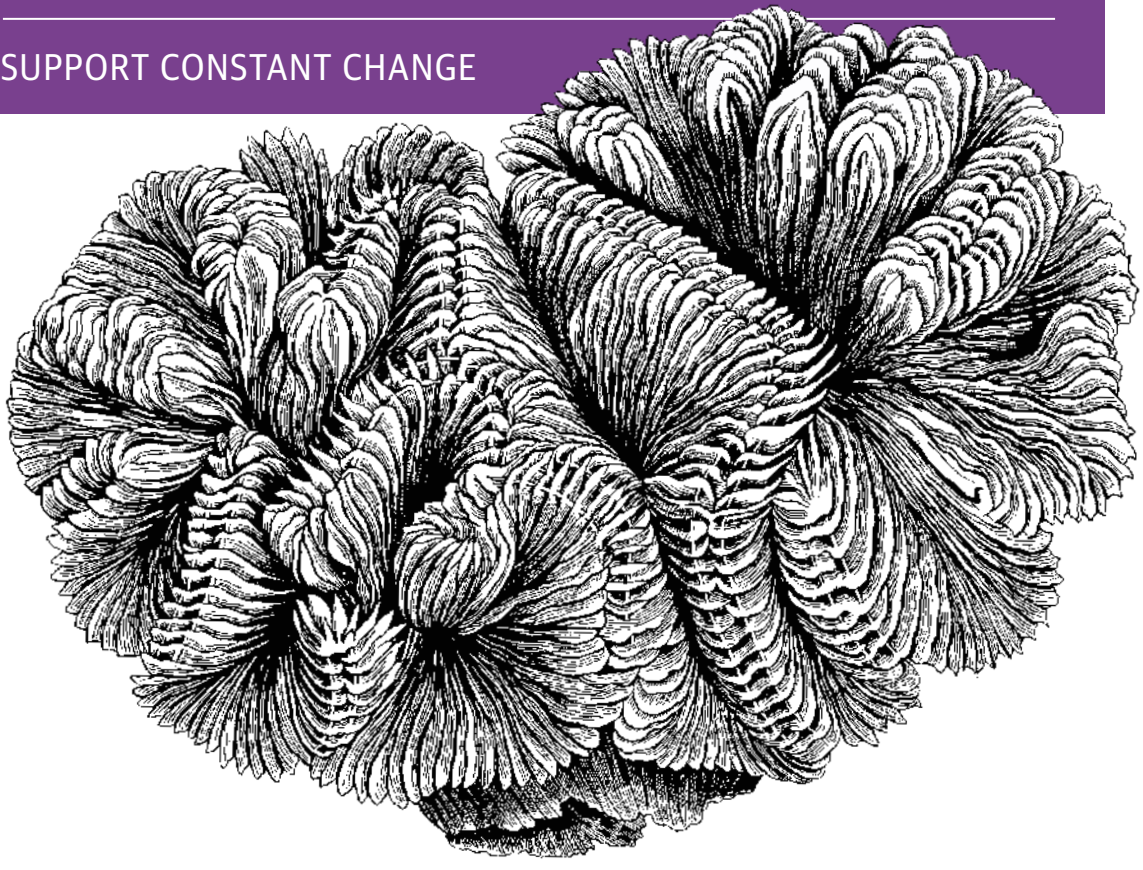
O'REILLY®

# Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE



Neal Ford, Rebecca Parsons & Patrick Kua

O'REILLY®

# Building Evolutionary Architectures

Enterprise architects can no longer rely on static planning. The software development ecosystem is constantly changing, providing a continuous stream of new tools, frameworks, techniques, and paradigms. This creates headaches for people with brittle systems, but also provides the ultimate solution. In recent years, incremental developments in core software engineering practices created the foundations for rethinking how architecture changes over time. This book ties those practices together, and offers a new way to think about the intersection of *architecture* and *time*.

Building an evolutionary architecture consists of three primary concerns: fitness functions, incremental change, and appropriate coupling. Three specialists at ThoughtWorks examine each aspect separately, and then combine them to show how you can build architectures that support constant change.

