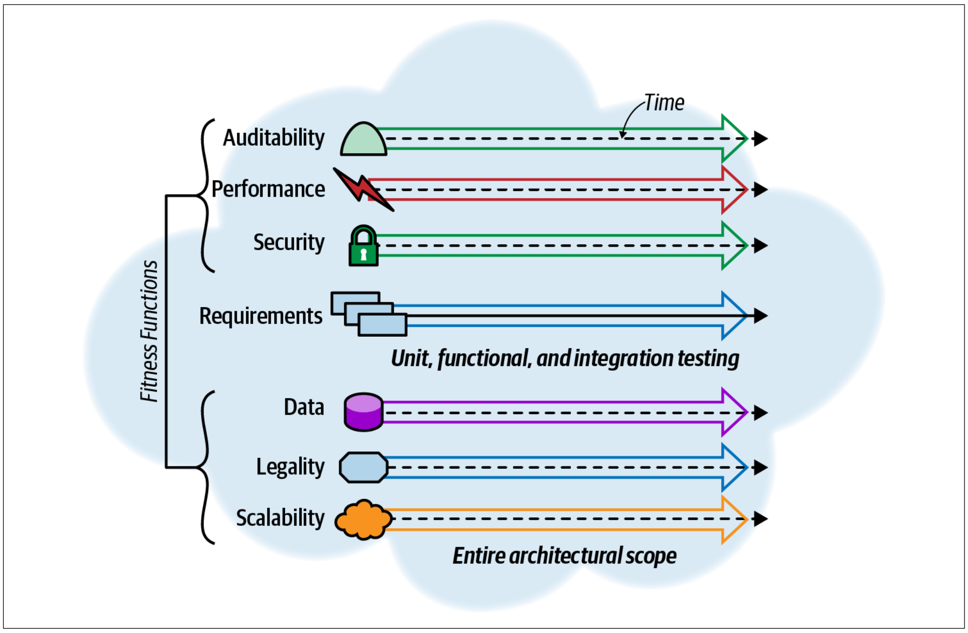

Figure 1-2. An architecture consists of requirements and other dimensions, each protected by fitness functions

In Figure 1-2, the architects have identified *auditability*, *data*, *security*, *performance*, *legality*, and *scalability* as the additional architectural characteristics important for this application. As the business requirements evolve over time, each of the architectural characteristics utilizes fitness functions to protect its integrity as well.

While the authors of this text stress the importance of a holistic view of architecture, we also realize that a large part of evolving architecture concerns technical architecture patterns and related topics like coupling and cohesion. We discuss how technical architecture coupling affects evolvability in Chapter 5 and the impacts of data coupling in Chapter 6.

Coupling applies to more than just structural elements in software projects. Many software companies have recently discovered the impact of team structure on surprising things like architecture. We discuss all aspects of coupling in software, but the team impact comes up so early and often that we need to discuss it here.

Evolutionary architecture helps answer two common questions that arise among architects in the modern software development ecosystem: *How is long-term planning possible when everything changes all the time?* and *Once I've built an architecture, how can I prevent it from degrading over time?* Let's explore these questions in more detail.

How Is Long-Term Planning Possible When Everything Changes All the Time?

The programming platforms we use exemplify constant evolution. Newer versions of a programming language offer better APIs to improve the flexibility of or applicability to new problems; newer programming languages offer a different paradigm and different set of constructs. For example, Java was introduced as a C++ replacement to ease the difficulty of writing networking code and to improve memory management issues. When we look at the past 20 years, we observe that many languages still continually evolve their APIs while totally new programming languages appear to regularly attack newer problems. The evolution of programming languages is demonstrated in Figure 1-3.

