O'REILLY®

Software Architecture: The Hard Parts

Modern Trade-Off Analyses for Distributed Architectures

Neal Ford,
Mark Richards,
Pramod Sadalage &
Zhamak Dehghani

O'REILLY®

Software Architecture: The Hard Parts

There are no easy decisions in software architecture. Instead, there are many hard parts—difficult problems or issues with no best practices—that force you to choose among various compromises. With this book, you'll learn how to think critically about the trade-offs involved with distributed architectures.

Architecture veterans and practicing consultants Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani discuss strategies for choosing an appropriate architecture. By interweaving a story about a fictional group of technology professionals—the Sysops Squad—they examine everything from how to determine service granularity, manage workflows and orchestration, manage and decouple contracts, and manage distributed transactions to how to optimize operational characteristics, such as scalability, elasticity, and performance.

By focusing on commonly asked questions, this book provides techniques to help you discover and weigh the trade-offs as you confront the issues you face as an architect.

- **Analyze trade-offs and effectively document your decisions**
- **Make better decisions regarding service granularity**
- **Understand the complexities of breaking apart monolithic applications**
- **Manage and decouple contracts between services**
- **Handle data in a highly distributed architecture**
- **Learn patterns to manage workflow and transactions when breaking apart applications**

"This book is a must for every architect that is building modern distributed systems."

—Aleksandar Serafimoski
Lead Consultant, Thoughtworks

Neal Ford is a director, software architect, and meme wrangler at Thoughtworks, a leading global technology consultancy.

Mark Richards is a hands-on software architect with experience in designing and implementing microservices, service-oriented architectures, and distributed systems.

Pramod Sadalage is director of data and DevOps at Thoughtworks, with experience in application and evolutionary database development, data architecture, NoSQL databases, and database refactoring patterns.

Zhamak Dehghani is a director of technology at Thoughtworks, focusing on distributed architecture and emerging technologies. She is the founder of Data Mesh.

SOFTWARE DEVELOPMENT

US $69.99 CAN $92.99
ISBN: 978-1-492-08689-5

Twitter: @oreillymedia
facebook.com/oreilly

Praise for *Software Architecture: The Hard Parts*

"This book provides the missing manual around building microservices and analyzing the nuances of architectural decisions throughout the whole tech stack. In this book, you get a catalog of architectural decisions you can make when building your distributed system and what are the pros and cons associated with each decision. This book is a must for every architect that is building modern distributed systems."

—Aleksandar Serafimoski, Lead Consultant, Thoughtworks

"It's a must-read for technologists who are passionate about architecture. Great articulation of patterns."

—Vanya Seth, Head Of Tech, Thoughtworks India

"Whether you're an aspiring architect or an experienced one leading a team, no handwaving, this book will guide you through the specifics of how to succeed in your journey to create enterprise applications and microservices."

—Dr. Venkat Subramaniam,
Award-winning Author and Founder of Agile Developer, Inc.

"*Software Architecture: The Hard Parts* provides the reader with valuable insight, practices, and real-world examples on pulling apart highly coupled systems and building them back up again. By gaining effective trade-off analysis skills, you will start to make better architecture decisions."

—Joost van Wenen,
Managing Partner & Cofounder, Infuze Consulting

“I loved reading this comprehensive body of work on distributed architectures! A great mix of solid discussions on fundamental concepts, together with tons of practical advice.”

—David Kloet, Independent Software Architect

“Splitting a big ball of mud is no easy work. Starting from the code and getting to the data, this book will help you see the services that should be extracted and the services that should remain together.”

—Rubén Díaz-Martínez, Software Developer at Codesai

“This book will equip you with the theoretical background and with a practical framework to help answer the most difficult questions faced in modern software architecture.”

—James Lewis, Technical Director, Thoughtworks

Software Architecture: The Hard Parts

Modern Trade-Off Analyses for Distributed Architectures

Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Software Architecture: The Hard Parts

by Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani

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: Nicole Taché
Production Editor: Christopher Faucher
Copyeditor: Sonia Saruba
Proofreader: Sharon Wilkey
Indexer: Sue Klefstad
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: O'Reilly Media, Inc.

October 2021: First Edition

Revision History for the First Edition

2021-09-23: First Release
2021-12-03: Second Release
2022-08-26: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492086895* for release details.

978-1-492-08689-5

[LSI]

Table of Contents

Part II. Putting Things Back Together

Preface

When two of your authors, Neal and Mark, were writing the book *Fundamentals of Software Architecture*, we kept coming across complex examples in architecture that we wanted to cover but that were too difficult. Each one offered no easy solutions but rather a collection of messy trade-offs. We set those examples aside into a pile we called "The Hard Parts." Once that book was finished, we looked at the now gigantic pile of hard parts and tried to figure out: *why are these problems so difficult to solve in modern architectures*?

We took all the examples and worked through them like architects, applying trade-off analysis for each situation, but also paying attention to the process we used to arrive at the trade-offs. One of our early revelations was the increasing importance of data in architecture decisions: who can/should access data, who can/should write to it, and how to manage the separation of analytical and operational data. To that end, we asked experts in those fields to join us, which allows this book to fully incorporate decision making from both angles: architecture to data and data to architecture.

The result is this book: a collection of difficult problems in modern software architecture, the trade-offs that make the decisions hard, and ultimately an illustrated guide to show you how to apply the same trade-off analysis to your own unique problems.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file paths.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> This element signifies a tip or suggestion.

Using Code Examples

Supplemental resources for this book are available at *http://architecturethehardparts.com*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Software Architecture: The Hard Parts* by Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani (O'Reilly). Copyright 2022 Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani, 978-1-492-08689-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/sa-the-hard-parts*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

Mark and Neal would like to thank all the people who attended our (almost exclusively online) classes, workshops, conference sessions, and user group meetings, as well as all the other people who listened to versions of this material and provided invaluable feedback. Iterating on new material is especially tough when we can't do it live, so we appreciate those who commented on the many iterations. We thank the publishing team at O'Reilly, who made this as painless an experience as writing a book can be. We also thank a few random oases of sanity-preserving and idea-sparking groups that have names like Pasty Geeks and the Hacker B&B.

Thanks to those who did the technical review of our book—Vanya Seth, Venkat Subramanian, Joost van Weenen, Grady Booch, Ruben Diaz, David Kloet, Matt Stein, Danilo Sato, James Lewis, and Sam Newman. Your valuable insights and feedback helped validate our technical content and make this a better book.

We especially want to acknowledge the many workers and families impacted by the unexpected global pandemic. As knowledge workers, we faced inconveniences that pale in comparison to the massive disruption and devastation wrought on so many of our friends and colleagues across all walks of life. Our sympathies and appreciation especially go out to health care workers, many of whom never expected to be on the front line of a terrible global tragedy yet handled it nobly. Our collective thanks can never be adequately expressed.

Acknowledgments from Mark Richards

In addition to the preceding acknowledgments, I once again thank my lovely wife, Rebecca, for putting up with me through yet another book project. Your unending support and advice helped make this book happen, even when it meant taking time away from working on your own novel. You mean the world to me, Rebecca. I also thank my good friend and coauthor Neal Ford. Collaborating with you on the materials for this book (as well as our last one) was truly a valuable and rewarding experience. You are, and always will be, my friend.