You'll explore:

- **Fitness functions:** objectives you want your architecture to exhibit or move towards
- **Incremental change:** making gradual changes through development and operations
- **Architectural coupling:** finding the correct level of architectural coupling to support change without creating brittleness
- **Evolutionary data:** evolving database structures as requirements and architecture shift over time
- **Building evolvable architectures:** how to combine all of these aspects to create evolutionary architectures
- **Putting evolutionary architecture into practice:** practical guidelines to get you started

"This book is packed with nomenclatures and deliberate practices that will significantly benefit anyone in the role of an architect. Wish I had this in my hands decades ago; glad its here now."

—**Dr. Venkat Subramaniam**
award-winning author and founder of Agile Developer, Inc.

**Neal Ford** is Software Architect and Meme Wrangler at ThoughtWorks, a global IT consultancy with an exclusive focus on end-to- end software development and delivery.

**Dr. Rebecca Parsons**, ThoughtWorks' Chief Technology Officer, has extensive experience creating large-scale distributed object applications and the integration of disparate systems.

**Patrick Kua**, a Tech Principal and Generalizing Specialist at ThoughtWorks, has over a decade of experience in agile and lean development processes.

US $59.99 CAN $79.99
ISBN: 978-1-491-98636-3


Twitter: @oreillymedia
facebook.com/oreilly

# Building Evolutionary Architectures

## *Support Constant Change*

*Neal Ford, Rebecca Parsons, and Patrick Kua*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

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



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/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Editor:** Rachel Roumeliotis
**Production Editor:** Justin Billing
**Copyeditor:** Christina Edwards
**Proofreader:** Matthew Burgoyne

**Indexer:** WordCo, Inc.
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

October 2017: First Edition

**Revision History for the First Edition**
2017-09-15: First Release

See *http://www.oreilly.com/catalog/errata.csp?isbn=9781491986363* for release details.



978-1-491-98636-3

[LSI]

# Table of Contents

**Foreword** ..... vii

**Preface** ..... ix

**1. Software Architecture** ..... 1
Evolutionary Architecture 3
How Is Long-term Planning Possible When Everything Changes All the Time? 3
Once I've Built an Architecture, How Can I Prevent It from Gradually Degrading Over Time? 6
Incremental Change 6
Guided Change 7
Multiple Architectural Dimensions 8
Conway's Law 11
Why Evolutionary? 13
Summary 14

**2. Fitness Functions** ..... 15
What is a Fitness Function? 17
Categories 18
Atomic Versus Holistic 19
Triggered Versus Continual 19
Static Versus Dynamic 20
Automated Versus Manual 20
Temporal 21
Intentional Over Emergent 21
Domain-specific 21
Identify Fitness Functions Early 22

Review Fitness Functions 23

**3. Engineering Incremental Change. . . . . 25**
Building Blocks 28
Testable 29
Deployment Pipelines 31
Combining Fitness Function Categories 35
Case Study: Architectural Restructuring while Deploying 60 Times/Day 37
Conflicting Goals 39
Case Study: Adding Fitness Functions to PenultimateWidgets' Invoicing Service 40
Hypothesis- and Data-Driven Development 42
Case Study: What to Port? 44

**4. Architectural Coupling. . . . . 47**
Modularity 47
Architectural Quanta and Granularity 48
Evolvability of Architectural Styles 51
Big Ball of Mud 52
Monoliths 53
Event-Driven Architectures 60
Service-Oriented Architectures 65
"Serverless" Architectures 76
Controlling Quantum Size 78
Case Study: Guarding Against Component Cycles 79

**5. Evolutionary Data. . . . . 83**
Evolutionary Database Design 83
Evolving Schemas 83
Shared Database Integration 85
Inappropriate Data Coupling 89
Two-Phase Commit Transactions 90
Age and Quality of Data 92
Case Study: Evolving PenultimateWidgets' Routing 93

**6. Building Evolvable Architectures. . . . . 95**
Mechanics 95
1. Identify Dimensions Affected by Evolution 95
2. Define Fitness Function(s) for Each Dimension 96
3. Use Deployment Pipelines to Automate Fitness Functions 96
Greenfield Projects 97
Retrofitting Existing Architectures 97