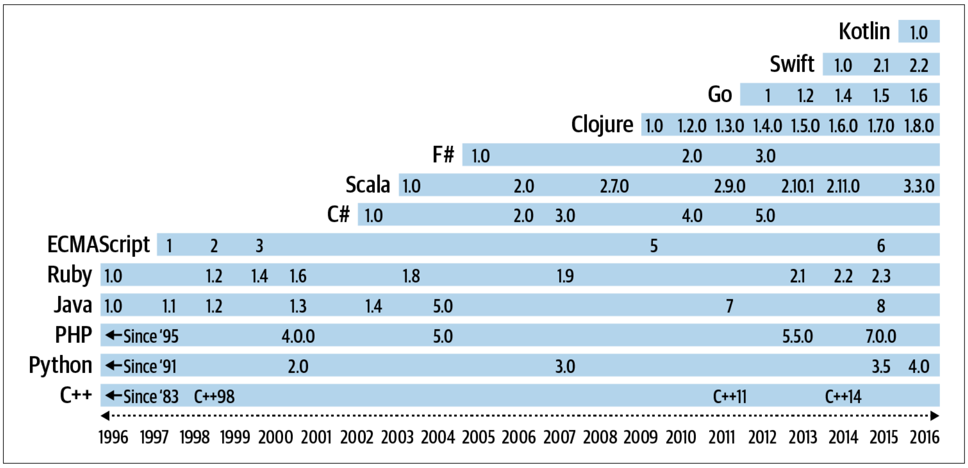

Figure 1-3. The evolution of popular programming languages

Regardless of the particular aspect of software development—the programming platform, languages, operating environment, persistence technologies, cloud offerings, and so on—we expect constant change. Although we cannot predict when changes in the technical or domain landscape will occur, or which changes will persist, we know change is inevitable. Consequently, we should architect our systems knowing the technical landscape will change.

If the ecosystem constantly changes in unexpected ways, and if predictability is impossible, what is the *alternative* to fixed plans? Enterprise architects and other developers must learn to adapt. Part of the traditional reasoning behind making long-term plans was financial; software changes were expensive. However, modern engineering practices invalidate that premise by making change less expensive through the automation of formerly manual processes and other advances such as DevOps.

For years, many smart developers recognized that some parts of their systems were harder to modify than others. That's why *software architecture* is defined as "the parts that are hard to change later." This convenient definition partitioned the things you *can* modify without much effort from truly difficult changes. Unfortunately, this definition also evolved into a blind spot when thinking about architecture: developers' assumption that change is difficult becomes a self-fulfilling prophecy.

Several years ago, some innovative software architects revisited the "hard to change later" problem: what if we build changeability *into* the architecture? In other words, if *ease of change* is a bedrock principle of the architecture, then change is no longer difficult. Building evolvability into architecture in turn allows a whole new set of behaviors to emerge, upsetting the dynamic equilibrium again.

Even if the ecosystem doesn't change, what about the gradual erosion of architectural characteristics that occurs? Architects design architectures but then expose them to the messy real world of *implementing* things atop the architecture. How can architects protect the important parts they have defined?

Once I've Built an Architecture, How Can I Prevent It from Degrading Over Time?

An unfortunate decay, often called *bit rot*, occurs in many organizations. Architects choose particular architectural patterns to handle the business requirements and "-ilities," but those characteristics often accidentally degrade over time. For example, if an architect has created a layered architecture with presentation at the top, persistence at the bottom, and several layers in between, developers who are working on reporting will often ask permission to directly access persistence from the presentation layer, bypassing the other layers, for performance reasons. Architects build layers to isolate change. Developers then bypass those layers, increasing coupling and invalidating the reasoning behind the layers.

Once they have defined the important architectural characteristics, how can architects *protect* those characteristics to ensure they don't erode? Adding *evolvability* as an architectural characteristic implies protecting the other characteristics as the system evolves. For example, if an architect has designed an architecture for scalability, they don't want that characteristic to degrade as the system evolves. Thus, *evolvability* is a meta-characteristic, an architectural wrapper that protects all the other architectural characteristics.

The mechanism of evolutionary architecture heavily overlaps with the concerns and goals of architectural governance—defined principles around design, quality, security, and other quality concerns. This book illustrates the many ways that evolutionary architecture approaches enable automating architectural governance.