Acknowledgments from Neal Ford

I would like to thank my extended family, Thoughtworks as a collective, and Rebecca Parsons and Martin Fowler as individual parts of it. Thoughtworks is an extraordinary group of people who manage to produce value for customers while keeping a keen eye toward why things work so that we can improve them. Thoughtworks supported this book in many ways and continues to grow Thoughtworkers who challenge and inspire me every day. I also thank our neighborhood cocktail club for a regular escape from routine, including the weekly outside, socially distanced versions that helped us all survive the odd time we just lived through. I thank my long-time friend Norman Zapien, who never ceases to provide enjoyable conversation. Lastly, I thank my wife, Candy, who continues to support this lifestyle that has me staring at things like book writing rather than our cats too much.

Acknowledgments from Pramod Sadalage

I thank my wife, Rupali, for all the support and understanding, and my lovely girls, Arula and Arhana, for the encouragement; daddy loves you both. All the work I do would not be possible without the clients I work with and various conferences that have helped me iterate on the concepts and content. I thank AvidXchange, the latest client I am working with, for its support and providing great space to iterate on new concepts. I also thank Thoughtworks for its continued support in my life, and Neal Ford, Rebecca Parsons, and Martin Fowler for being amazing mentors; you all make me a better person. Lastly, thank you to my parents, especially my mother, Shobha, whom I miss every day. *I miss you, MOM*.

Acknowledgments from Zhamak Dehghani

I thank Mark and Neal for their open invitation to contribute to this amazing body of work. My contribution to this book would not have been possible without the continuous support of my husband, Adrian, and patience of my daughter, Arianna. I love you both.

CHAPTER 1
What Happens When There Are No "Best Practices"?

Why does a technologist like a software architect present at a conference or write a book? Because they have discovered what is colloquially known as a "best practice," a term so overused that those who speak it increasingly experience backlash. Regardless of the term, technologists write books when they have figured out a novel solution to a general problem and want to broadcast it to a wider audience.

But what happens for that vast set of problems that have no good solutions? Entire classes of problems exist in software architecture that have no general good solutions, but rather present one messy set of trade-offs cast against an (almost) equally messy set.

Software developers build outstanding skills in searching online for solutions to a current problem. For example, if they need to figure out how to configure a particular tool in their environment, expert use of Google finds the answer.

But that's not true for architects.

For architects, many problems present unique challenges because they conflate the exact environment and circumstances of your organization—what are the chances that someone has encountered exactly this scenario *and* blogged it or posted it on Stack Overflow?

Architects may have wondered why so few books exist about architecture compared to technical topics like frameworks, APIs, and so on. Architects rarely experience common problems but constantly struggle with decision making in novel situations. For architects, every problem is a snowflake. In many cases, the problem is novel not just within a particular organization but rather throughout the world. No books or conference sessions exist for those problems!

Architects shouldn't constantly seek out silver-bullet solutions to their problems; they are as rare now as in 1986, when Fred Brooks coined the term:

> There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity.
>
> —Fred Brooks from "No Silver Bullet"

Because virtually every problem presents novel challenges, the real job of an architect lies in their ability to objectively determine and assess the set of trade-offs on either side of a consequential decision to resolve it as well as possible. The authors don't talk about "best solutions" (in this book or in the real world) because "best" implies that an architect has managed to maximize all the possible competing factors within the design. Instead, our tongue-in-cheek advice is as follows:

> Don't try to find the *best* design in software architecture; instead, strive for the *least worst* combination of trade-offs.

Often, the best design an architect can create is the least worst collection of trade-offs—no single architecture characteristics excels as it would alone, but the balance of all the competing architecture characteristics promote project success.

Which begs the question: "How can an architect *find* the least worst combination of trade-offs (and document them effectively)?" This book is primarily about decision making, enabling architects to make better decisions when confronted with novel situations.

Why "The Hard Parts"?

Why did we call this book *Software Architecture: The Hard Parts?* Actually, the "hard" in the title performs double duty. First, *hard* connotes *difficult*, and architects constantly face difficult problems that literally (and figuratively) no one has faced before, involving numerous technology decisions with long-term implications layered on top of the interpersonal and political environment where the decision must take place.

Second, *hard* connotes *solidity*—just as in the separation of *hardware* and *software*, the *hard* one should change much less because it provides the foundation for the *soft* stuff. Similarly, architects discuss the distinction between *architecture* and *design*, where the former is structural and the latter is more easily changed. Thus, in this book, we talk about the foundational parts of architecture.

The definition of software architecture itself has provided many hours of non-productive conversation among its practitioners. One favorite tongue-in-cheek definition is that "software architecture is the *stuff* that's hard to change later." That *stuff* is what our book is about.

Giving Timeless Advice About Software Architecture

The software development ecosystem constantly and chaotically shifts and grows. Topics that were all the rage a few years ago have either been subsumed by the ecosystem and disappeared or replaced by something different/better. For example, 10 years ago, the predominant architecture style for large enterprises was orchestration-driven, service-oriented architecture. Now, virtually no one builds in that architecture style anymore (for reasons we'll uncover along the way); the current favored style for many distributed systems is microservices. How and why did that transition happen?

When architects look at a particular style (especially a historical one), they must consider the constraints in place that lead to that architecture becoming dominant. At the time, many companies were merging to become *enterprises*, with all the attendant integration woes that come with that transition. Additionally, open source wasn't a viable option (often for political rather than technical reasons) for large companies. Thus, architects emphasized shared resources and centralized orchestration as a solution.

However, in the intervening years, open source and Linux became viable alternatives, making operating systems *commercially* free. However, the real tipping point occurred when Linux became *operationally* free with the advent of tools like Puppet and Chef, which allowed development teams to programmatically spin up their environments as part of an automated build. Once that capability arrived, it fostered an architectural revolution with microservices and the quickly emerging infrastructure of containers and orchestration tools like Kubernetes.

This illustrates that the software development ecosystem expands and evolves in completely unexpected ways. One new capability leads to another one, which unexpectedly creates new capabilities. Over the course of time, the ecosystem completely replaces itself, one piece at a time.

This presents an age-old problem for authors of books about technology generally and software architecture specifically—how can we write something that isn't old immediately?

We don't focus on technology or other implementation details in this book. Rather, we focus on *how* architects make decisions, and how to objectively weigh trade-offs when presented with novel situations. We use contemporaneous scenarios and examples to provide details and context, but the underlying principles focus on trade-off analysis and decision making when faced with new problems.

The Importance of Data in Architecture

> Data is a precious thing and will last longer than the systems themselves.
>
> —Tim Berners-Lee

For many in architecture, data is everything. Every enterprise building any system must deal with data, as it tends to live much longer than systems or architecture, requiring diligent thought and design. However, many of the instincts of data architects to build tightly coupled systems create conflicts within modern distributed architectures. For example, architects and DBAs must ensure that business data survives the breaking apart of monolith systems and that the business can still derive value from its data regardless of architecture undulations.

It has been said that *data is the most important asset in a company*. Businesses want to extract value from the data that they have and are finding new ways to deploy data in decision making. Every part of the enterprise is now data driven, from servicing existing customers, to acquiring new customers, increasing customer retention, improving products, predicting sales, and other trends. This reliance on data means that all software architecture is in the service of data, ensuring the right data is available and usable by all parts of the enterprise.