Appropriate Coupling and Cohesion 97
Engineering Practices 98
Fitness Functions 98
COTS Implications 99
Migrating Architectures 100
Migration Steps 101
Evolving Module Interactions 104
Guidelines for Building Evolutionary Architectures 107
Remove Needless Variability 107
Make Decisions Reversible 109
Prefer Evolvable over Predictable 110
Build Anticorruption Layers 111
Case Study: Service Templates 113
Build Sacrificial Architectures 114
Mitigate External Change 115
Updating Libraries Versus Frameworks 117
Prefer Continuous Delivery to Snapshots 118
Version Services Internally 119
Case Study: Evolving PenultimateWidgets' Ratings 119

**7. Evolutionary Architecture Pitfalls and Antipatterns. . . . . 123**
Technical Architecture 123
Antipattern: Vendor King 123
Pitfall: Leaky Abstractions 125
Antipattern: Last 10% Trap 127
Antipattern: Code Reuse Abuse 128
Case Study: Reuse at PenultimateWidgets 130
Pitfall: Resume-Driven Development 131
Incremental Change 131
Antipattern: Inappropriate Governance 132
Case Study: Goldilocks Governance at PenultimateWidgets 134
Pitfall: Lack of Speed to Release 134
Business Concerns 136
Pitfall: Product Customization 136
Antipattern: Reporting 137
Pitfall: Planning Horizons 138

**8. Putting Evolutionary Architecture into Practice. . . . . 141**
Organizational Factors 141
Cross-Functional Teams 141
Organized Around Business Capabilities 143
Product over Project 144

Dealing with External Change 145
Connections Between Team Members 146
Team Coupling Characteristics 147
Culture 148
Culture of Experimentation 149
CFO and Budgeting 151
Building Enterprise Fitness Functions 152
Case Study: PenultimateWidgets as a Platform 153
Where Do You Start? 153
Low-Hanging Fruit 153
Highest-Value 154
Testing 154
Infrastructure 155
Case Study: Enterprise Architecture at PenultimateWidgets 156
Future State? 157
Fitness Functions Using AI 157
Generative Testing 157
Why (or Why Not)? 157
Why Should a Company Decide to Build an Evolutionary Architecture? 158
Case Study: Selective Scale at PenultimateWidgets 160
Why Would a Company Choose Not to Build an Evolutionary Architecture? 161
Convincing Others 162
Case Study: Consulting Judo 163
The Business Case 163
"The Future Is Already Here…" 163
Moving Fast Without Breaking Things 164
Less Risk 164
New Capabilities 164
Building Evolutionary Architectures 164

**Index** ..... **167**

# Foreword

For a long time, the software industry followed the notion that architecture was something that ought to be developed and completed before writing the first line of code. Inspired by the construction industry, it was felt that the sign of a successful software architecture was something that didn't need to change during development, often a reaction to the high costs of scrap and rework that would occur due to a re-architecture event.

This vision of architecture was rudely challenged by the rise of agile software methods. The pre-planned architecture approach was founded on the notion that requirements should also be fixed before coding began, leading to a phased (or waterfall) approach where requirements was followed by architecture which itself was followed by construction (programming). The agile world, however, challenged the very notion of fixed requirements, observing that regular changes in requirements were a business necessity in the modern world, and providing project planning techniques to embrace controlled change.

In this new agile world, many people questioned the role of architecture. And certainly the pre-planned architecture vision couldn't fit in with modern dynamism. But there is another approach to architecture, one that embraces change in the agile manner. In this view architecture is an constant effort, one that works closely with programming so that architecture can react both to changing requirements but also to feedback from programming. We've come to call this evolutionary architecture, to highlight that while the changes are unpredictable, the architecture can still move in a good direction.