Why Evolutionary?

A common question about evolutionary architecture concerns the name itself: why call it *evolutionary* architecture and not something else? Other possible terms include *incremental*, *continual*, *agile*, *reactive*, and *emergent*, to name just a few. But each of these terms misses the mark. The definition of evolutionary architecture that we state here includes two critical characteristics: incremental and guided.

The terms *continual*, *agile*, and *emergent* all capture the notion of change over time, which is clearly a critical characteristic of an evolutionary architecture, but none of these terms explicitly captures any notion of *how* an architecture changes or what the desired end-state architecture might be. While all the terms imply a changing environment, none of them covers what the architecture should look like. The *guided* part of our definition reflects the architecture we want to achieve—our end goal.

We prefer the word *evolutionary* over *adaptable* because we are interested in architectures that undergo fundamental evolutionary change, not ones that have been patched and adapted into increasingly incomprehensible accidental complexity. *Adapting* implies finding some way to make something work regardless of the elegance or longevity of the solution. To build architectures that truly evolve, architects must support genuine change, not jury-rigged solutions. Going back to our biological metaphor, *evolutionary* concerns the process of having a system that is fit for purpose and can survive the ever-changing environment in which it operates. Systems may have individual adaptations, but as architects, we should care about the overall evolvable system.

Another useful comparison architects can make is between *evolutionary architecture* and *emergent design*, and why there is not such a thing as an "emergent architecture." One common misconception about agile software development is the alleged lack of architecture: "Let's just start coding and the architecture will emerge as we go." However, this depends on how simple the problem is. Consider a physical building. If you need to build a dog house, you don't need an architecture; you can go to the hardware store and buy lumber and bang it together. If, on the other hand, you need to build a 50-floor office building, architecture is definitely required! Similarly, if you are building a simple catalog system for a small number of users, you likely don't need a lot of up-front planning. However, if you are designing a software system that needs strict performance for a large number of users, planning is necessary! The purpose of agile architecture isn't *no* architecture; it's *no useless* architecture: don't go through bureaucratic processes that don't add value to the software development process.

Another complicating factor in software architecture is the different types of essential complexity architects must design for. When evaluating trade-offs, it's often not the easy *simple* versus *complex* system distinction but rather systems that are complex in different ways. In other words, each system has a unique set of criteria for success. While we discuss architectural styles such as microservices, each style is a starting point for a complex system that grows to look like no other.

Similarly, if an architect builds a very simple system, they can afford to pay little attention to architectural concerns. However, sophisticated systems require purposeful design, and they need a starting point. *Emergence* suggests that you can start with nothing, whereas architecture provides the scaffolding or structure for all the other parts of the system; something must be in place to begin.

The concept of *emergence* also implies that teams can slowly crescendo their design toward the ideal architectural solution. However, like building architecture, there is no perfect architecture, only different ways architects deal with trade-offs. Architects can implement most problems in a wide variety of different architecture styles and be successful. However, some of them will fit the problem better, offering less resistance and fewer workarounds.

One key to evolutionary architecture is the balance between how much structure and governance is necessary to support long-term goals and needless formality and friction.

Summary

Useful software systems aren't static. They must grow and change as the problem domain changes and the ecosystem evolves, providing new capabilities and complexities. Architects and developers can gracefully evolve software systems, but they must understand both the necessary engineering practices to make that happen and how best to structure their architecture to facilitate change.

Architects are also tasked with governing the software they design, along with many of the development practices used to build it. Fortunately, the mechanisms we uncover to allow easier evolution also provide ways to automate important software governance activities. We take a deep dive into the mechanics of how to make this happen in the next chapter.

About the Authors

Neal Ford is director, software architect, and meme wrangler at ThoughtWorks, a software company and a community of passionate, purpose-led individuals who think disruptively to deliver technology to address the toughest challenges, all while seeking to revolutionize the IT industry and create positive social change. Before joining ThoughtWorks, Neal was the chief technology officer at The DSW Group, Ltd., a nationally recognized training and development firm.