The authors built many distributed systems a few decades ago when they first became popular, yet decision making in modern microservices seems more difficult, and we wanted to figure out why. We eventually realized that, back in the early days of distributed architecture, we mostly still persisted data in a single relational database. However, in microservices and the philosophical adherence to a *bounded context* from Domain-Driven Design, as a way of limiting the scope of implementation detail coupling, data has moved to an architectural concern, along with transactionality. Many of the hard parts of modern architecture derive from tensions between data and architecture concerns, which we untangle in both Part I and Part II.

One important distinction that we cover in a variety of chapters is the separation between *operational* versus *analytical* data:

Operational data
: Data used for the operation of the business, including sales, transactional data, inventory, and so on. This data is what the company runs on—if something interrupts this data, the organization cannot function for very long. This type of data is defined as *Online Transactional Processing* (OLTP), which typically involves inserting, updating, and deleting data in a database.

Analytical data
: Data used by data scientists and other business analysts for predictions, trending, and other business intelligence. This data is typically not transactional and often not relational—it may be in a graph database or snapshots in a different format than its original transactional form. This data isn't critical for the day-to-day operation but rather for the long-term strategic direction and decisions.

We cover the impact of both operational and analytical data throughout the book.

Architectural Decision Records

One of the most effective ways of documenting architecture decisions is through *Architectural Decision Records* (ADRs). ADRs were first evangelized by Michael Nygard in a blog post and later marked as "adopt" in the Thoughtworks Technology Radar. An ADR consists of a short text file (usually one to two pages long) describing a specific architecture decision. While ADRs can be written using plain text, they are usually written in some sort of text document format like AsciiDoc or Markdown. Alternatively, an ADR can also be written using a wiki page template. We devoted an entire chapter to ADRs in our previous book, *Fundamentals of Software Architecture* (O'Reilly).

We will be leveraging ADRs as a way of documenting various architecture decisions made throughout the book. For each architecture decision, we will be using the following ADR format with the assumption that each ADR is approved:

> *ADR: A short noun phrase containing the architecture decision*
>
> *Context*
> In this section of the ADR we will add a short one- or two-sentence description of the problem, and list the alternative solutions.
>
> *Decision*
> In this section we will state the architecture decision and provide a detailed justification of the decision.
>
> *Consequences*
> In this section of the ADR we will describe any consequences after the decision is applied, and also discuss the trade-offs that were considered.

A list of all the Architectural Decision Records created in this book can be found in Appendix B.

Documenting a decision is important for an architect, but governing the proper use of the decision is a separate topic. Fortunately, modern engineering practices allow automating many common governance concerns by using architecture fitness functions.

Architecture Fitness Functions

Once an architect has identified the relationship between components and codified that into a design, how can they make sure that the implementers will adhere to that design? More broadly, how can architects ensure that the design principles they define become reality if they aren't the ones to implement them?

These questions fall under the heading of *architecture governance*, which applies to any organized oversight of one or more aspects of software development. As this book primarily covers architecture structure, we cover how to automate design and quality principles via fitness functions in many places.

Software development has slowly evolved over time to adapt unique engineering practices. In the early days of software development, a manufacturing metaphor was commonly applied to software practices, both in the large (like the Waterfall development process) and small (integration practices on projects). In the early 1990s, a rethinking of software development engineering practices, lead by Kent Beck and the other engineers on the C3 project, called eXtreme Programming (XP), illustrated the importance of incremental feedback and automation as key enablers of software development productivity. In the early 2000s, the same lessons were applied to the intersection of software development and operations, spawning the new role of DevOps and automating many formerly manual operational chores. Just as before, automation allows teams to go faster because they don't have to worry about things breaking without good feedback. Thus, *automation* and *feedback* have become central tenets for effective software development.

Consider the environments and situations that lead to breakthroughs in automation. In the era before continuous integration, most software projects included a lengthy integration phase. Each developer was expected to work in some level of isolation from others, then integrate all the code at the end into an integration phase. Vestiges of this practice still linger in version control tools that force branching and prevent continuous integration. Not surprisingly, a strong correlation existed between project size and the pain of the integration phase. By pioneering continuous integration, the XP team illustrated the value of rapid, continuous feedback.

The DevOps revolution followed a similar course. As Linux and other open source software became "good enough" for enterprises, combined with the advent of tools that allowed programmatic definition of (eventually) virtual machines, operations personnel realized they could automate machine definitions and many other repetitive tasks.

In both cases, advances in technology and insights led to automating a recurring job that was handled by an expensive role—which describes the current state of architecture governance in most organizations. For example, if an architect chooses a particular architecture style or communication medium, how can they make sure that a developer implements it correctly? When done manually, architects perform code reviews or perhaps hold architecture review boards to assess the state of governance. However, just as in manually configuring computers in operations, important details can easily fall through superficial reviews.

Using Fitness Functions

In the 2017 book *Building Evolutionary Architectures* (O'Reilly), the authors (Neal Ford, Rebecca Parsons, and Patrick Kua) defined the concept of an *architectural fitness function*: any mechanism that performs an objective integrity assessment of some architecture characteristic or combination of architecture characteristics. Here is a point-by-point breakdown of that definition:

Any mechanism
: Architects can use a wide variety of tools to implement fitness functions; we will show numerous examples throughout the book. For example, dedicated testing libraries exist to test architecture structure, architects can use monitors to test operational architecture characteristics such as performance or scalability, and chaos engineering frameworks test reliability and resiliency.

Objective integrity assessment
: One key enabler for automated governance lies with objective definitions for architecture characteristics. For example, an architect can't specify that they want a "high performance" website; they must provide an object value that can be measured by a test, monitor, or other fitness function.

  Architects must watch out for *composite architecture characteristics*—ones that aren't objectively measurable but are really composites of other measurable things. For example, "agility" isn't measurable, but if an architect starts pulling the broad term *agility* apart, the goal is for teams to be able to respond quickly and confidently to change, either in ecosystem or domain. Thus, an architect can find measurable characteristics that contribute to agility: deployability, testability, cycle time, and so on. Often, the lack of ability to measure an architecture characteristic indicates too vague a definition. If architects strive toward measurable properties, it allows them to automate fitness function application.

Some architecture characteristic or combination of architecture characteristics
: This characteristic describes the two scopes for fitness functions:

  Atomic
  : These fitness functions handle a single architecture characteristic in isolation. For example, a fitness function that checks for component cycles within a codebase is atomic in scope.

Holistic

: *Holistic* fitness functions validate a combination of architecture characteristics. A complicating feature of architecture characteristics is the synergy they sometimes exhibit with other architecture characteristics. For example, if an architect wants to improve security, a good chance exists that it will affect performance. Similarly, scalability and elasticity are sometimes at odds—supporting a large number of concurrent users can make handling sudden bursts more difficult. Holistic fitness functions exercise a combination of interlocking architecture characteristics to ensure that the combined effect won't negatively affect the architecture.

An architect implements fitness functions to build protections around unexpected change in architecture characteristics. In the Agile software development world, developers implement unit, functional, and user acceptance tests to validate different dimensions of the *domain* design. However, until now, no similar mechanism existed to validate the *architecture characteristics* part of the design. In fact, the separation between fitness functions and unit tests provides a good scoping guideline for architects. Fitness functions validate architecture characteristics, not domain criteria; unit tests are the opposite. Thus, an architect can decide whether a fitness function or unit test is needed by asking the question: "Is any domain knowledge required to execute this test?" If the answer is "yes," then a unit/function/user acceptance test is appropriate; if "no," then a fitness function is needed.