At ThoughtWorks, we've been immersed in this architectural world-view. Rebecca led many of our most important projects in the early years of this millenium, and developed our technical leadership as our CTO. Neal has been a careful observer of our work, synthesizing and conveying the lessons we've learned. Pat has combined his project work with developing our technical leads. We've always felt that architecture is vitally important, and can't be left to idle chance. We've made mistakes, but learned

from them, growing a better understanding of how to build a code base that can respond gracefully to the many changes in its purpose.

The heart of doing evolutionary architecture is to make small changes, and put in feedback loops that allow everyone to learn from how the system is developing. The rise of Continuous Delivery has been a crucial enabling factor in making evolutionary architecture practical. The authorial trio use the notion of fitness functions to monitor the state of the architecture. They explore different styles of evolvability for architecture, and put emphasis on the issues around long-lived data—often a topic that gets neglected. Conway's Law towers over much of the discussion, as it should.

While I'm sure we have much to learn about doing software architecture in an evolutionary style, this book marks an essential road map on the current state of understanding. As more people are realizing the central role of software systems in our twenty-first century human world, knowing how best to respond to change while keeping on your feet will be an essential skill for any software leader.

*— Martin Fowler*
*martinfowler.com*
*September 2017*

# Preface

## Conventions Used in This Book

The following typographical conventions are used in this book:

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

`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.

## O'Reilly Safari


*Safari* (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

## 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 *http://oreil.ly/2eY9gT6*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

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

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

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

## Additional Information

The authors maintain a companion website for this book at *http://evolutionaryarchitecture.com*

## Acknowledgments

Neal would like to thank all the attendees of the various conferences at which he has spoken over the last few years to help hone and revise this material live. He would also like to thank the technical reviewers who went above and beyond to provide

excellent feedback and advice, especially Venkat Subramanium, Eoin Woods, Simon Brown, and Martin Fowler. Neal would also like to thank his cats Winston, Parker, and Isabella for providing useful distractions that always lead to insights. He thanks his friend John Drescher, all his ThoughtWorks colleagues, Norman Zapien for his crafty ear, his yearly Pasty Geeks vacation group and neighborhood Cocktail Club for support and friendship. And finally, he'd like to thank his long-suffering wife, who endures his travel and other professional indignities with a smile.

Rebecca would like to thank all of the colleagues, conference attendees and speakers, and authors who have, over the years, contributed ideas, tools, and methods and asked clarifying questions about the field of evolutionary architecture. She would echo Neal's thanks to the technical reviewers for their careful reading and commentary. Further, Rebecca would like to thank her co-authors for all the enlightening conversations and discussions while we worked together on this book. In particular, she thanks Neal for the great discussion, or perhaps debate, they had several years ago regarding the distinction between emergent and evolutionary architecture. These ideas have come a long way since that first conversation.

Patrick would like to thank all of his colleagues and customers at ThoughtWorks, who have driven the need and provided the testbed to articulate the ideas in building evolutionary architecture. He would also like to echo Neal and Rebecca's thanks to the technical reviewers, whose feedback helped to improve the book immensely. Finally, he would like to thank his co-authors for the past several years and for the opportunity to work closely together on this topic, despite the numerous time zones and flights that made meeting in person the rare occasion.

# CHAPTER 1
# Software Architecture

Developers have long struggled to coin a succinct, concise definition of software architecture because the scope is large and ever-changing. Ralph Johnson famously defined software architecture as "the important stuff (whatever that is)." The architect's job is to understand and balance all of those important things (whatever they are).

An initial part of an architect's job is to understand the business or domain requirements for a proposed solution. Though these requirements operate as the motivation for utilizing software to solve a problem, they are ultimately only one factor that architects should contemplate when crafting an architecture. Architects must also consider numerous other factors, some explicit (e.g., performance service-level agreements) and others implicit to the nature of the business (e.g., the company is embarking on a mergers and acquisition spree). Therefore, the craft of software architecture manifests in the ability of architects to analyze business and domain requirements along with other important factors to find a solution that balances all concerns optimally. The scope of software architecture is derived from the combination of all these architectural factors, as shown in Figure 1-1.