Neal has a degree in computer science from Georgia State University specializing in languages and compilers and a minor in mathematics specializing in statistical analysis. He is an internationally recognized expert on software development and delivery, especially at the intersection of agile engineering techniques and software architecture. Neal has authored magazine articles, nine books (and counting), and dozens of video presentations and has spoken at hundreds of developer conferences worldwide. The topics of these works include software architecture, Continuous Delivery, functional programming, and cutting-edge software innovations, as well as a business-focused book and video on improving technical presentations. His primary consulting focus is the design and construction of large-scale enterprise applications. If you have an insatiable curiosity about Neal, visit his website at *nealford.com*.

Dr. Rebecca Parsons is ThoughtWorks' chief technology officer with decades-long applications development experience across a range of industries and systems. Her technical experience includes leading the creation of large-scale distributed object applications, the integration of disparate systems, and working with architecture teams. Separate from her passion for deep technology, Dr. Parsons is a strong advocate for diversity in the technology industry.

Before joining ThoughtWorks, Dr. Parsons worked as an assistant professor of computer science at the University of Central Florida where she taught courses in compilers, program optimization, distributed computation, programming languages, theory of computation, machine learning, and computational biology. She also worked as a Director's Postdoctoral Fellow at the Los Alamos National Laboratory researching issues in parallel and distributed computation, genetic algorithms, computational biology, and nonlinear dynamical systems.

Dr. Parsons received a Bachelor of Science degree in computer science and economics from Bradley University, a Master of Science in computer science from Rice University, and her PhD in computer science from Rice University. She is also the coauthor of *Domain-Specific Languages*, *The ThoughtWorks Anthology*, and *Building Evolutionary Architectures*, 1st edition.

Patrick Kua is an independent CTO coach, former CTO of N26, and former principal technical consultant at ThoughtWorks, having worked in the technology industry for over 20 years. His personal mission is to accelerate the growth of technical leaders, and he does that through one-on-one coaching, online and in-person technical leadership workshops, and his popular newsletter for leaders in tech, *Level Up*.

He is the author of *The Retrospective Handbook: A Guide for Agile Teams and Talking with Tech Leads: From Novices to Practitioners* and offers training via the *The Tech Lead Academy*.

You can discover more about him at his website, *patkua.com*, or reach out to him on twitter at @patkua.

Pramod Sadalage is director of Data & DevOps at ThoughtWorks, where he enjoys the rare role of bridging the divide between database professionals and application developers. He is usually sent to clients with particularly challenging data needs that require new technologies and techniques. In the early 2000s he developed techniques to allow relational databases to be designed in an evolutionary manner based on version-controlled schema migrations.

He is coauthor of *Software Architecture: The Hard Parts*, coauthor of *Refactoring Databases*, coauthor of *NoSQL Distilled*, and author of *Recipes for Continuous Database Integration*, and he continues to speak and write about the insights he and his clients learn.

Colophon

The animal on the cover of *Building Evolutionary Architectures* is the open brain coral (*Trachyphyllia geoffroyi*). Also known as a “folded brain” or “crater” coral, this large-polyp stony (LPS) coral is native to the Indian Ocean.

Known for its distinctive folds, bright colors, and hardiness, this free-living coral subsists on the photosynthetic output of a surface layer of zooxanthellae during the day, while at night it extends tentacles from its polyps to steer prey, which include various plankton as well as small fish, into one of its mouths (some open brain corals have two or three of them).

Because of its striking appearance and easy-to-accommodate diet, *Trachyphyllia geoffroyi* is a popular choice for aquariums, where it thrives in the bottom layer of sand and/or silt resembling the shallow seafloors of its native habitat. They benefit from an environment with moderate water flow and rich with plant and animal matter to consume.

Trachyphyllia geoffroyi is listed on the IUCN Red List at Near Threatened status. Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from Jean Vincent Félix Lamouroux’s *Exposition Methodique des genres de L’Ordre des Polypiers*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.