For example, when architects talk about *elasticity*, it's the ability of the application to withstand a sudden burst of users. Notice that the architect doesn't need to know any details about the domain—this could be an ecommerce site, an online game, or something else. Thus, *elasticity* is an architectural concern and within the scope of a fitness function. If on the other hand the architect wanted to validate the proper parts of a mailing address, that is covered via a traditional test. Of course, this separation isn't purely binary—some fitness functions will touch on the domain and vice versa, but the differing goals provide a good way to mentally separate them.

Here are a couple of examples to make the concept less abstract.

One common architect goal is to maintain good internal structural integrity in the codebase. However, malevolent forces work against the architect's good intentions on many platforms. For example, when coding in any popular Java or .NET development environment, as soon as a developer references a class not already imported, the IDE helpfully presents a dialog asking the developer if they would like to auto-import the reference. This occurs so often that most programmers develop the habit of swatting the auto-import dialog away like a reflex action.

However, arbitrarily importing classes or components among one another spells disaster for modularity. For example, Figure 1-1 illustrates a particularly damaging antipattern that architects aspire to avoid.

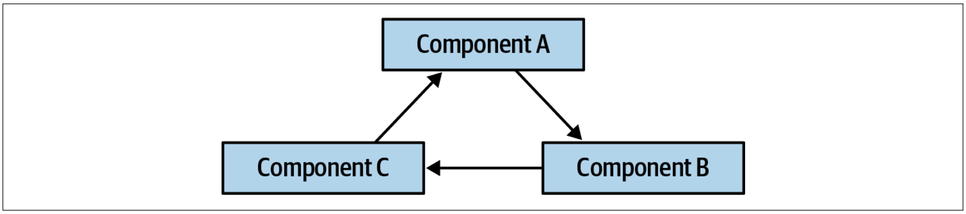


Figure 1-1. Cyclic dependencies between components

In this anti-pattern, each component references something in the others. Having a network of components such as this damages modularity because a developer cannot reuse a single component without also bringing the others along. And, of course, if the other components are coupled to other components, the architecture tends more and more toward the Big Ball of Mud anti-pattern. How can architects govern this behavior without constantly looking over the shoulders of trigger-happy developers? Code reviews help but happen too late in the development cycle to be effective. If an architect allows a development team to rampantly import across the codebase for a week until the code review, serious damage has already occurred in the codebase.

The solution to this problem is to write a fitness function to avoid component cycles, as shown in Example 1-1.

Example 1-1. Fitness function to detect component cycles

```
public class CycleTest {
    private JDepend jdepend;

    @BeforeEach
    void init() {
          jdepend = new JDepend();
          jdepend.addDirectory("/path/to/project/persistence/classes");
          jdepend.addDirectory("/path/to/project/web/classes");
          jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
          Collection packages = jdepend.analyze();
          assertEquals("Cycles exist", false, jdepend.containsCycles());
    }
}
```

In the code, an architect uses the metrics tool JDepend to check the dependencies between packages. The tool understands the structure of Java packages and fails the test if any cycles exist. An architect can wire this test into the continuous build on a project and stop worrying about the accidental introduction of cycles by trigger-happy developers. This is a great example of a fitness function guarding the important rather than urgent practices of software development: it's an important concern for architects, yet has little impact on day-to-day coding.

Example 1-1 shows a very low-level, code-centric fitness function. Many popular code hygiene tools (such as SonarQube) implement many common fitness functions in a turnkey manner. However, architects may also want to validate the macro structure of the architecture as well as the micro. When designing a layered architecture such as the one in Figure 1-2, the architect defines the layers to ensure separation of concerns.

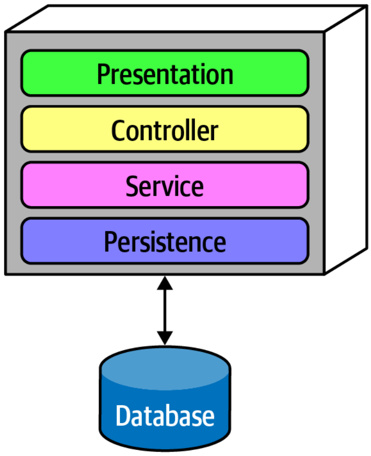


Figure 1-2. Traditional layered architecture

However, how can the architect ensure that developers will respect these layers? Some developers may not understand the importance of the patterns, while others may adopt a "better to ask forgiveness than permission" attitude because of some overriding local concern, such as performance. But allowing implementers to erode the reasons for the architecture hurts the long-term health of the architecture.

ArchUnit allows architects to address this problem via a fitness function, shown in Example 1-2.

Example 1-2. ArchUnit fitness function to govern layers

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

In Example 1-2, the architect defines the desirable relationship between layers and writes a verification fitness function to govern it. This allows an architect to establish architecture principles outside the diagrams and other informational artifacts, and verify them on an ongoing basis.

A similar tool in the .NET space, NetArchTest, allows similar tests for that platform. A layer verification in C# appears in Example 1-3.