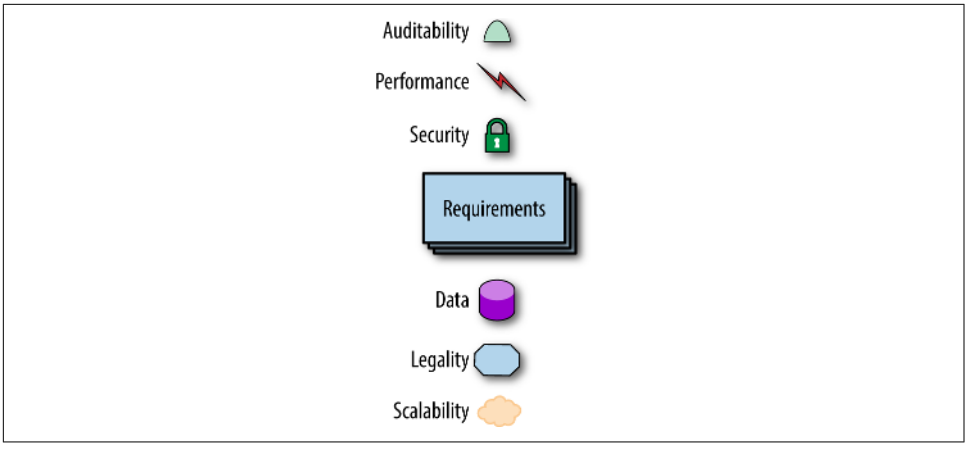



*Figure 1-1. The entire scope of architecture encompasses requirements plus "-ilities"*

As seen in Figure 1-1, business and domain requirements exist alongside other architecture concerns (defined by architects). This includes a wide range of external factors that can alter the decision process on what and how to build a software system. For a sampling, check out the list in Table 1-1:

*Table 1-1. Partial list of "-ilities"*

| | | | | |
|---|---|---|---|---|
| accessibility | accountability | accuracy | adaptability | administrability |
| affordability | agility | auditability | autonomy | availability |
| compatibility | composability | configurability | correctness | credibility |
| customizability | debugability | degradability | determinability | demonstrability |
| dependability | deployability | discoverability | distributability | durability |
| effectiveness | efficiency | usability | extensibility | failure transparency |
| fault tolerance | fidelity | flexibility | inspectability | installability |
| integrity | interoperability | learnability | maintainability | manageability |
| mobility | modifiability | modularity | operability | orthogonality |
| portability | precision | predictability | process capabilities | producibility |
| provability | recoverability | relevance | reliability | repeatability |
| reproducibility | resilience | responsiveness | reusability | robustness |
| safety | scalability | seamlessness | self-sustainability | serviceability |
| securability | simplicity | stability | standards compliance | survivability |
| sustainability | tailorability | testability | timeliness | traceability |

When building software, architects must determine the most important of these "-ilities." However, many of these factors oppose one another. For example, achieving both high performance and extreme scalability can be difficult because achieving

both requires a careful balance of architecture, operations, and many other factors. As a result, the necessary analysis in architecture design and the inevitable clash of competing factors requires balance, but balancing the pros and cons of each architectural decision leads to the *tradeoffs* so commonly lamented by architects. In the last few years, incremental developments in core engineering practices for software development have laid the foundation for rethinking how architecture changes over time and on ways to protect important architectural characteristics as this evolution occurs. This book ties those parts together with a new way to think about *architecture* and *time*.

We want to add a new standard "-ility" to software architecture—*evolvability*.

# Evolutionary Architecture

Despite our best efforts, software becomes harder to change over time. For a variety of reasons, the parts that comprise software systems defy easy modification, becoming more brittle and intractable over time. Changes in software projects are usually driven by a reevaluation of functionality and/or scope. But another type of change occurs outside the control of architects and long-term planners. Though architects like to be able to strategically plan for the future, the constantly changing software development ecosystem makes that difficult. Since we can't avoid change, we need to exploit it.