Example 1-3. NetArchTest for layer dependencies

```
// Classes in the presentation should not directly reference repositories
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

Tools continue to appear in this space with increasing degrees of sophistication. We will continue to highlight many of these techniques as we illustrate fitness functions alongside many of our solutions.

Finding an objective outcome for a fitness function is critical. However, *objective* doesn't imply *static*. Some fitness functions will have noncontextual return values, such as true/false or a numeric value such as a performance threshold. However, other fitness functions (deemed *dynamic*) return a value based on some context. For example, when measuring *scalability*, architects measure the number of concurrent users and also generally measure the performance for each user. Often, architects design systems so that as the number of users goes up, performance per user declines slightly—but doesn't fall off a cliff. Thus, for these systems, architects design performance fitness functions that take into account the number of concurrent users. As long as the measure of an architecture characteristic is objective, architects can test it.

While most fitness functions should be automated and run continually, some will necessarily be manual. A manual fitness function requires a person to handle the validation. For example, for systems with sensitive legal information, a lawyer may need to review changes to critical parts to ensure legality, which cannot be automated. Most deployment pipelines support manual stages, allowing teams to accommodate manual fitness functions. Ideally, these are run as often as reasonably possible—a validation that doesn't run can't validate anything. Teams execute fitness functions either on demand (rarely) or as part of a continuous integration work stream (most common). To fully achieve the benefit of validations such as fitness functions, they should be run continually.

Continuity is important, as illustrated in this example of enterprise-level governance using fitness functions. Consider the following scenario: what does a company do when a zero-day exploit is discovered in one of the development frameworks or libraries the enterprise uses? If it's like most companies, security experts scour projects to find the offending version of the framework and make sure it's updated, but that process is rarely automated, relying on many manual steps. This isn't an abstract question; this exact scenario affected a major financial institution described in The Equifax Data Breach. Like the architecture governance described previously, manual processes are error prone and allow details to escape.

The Equifax Data Breach

On September 7, 2017, Equifax, a major credit scoring agency in the US, announced that a data breach had occurred. Ultimately, the problem was traced to a hacking exploit of the popular Struts web framework in the Java ecosystem (Apache Struts vCVE-2017-5638). The foundation issued a statement announcing the vulnerability and released a patch on March 7, 2017. The Department of Homeland Security contacted Equifax and similar companies the next day, warning them of this problem, and they ran scans on March 15, 2017, which didn't reveal all of the affected systems. Thus, the critical patch wasn't applied to many older systems until July 29, 2017, when Equifax's security experts identified the hacking behavior that lead to the data breach.

Imagine an alternative world in which every project runs a deployment pipeline, and the security team has a "slot" in each team's deployment pipeline where they can deploy fitness functions. Most of the time, these will be mundane checks for safeguards like preventing developers from storing passwords in databases and similar regular governance chores. However, when a zero-day exploit appears, having the same mechanism in place everywhere allows the security team to insert a test in every project that checks for a certain framework and version number; if it finds the dangerous version, it fails the build and notifies the security team. Teams configure deployment pipelines to awaken for any change to the ecosystem: code, database schema, deployment configuration, and fitness functions. This allows enterprises to universally automate important governance tasks.

Fitness functions provide many benefits for architects, not the least of which is the chance to do some coding again! One of the universal complaints among architects is that they don't get to code much anymore—but fitness functions are often code! By building an executable specification of the architecture, which anyone can validate anytime by running the project's build, architects must understand the system and its ongoing evolution well, which overlaps with the core goal of keeping up with the code of the project as it grows.

However powerful fitness functions are, architects should avoid overusing them. Architects should not form a cabal and retreat to an ivory tower to build an impossibly complex, interlocking set of fitness functions that merely frustrate developers and teams. Instead, it's a way for architects to build an executable checklist of *important* but not *urgent* principles on software projects. Many projects drown in urgency, allowing some important principles to slip by the side. This is the frequent cause of technical debt: "We know this is bad, but we'll come back to fix it later"—and later never comes. By codifying rules about code quality, structure, and other safeguards against decay into fitness functions that run continually, architects build a quality checklist that developers can't skip.

A few years ago, the excellent book *The Checklist Manifesto* by Atul Gawande (Picador) highlighted the use of checklists by professionals like surgeons, airline pilots, and those other fields who commonly use (sometimes by force of law) checklists as part of their job. It isn't because they don't know their job or are particularly forgetful; when professionals perform the same task over and over, it becomes easy to fool themselves when it's accidentally skipped, and checklists prevent that. Fitness functions represent a checklist of important principles defined by architects and run as part of the build to make sure developers don't accidentally (or purposefully, because of external forces like schedule pressure) skip them.

We utilize fitness functions throughout the book when an opportunity arises to illustrate governing an architectural solution as well as the initial design.

Architecture Versus Design: Keeping Definitions Simple

A constant area of struggle for architects is keeping *architecture* and *design* as separate but related activities. While we don't want to wade into the never-ending argument about this distinction, we strive in this book to stay firmly on the *architecture* side of that spectrum for several reasons.

First, architects must understand underlying architecture principles to make effective decisions. For example, the decision between *synchronous* versus *asynchronous* communication has a number of trade-offs before architects layer in implementation details. In the book *Fundamentals of Software Architecture*, the authors coined the second law of software architecture: *why* is more important than *how*. While ultimately architects must understand how to implement solutions, they must first understand why one choice has better trade-offs than another.

Second, by focusing on architecture concepts, we can avoid the numerous implementations of those concepts. Architects can implement asynchronous communication in a variety of ways; we focus on why an architect would choose asynchronous communication and leave the implementation details to another place.

Third, if we start down the path of implementing all the varieties of options we show, this would be the longest book ever written. Focus on architecture principles allows us to keep things as generic as they can be.

To keep subjects as grounded in architecture as possible, we use the simplest definitions possible for key concepts. For example, *coupling* in architecture can fill entire books (and it has). To that end, we use the following simple, verging on simplistic, definitions:

Service
: In colloquial terms, a *service* is a cohesive collection of functionality deployed as an independent executable. Most of the concepts we discuss with regard to services apply broadly to distributed architectures, and specifically microservices architectures.

  In the terms we define in Chapter 2, a *service* is part of an architecture quantum, which includes further definitions of both static and dynamic coupling between services and other quanta.

Coupling
: Two artifacts (including services) are coupled if a change in one might require a change in the other to maintain proper functionality.

Component
: An architectural building block of the application that does some sort of business or infrastructure function, usually manifested through a package structure (Java), namespace (C#), or a physical grouping of source code files within some sort of directory structure. For example, the component Order History might be implemented through a set of class files located in the namespace `app.business.order.history`.

Synchronous communication
: Two artifacts communicate synchronously if the caller must wait for the response before proceeding.

Asynchronous communication
: Two artifacts communicate asynchronously if the caller does not wait for the response before proceeding. Optionally, the caller can be notified by the receiver through a separate channel when the request has completed.

Orchestrated coordination
: A workflow is orchestrated if it includes a service whose primary responsibility is to coordinate the workflow.

Choreographed coordination
: A workflow is choreographed when it lacks an orchestrator; rather, the services in the workflow share the coordination responsibilities of the workflow.

Atomicity
: A workflow is *atomic* if all parts of the workflow maintain a consistent state at all times; the opposite is represented by the spectrum of *eventual consistency*, covered in Chapter 6.

Contract
: We use the term *contract* broadly to define the interface between two software parts, which may encompass method or function calls, integration architecture remote calls, dependencies, and so on. Anywhere two pieces of software join, a contract is involved.

Software architecture is by its nature abstract: we cannot know what unique combination of platforms, technologies, commercial software, and the other dizzying array of possibilities our readers might have, except that no two are exactly alike. We cover many abstract ideas, but must ground them with some implementation details to make them concrete. To that end, we need a problem to illustrate architecture concepts against—which leads us to the Sysops Squad.

Introducing the Sysops Squad Saga

> *saga*
> A long story of heroic achievement.
>
> —Oxford English Dictionary

We discuss a number of sagas in this book, both literal and figurative. Architects have co-opted the term *saga* to describe transactional behavior in distributed architectures (which we cover in detail in Chapter 12). However, discussions about architecture tend to become abstract, especially when considering abstract problems such as the hard parts of architecture. To help solve this problem and provide some real-world context for the solutions we discuss, we kick off a literal saga about the *Sysops Squad*.

We use the Sysops Squad saga within each chapter to illustrate the techniques and trade-offs described in this book. While many books on software architecture cover new development efforts, many real-world problems exist within existing systems. Therefore, our story starts with the existing Sysops Squad architecture highlighted here.

Penultimate Electronics is a large electronics giant that has numerous retail stores throughout the country. When customers buy computers, TVs, stereos, and other electronic equipment, they can choose to purchase a support plan. When problems occur, customer-facing technology experts (the Sysops Squad) come to the customer's residence (or work office) to fix problems with the electronic device.

The four main users of the Sysops Squad ticketing application are as follows:

Administrator
: The administrator maintains the internal users of the system, including the list of experts and their corresponding skill set, location, and availability. The administrator also manages all of the billing processing for customers using the system, and maintains static reference data (such as supported products, name-value pairs in the system, and so on).

Customer
: The customer registers for the Sysops Squad service and maintains their customer profile, support contracts, and billing information. Customers enter problem tickets into the system, and also fill out surveys after the work has been completed.

Sysops Squad expert
: Experts are assigned problem tickets and fix problems based on the ticket. They also interact with the knowledge base to search for solutions to customer problems and enter notes about repairs.

Manager
: The manager keeps track of problem ticket operations and receives operational and analytical reports about the overall Sysops Squad problem ticket system.

Nonticketing Workflow

The nonticketing workflows include those actions that administrators, managers, and customers perform that do not relate to a problem ticket. These workflows are outlined as follows:

1. Sysops Squad experts are added and maintained in the system through an administrator, who enters in their locale, availability, and skills.
2. Customers register with the Sysops Squad system and have multiple support plans based on the products they purchased.
3. Customers are automatically billed monthly based on credit card information contained in their profile. Customers can view billing history and statements through the system.
4. Managers request and receive various operational and analytical reports, including financial reports, expert performance reports, and ticketing reports.

Ticketing Workflow

The ticketing workflow starts when a customer enters a problem ticket into the system, and ends when the customer completes the survey after the repair is done. This workflow is outlined as follows:

1. Customers who have purchased the support plan enter a problem ticket by using the Sysops Squad website.
2. Once a problem ticket is entered in the system, the system then determines which Sysops Squad expert would be the best fit for the job based on skills, current location, service area, and availability.
3. Once assigned, the problem ticket is uploaded to a dedicated custom mobile app on the Sysops Squad expert's mobile device. The expert is also notified via a text message that they have a new problem ticket.
4. The customer is notified through an SMS text message or email (based on their profile preference) that the expert is on their way.
5. The expert uses the custom mobile application on their phone to retrieve the ticket information and location. The Sysops Squad expert can also access a knowledge base through the mobile app to find out what has been done in the past to fix the problem.
6. Once the expert fixes the problem, they mark the ticket as "complete." The sysops squad expert can then add information about the problem and repair the knowledge base.
7. After the system receives notification that the ticket is complete, it sends an email to the customer with a link to a survey, which the customer then fills out.
8. The system receives the completed survey from the customer and records the survey information.

A Bad Scenario

Things have not been good with the Sysops Squad problem ticket application lately. The current trouble ticket system is a large monolithic application that was developed many years ago. Customers are complaining that consultants are never showing up because of lost tickets, and often the wrong consultant shows up to fix something they know nothing about. Customers have also been complaining that the system is not always available to enter new problem tickets.

Change is also difficult and risky in this large monolith. Whenever a change is made, it usually takes too long and something else usually breaks. Because of reliability issues, the Sysops Squad system frequently "freezes up," or crashes, resulting in all application functionality not being available anywhere from five minutes to two hours while the problem is identified and the application restarted.

If something isn't done soon, Penultimate Electronics will be forced to abandon the very lucrative support contract business line and lay off all the Sysops Squad administrators, experts, managers, and IT development staff—including the architects.

Sysops Squad Architectural Components

The monolithic system for the Sysops Squad application handles ticket management, operational reporting, customer registration, and billing, as well as general administrative functions such as user maintenance, login, and expert skills and profile maintenance. Figure 1-3 and the corresponding Table 1-1 illustrate and describe the components of the existing monolithic application (the `ss.` part of the namespace specifies the Sysops Squad application context).

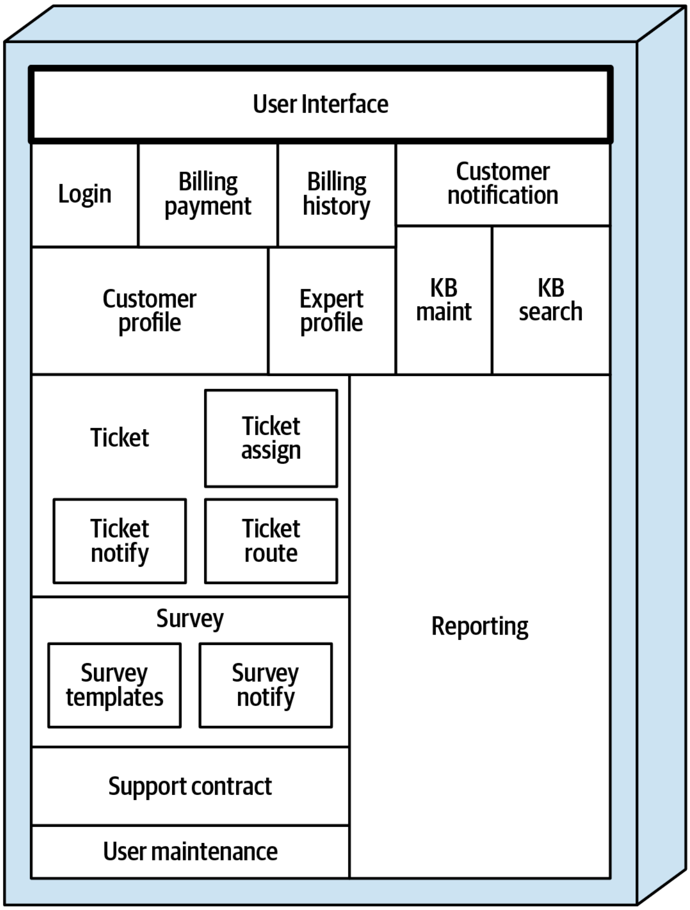