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

In the biological world, the environment changes constantly from both natural and man-made causes. For example, in the early 1930s, Australia had problems with cane beetles, which rendered the production and harvesting sugar cane crops less profitable. In response, in June 1935, the then Bureau of Sugar Experiment Stations introduced a predator, the cane toad, previously only native to south and middle America.[1] After being bred in captivity a number of young toads were released in North Queensland in July and August 1935. With poisonous skin and no native predators, the cane toads spread widely; there are an estimated 200 million in existence today. The moral: introducing changes to a highly dynamic (eco)system can yield unpredictable results.

The software development ecosystem consists of all the tools, frameworks, libraries, and best practices—the accumulated state of the art in software development. This

1 Clarke, G. M., Gross, S., Matthews, M., Catling, P. C., Baker, B., Hewitt, C. L., Crowther, D., & Saddler, S. R. 2000, Environmental Pest Species in Australia, Australia: State of the Environment, Second Technical Paper Series (Biodiversity), Department of the Environment and Heritage, Canberra.

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 she continues to adjust to stay upright and *equilibrium* because she continues 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 her 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 5-year plans because the entire software development universe will evolve in that timeframe, 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 or 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 programming platforms we use exemplify constant evolution. Newer versions of a programming language offer better application programming interfaces (APIs) to improve the flexibility or applicability toward 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 1-2.

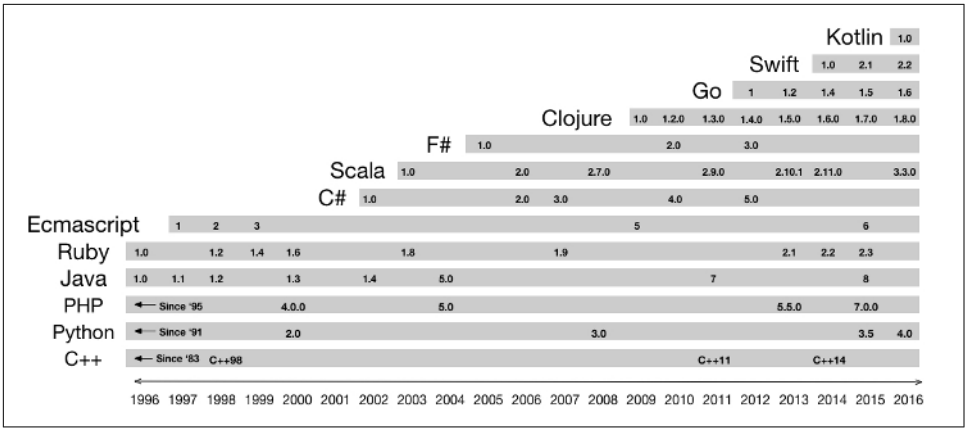



*Figure 1-2. The evolution of popular programming languages*

Regardless of which particular aspect of software development—the programming platform, languages, the operating environment, persistence technologies, 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 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 by automating 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 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 in a new light: 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 Gradually 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 characteristics implies protecting the other characteristics as the system evolves. For example, if an architect has designed an architecture for scalability, she doesn'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.

In this book, we illustrate that a side effect of an evolutionary architecture is mechanisms to protect the important architecture characteristics. We explore the ideas behind *continual architecture*: building architectures that have no end state and are designed to evolve with the ever-changing software development ecosystem, and including built-in protections around important architectural characteristics. We don't attempt to define software architecture in totality; many other definitions exist. We focus instead on extending current definitions to adding *time* and *change* as first-class architectural elements.

Here is our definition of evolutionary architecture:

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

## 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, incre-

mental 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 microservice architecture and modern engineering practices. One of the page's features is the ability of users to rate different widgets with star ratings. Other services within PenultimateWidgets' business also need ratings (customer service representatives, shipping provider evaluation, and so on), 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 rather 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 these practices are required in all cases but rather commonly occur together in the wild. We discuss how to achieve incremental change in Chapter 3.

## Guided Change

Once architects 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.

## 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 physics 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 phenomenon 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, but that is only one dimension of a software project. If architects want to create an architecture that can evolve, they must consider all 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" above), but *dimensions* are actually broader, encapsulating things traditionally outside the purview of technical

architecture. Each project has dimensions the architect 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 the implementation language(s).