Figure 1-3. Components within the existing Sysops Squad application

Table 1-1. Existing Sysops Squad components

Component	Namespace	Responsibility
Login	`ss.login`	Internal user and customer login and security logic
Billing payment	`ss.billing.payment`	Customer monthly billing and customer credit card info
Billing history	`ss.billing.history`	Payment history and prior billing statements
Customer notification	`ss.customer.notification`	Notify customer of billing, general info
Customer profile	`ss.customer.profile`	Maintain customer profile, customer registration
Expert profile	`ss.expert.profile`	Maintain expert profile (name, location, skills, etc.)
KB maint	`ss.kb.maintenance`	Maintain and view items in the knowledge base
KB search	`ss.kb.search`	Query engine for searching the knowledge base
Reporting	`ss.reporting`	All reporting (experts, tickets, financial)
Ticket	`ss.ticket`	Ticket creation, maintenance, completion, common code
Ticket assign	`ss.ticket.assign`	Find an expert and assign the ticket
Ticket notify	`ss.ticket.notify`	Notify customer that the expert is on their way
Ticket route	`ss.ticket.route`	Send the ticket to the expert's mobile device app
Support contract	`ss.supportcontract`	Support contracts for customers, products in the plan
Survey	`ss.survey`	Maintain surveys, capture and record survey results
Survey notify	`ss.survey.notify`	Send survey email to customer
Survey templates	`ss.survey.templates`	Maintain various surveys based on type of service
User maintenance	`ss.users`	Maintain internal users and roles

These components will be used in subsequent chapters to illustrate various techniques and trade-offs when dealing with breaking applications into distributed architectures.

Sysops Squad Data Model

The Sysops Squad application with its various components listed in Table 1-1 uses a single schema in the database to host all its tables and related database code. The database is used to persist customers, users, contracts, billing, payments, knowledge base, and customer surveys; the tables are listed in Table 1-2, and the ER model is illustrated in Figure 1-4.

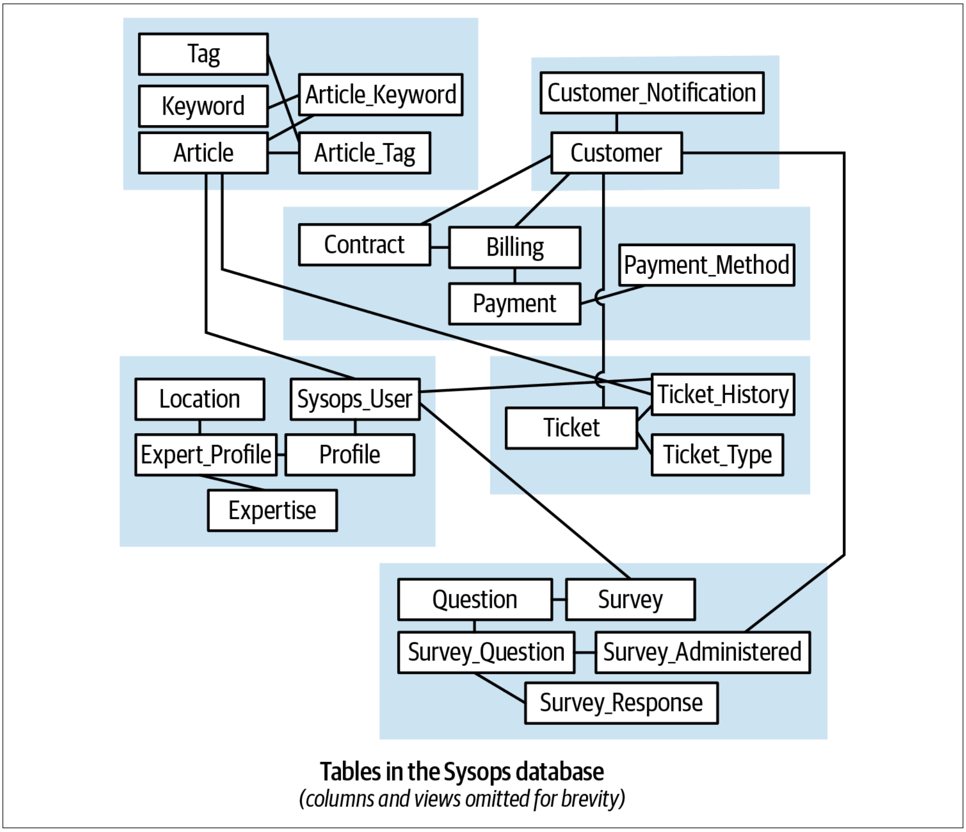


Figure 1-4. Data model within the existing Sysops Squad application

Table 1-2. Existing Sysops Squad database tables

Table	Responsibility
Customer	Entities needing Sysops support
Customer_Notification	Notification preferences for customers
Survey	A survey for after-support customer satisfaction
Question	Questions in a survey
Survey_Question	A question is assigned to the survey
Survey_Administered	Survey question is assigned to customer
Survey_Response	A customer's response to the survey
Billing	Billing information for support contract
Contract	A contract between an entity and Sysops for support
Payment_Method	Payment methods supported for making payment
Payment	Payments processed for billings

Table	Responsibility
SysOps_User	The various users in Sysops
Profile	Profile information for Sysops users
Expert_Profile	Profiles of experts
Expertise	Various expertise within Sysops
Location	Locations served by the expert
Article	Articles for the knowledge base
Tag	Tags on articles
Keyword	Keyword for an article
Article_Tag	Tags associated to articles
Article_Keyword	Join table for keywords and articles
Ticket	Support tickets raised by customers
Ticket_Type	Different types of tickets
Ticket_History	The history of support tickets

The Sysops data model is a standard third normal form data model with only a few stored procedures or triggers. However, a fair number of views exist that are mainly used by the Reporting component. As the architecture team tries to break up the application and move toward distributed architecture, it will have to work with the database team to accomplish the tasks at the database level. This setup of database tables and views will be used throughout the book to discuss various techniques and trade-offs to accomplish the task of breaking apart the database.

PART I
Pulling Things Apart

As many of us discovered when we were children, a great way to understand how something fits together is to first pull it apart. To understand complex subjects (such as trade-offs in distributed architectures), an architect must figure out where to start untangling.

In the book *What Every Programmer Should Know About Object-Oriented Design* (Dorset House), Meilir Page-Jones made the astute observation that coupling in architecture may be split into static and dynamic coupling. *Static* coupling refers to the way architectural parts (classes, components, services, and so on) are *wired* together: dependencies, coupling degree, connection points, and so on. An architect can often measure static coupling at compile time as it represents the static dependencies within the architecture.

Dynamic coupling refers to how architecture parts *call* one another: what kind of communication, what information is passed, strictness of contracts, and so on.

Our goal is to investigate how to do trade-off analysis in distributed architectures; to do that, we must pull the moving pieces apart so that we can discuss them in isolation to understand them fully before putting them back together.

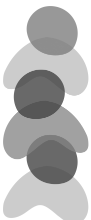
Part I primarily deals with *architectural structure*, how things are statically coupled together. In Chapter 2, we tackle the problem of defining the scope of static and dynamic coupling in architectures, and present the entire picture that we must pull apart to understand. Chapter 3 begins that process, defining modularity and separation in architecture. Chapter 4 provides tools to evaluate and deconstruct codebases, and Chapter 5 supplies patterns to assist the process.

Data and transactions have become increasingly important in architecture, driving many trade-off decisions by architects and DBAs. Chapter 6 addresses the architectural impacts of data, including how to reconcile service and data boundaries. Finally, Chapter 7 ties together architecture coupling with data concerns to define *integrators* and *disintegrators*—forces that encourage a larger or smaller service size and boundary.

CHAPTER 2

Discerning Coupling in Software Architecture

`Wednesday, November 3, 13:00`



Logan, the lead architect for Penultimate Electronics, interrupted a small group of architects in the cafeteria, discussing distributed architectures. “Austen, are you wearing a cast *again*?”

“No, it’s just a splint,” replied Austen. “I sprained my wrist playing extreme disc golf over the weekend—it’s almost healed.”

“What is…never mind. What is this impassioned conversation I barged in on?”

“Why wouldn’t someone always choose the *saga pattern* in microservices to wire together transactions?” asked Austen. “That way, architects can make the services as small as they want.”

“But don’t you have to use *orchestration* with sagas?” asked Addison. “What about times when we need asynchronous communication? And, how complex will the transactions get? If we break things down too much, can we really guarantee data fidelity?”

“You know,” said Austen, “if we use an *enterprise service bus*, we can get it to manage most of that stuff for us.”

“I thought no one used ESBs anymore—shouldn’t we use Kafka for stuff like that?”

“They aren’t even the same thing!” said Austen.

Logan interrupted the increasingly heated conversation. “It is an apples-to-oranges comparison, but none of these tools or approaches is a silver bullet. Distributed architectures like microservices are difficult, especially if architects cannot untangle all the forces at play. What we need is an approach or framework that helps us figure out the hard problems in our architecture.”

"Well," said Addison, "whatever we do, it has to be as decoupled as possible—everything I've read says that architects should embrace decoupling as much as possible."

"If you follow that advice," said Logan, "Everything will be so decoupled that nothing can communicate with anything else—it's hard to build software that way! Like a lot of things, coupling isn't inherently bad; architects just have to know how to apply it appropriately. In fact, I remember a famous quote about that from a philosopher…."

> All things are poison, and nothing is without poison; the dosage alone makes it so a thing is not a poison.
>
> —Paracelsus

One of the most difficult tasks an architect will face is untangling the various forces and trade-offs at play in distributed architectures. People who provide advice constantly extol the benefits of "loosely coupled" systems, but how can architects design systems where nothing connects to anything else? Architects design fine-grained microservices to achieve decoupling, but then orchestration, transactionality, and asynchronicity become huge problems. Generic advice says "decouple," but provides no guidelines for *how* to achieve that goal while still constructing useful systems.