*Data*
: Database schemas, table layouts, optimization planning, etc. The database administrator generally handles this type of architecture.

*Security*
: Defines security policies, 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.

A variety of partitioning techniques exist for conceptually carving up architectures. For example, the 4 + 1 architecture View Model, which focuses on different perspectives from different roles and was incorporated into the IEEE definition of software architecture, splits the ecosystem into *logical*, *development*, *process*, and *physical* views. In the well-known book Software Systems Architecture, the authors posit a catalog of viewpoints on software architecture, spanning a larger set of roles. Similarly, Simon Brown's C4 notation partitions concerns for aid in conceptual organization. In this text, in contrast, we don't attempt to create a taxonomy of dimensions but rather recognize the ones extant in existing projects. Pragmatically, regardless of which category a particular important concern falls into, the architect must still protect that dimension. Different projects have differing concerns, leading to unique sets of dimensions for a given project. Any of these techniques provide useful insight, particularly in new projects, but existing projects must deal with the realities of what exists.

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, and so on), 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 1-3.

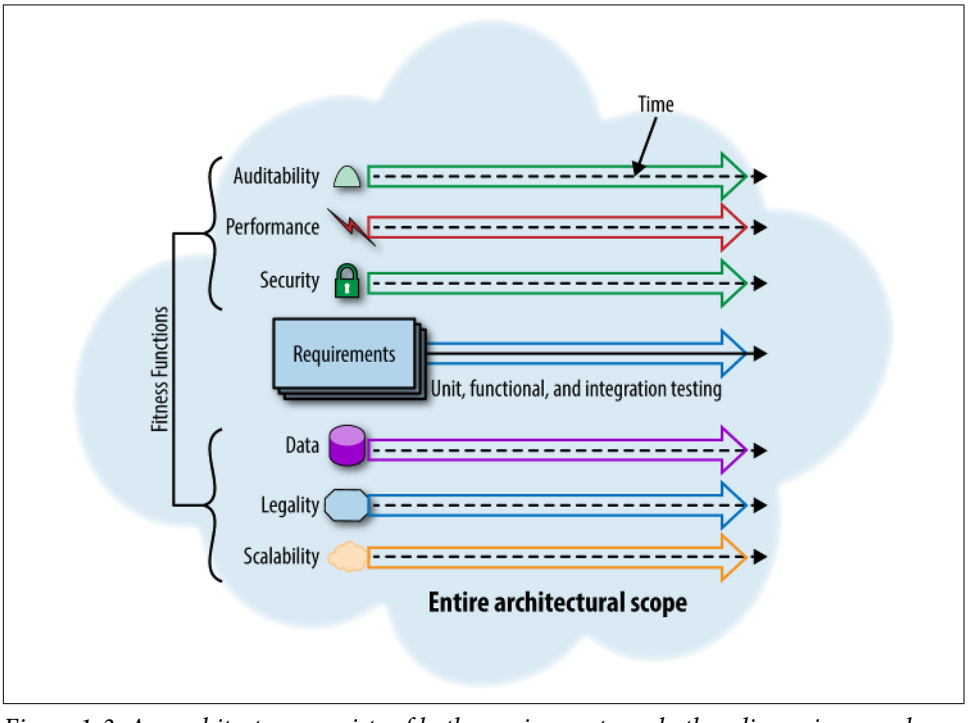



*Figure 1-3. An architecture consists of both requirements and other dimensions, each protected by fitness functions*

In Figure 1-3, 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 utilize fitness functions to protect their 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 4 and the impacts of data coupling in Chapter 5.

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.

## Conway's Law

In April 1968, Melvin Conway submitted a paper to Harvard Business Review called, "How Do Committees Invent?". In this paper, Conway introduced the notion that the social structures, particularly the communication paths between people, inevitably influence final product design.