Architects struggle with granularity and communication decisions because there are no clear universal guides for making decisions—no best practices exist that can apply to real-world complex systems. Until now, architects lacked the correct perspective and terminology to allow a careful analysis that could determine the best (or least worst) set of trade-offs on a case-by-case basis.

Why have architects struggled with decisions in distributed architectures? After all, we've been building distributed systems since the last century, using many of the same mechanisms (message queues, events, and so on). Why has the complexity ramped up so much with microservices?

The answer lies with the fundamental philosophy of microservices, inspired by the idea of a *bounded context*. Building services that model bounded contexts required a subtle but important change to the way architects designed distributed systems because now transactionality is a first-class architectural concern. In many of the distributed systems architects designed prior to microservices, event handlers typically connected to a single relational database, allowing it to handle details such as integrity and transactions. Moving the database within the service boundary moves data concerns into architecture concerns.

As we've said before, *"Software architecture" is the stuff you can't Google answers for*. A skill that modern architects must build is the ability to do trade-off analysis. While several frameworks have existed for decades (such as Architecture Trade-off Analysis Method, or ATAM), they lack focus on real problems architects face on a daily basis.

This book focuses on how architects can perform trade-off analysis for any number of scenarios unique to their situation. As in many things in architecture, the advice is simple; the hard parts lie in the details, particularly how difficult parts become entangled, making it difficult to see and understand the individual parts, as illustrated in Figure 2-1.

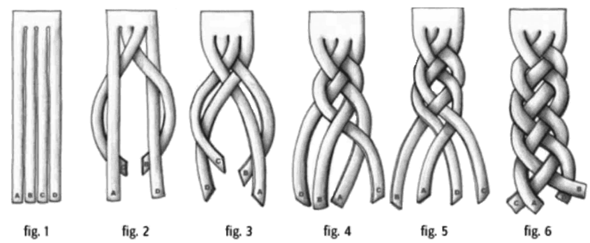


Figure 2-1. A braid entangles hair, making the individual strands hard to identify

When architects look at entangled problems, they struggle with performing trade-off analysis because of the difficulties separating the concerns, so that they may consider them independently. Thus, the first step in trade-off analysis is untangle the dimensions of the problem, analyzing what parts are coupled to one another and what impact that coupling has on change. For this purpose, we use the simplest definition of the word *coupling*:

Coupling
: Two parts of a software system are coupled if a change in one might cause a change in the other.

Often, software architecture creates multidimensional problems, where multiple forces all interact in interdependent ways. To analyze trade-offs, an architect must first determine what forces need to trade off with each other.

Thus, here's our advice for modern trade-off analysis in software architecture:

1. Find what parts are entangled together.
2. Analyze how they are coupled to one another.
3. Assess trade-offs by determining the impact of change on interdependent systems.

While the steps are simple, the hard parts lurk in the details. Thus, to illustrate this framework in practice, we take one of the most difficult (and probably the closest to generic) problems in distributed architectures, which is related to microservices:

How do architects determine the size and communication styles for microservices?
: Determining the proper size for microservices seems a pervasive problem—too-small services create transactional and orchestration issues, and too-large services create scale and distribution issues.

To that end, the remainder of this book untangles the many aspects to consider when answering the preceding question. We provide new terminology to differentiate similar but distinct patterns and show practical examples of applying these and other patterns.

However, the overarching goal of this book is to provide you with example-driven techniques to learn how to construct your own trade-off analysis for the unique problems within your realm. We start with our first great untangling of forces in distributed architectures: defining architecture quantum along with the two types of coupling, static and dynamic.

Architecture (Quantum | Quanta)

The term *quantum* is, of course, used heavily in the field of physics known as *quantum mechanics*. However, the authors chose the word for the same reasons physicists did. *Quantum* originated from the Latin word *quantus*, meaning "how great" or "how many." Before physics co-opted it, the legal profession used it to represent the "required or allowed amount" (for example, in damages paid). The term also appears in the mathematics field of topology, concerning the properties of families of shapes. Because of its Latin roots, the singular is *quantum*, and the plural is *quanta*, similar to the datum/data symmetry.

An architecture quantum measures several aspects of both topology and behavior in software architecture related to how parts connect and communicate with one another:

Architecture quantum
: An architecture quantum is an independently deployable artifact with high functional cohesion, high static coupling, and synchronous dynamic coupling. A common example of an architecture quantum is a well-formed microservice within a workflow.

Static coupling
: Represents how static dependencies resolve within the architecture via contracts. These dependencies include operating system, frameworks, and/or libraries delivered via transitive dependency management, and any other operational requirement to allow the quantum to operate.

Dynamic coupling

Represents how quanta communicate at runtime, either synchronously or asynchronously. Thus, fitness functions for these characteristics must be *continuous*, typically utilizing monitors.

Even though both static and dynamic coupling seem similar, architects must distinguish two important differences. An easy way to think about the difference is that *static coupling* describes how services are *wired* together, whereas *dynamic coupling* describes how services *call* one another at runtime. For example, in a microservices architecture, a service must contain dependent components such as a database, representing static coupling—the service isn't operational without the necessary data. That service may call other services during the course of a workflow, which represents dynamic coupling. Neither service requires the other to be present to function, except for this runtime workflow. Thus, static coupling analyzes operational dependencies, and dynamic coupling analyzes communication dependencies.

These definitions include important characteristics; let's cover each in detail as they inform most of the examples in the book.

Independently Deployable

Independently deployable implies several aspects of an architecture quantum—each quantum represents a separate deployable unit within a particular architecture. Thus, a monolithic architecture—one that is deployed as a single unit—is by definition a single architecture quantum. Within a distributed architecture such as microservices, developers tend toward the ability to deploy services independently, often in a highly automated way. Thus, from an independently deployable standpoint, a service within a microservices architecture represents an architecture quantum (contingent on coupling—as discussed next).

Making each architecture quantum represent a deployable asset within the architecture serves several useful purposes. First, the boundary represented by an architecture quantum serves as a useful common language among architects, developers, and operations. Each understands the common scope under question: architects understand the coupling characteristics, developers understand the scope of behavior, and the operations team understands the deployable characteristics.

Second, the architecture quantum represents one of the forces (static coupling) architects must consider when striving for proper granularity of services within a distributed architecture. Often, in microservices architectures, developers face the difficult question of what service granularity offers the optimum set of trade-offs. Some of those trade-offs revolve around deployability: what release cadence does this service require, what other services might be affected, what engineering practices are involved, and so on. Architects benefit from a firm understanding of exactly where