As Conway describes, in the very early stage of the design, a high-level understanding of the system is made to understand how to break down areas of responsibility into different patterns. The way that a group breaks down a problem affects choices that they can make later. He codified what has become known as *Conway's Law*:

> Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.
>
> —Melvin Conway

As Conway notes, when technologists break down problems into smaller chunks to delegate, we introduce coordination problems. In many organizations, formal communication structures or rigid hierarchy appear to solve this coordination problem but often lead to inflexible solutions. For example, in a layered architecture where the team is separated by technical function (user interface, business logic, and so on), solving common problems that cut vertically across layers increases coordination overhead. People who have worked in startups and then have joined joined large multinational corporations have likely experienced the contrast between the nimble, adaptable culture of the former and the inflexible communication structures of the latter. A good example of Conway's Law in action might be trying to change the contract between two services, which could be difficult if the successful change of a service owned by one team requires athecoordinated and agreed-upon effort of another.

In his paper, Conway was effectively warning software architects to pay attention not only to the architecture and design of the software, but also the delegation, assignment, and coordination of the work between teams.

In many organizations teams are divided according to their functional skills. Some common examples include:

*Front-end developers*
: A team with specialized skills in a particular user interface (UI) technology (e.g., HTML, mobile, desktop).

*Back-end developers*
: A team with unique skills in building back-end services, sometimes API tiers.

*Database developers*
: A team with unique skills in building storage and logic services.

In organizations with functional silos, management divides teams to make their Human Resources department happy without much regard to engineering efficiency. Although each team may be good at their part of the design (e.g., building a screen, adding a back-end API or service, or developing a new storage mechanism), to release a new business capability or feature, all three teams must be involved in building the feature. Teams typically optimize for efficiency for their immediate tasks rather than the more abstract, strategic goals of the business, particularly when under schedule pressure. Instead of delivering an end-to-end feature value, teams often focus on delivering components that may or may not work well with each other.

In this organizational split, a feature dependent on all three teams takes longer as each team works on their component at different times. For example, consider the common business change of updating the Catalog page. That change entails the UI, business rules, and database schema changes. If each team works in their own silo, they must coordinate schedules, extending the time required to implement the feature. This is a great example of how team structure can impact architecture and the ability to evolve.

As Conway noted in his paper by noting *every time a delegation is made and somebody's scope of inquiry is narrowed, the class of design alternatives which can be effectively pursued is also narrowed*. Stated another way, it's hard for someone to change something if the thing she wants to change is owned by someone else. Software architects should pay attention to how work is divided and delegated to align architectural goals with team structure.

Many companies who build architectures such as microservices structure their teams around service boundaries rather than siloed technical architecture partitions. In the ThoughtWorks Technology Radar, we call this the Inverse Conway Maneuver. Organization of teams in such a manner is ideal because team structure will impact myriad dimensions of software development and should reflect the problem size and scope. For example, when building a microservices architecture, companies typically structure teams that that resemble the architecture by cutting across functional silos and including team members who cover every angle of the business and technical aspects of the architecture.

Structure teams to look like your target architecture, and it will be easier to achieve it.

### Introducing PenultimateWidgets and Their Inverse Conway Moment

Throughout this book, we use the example company, PenultimateWidgets, the Second to Last Widget Dealer, a large online seller of widgets (a variety of little unspecified things). The company is gradually updating much of their IT infrastructure. They have some legacy systems they want to keep around for a while, and new strategic systems that require more iterative approaches. Throughout the chapters, we'll highlight many of the problems and solutions PenultimateWidgets develops to address those needs.

The first observation their architects made concerned the software development teams. The old monolithic applications utilized a layered architecture, separating presentation, business logic, persistence, and operations. Their team mirrors these functions: All the UI developers sit together, developers and database administrators have their own silo, and operations is outsourced to a third party.

When the developers started working on the new architectural elements, a microservices architecture with fine-grained services, the coordination costs skyrocketed. Because the services were built around domains (such as *CustomerCheckout*) rather than technical architecture, making a change to a single domain required a crippling amount of coordination across their silos.

Instead, PenultimateWidgets applied the Inverse Conway Maneuver and built cross-functional teams that matched the purview of the service: each service team consists of service owner, a few developers, a business analyst, a database administrator (DBA), a quality assurance (QA) person, and an operations person.

Team impact shows up in many places throughout the book, with examples of how many consequences it has.

## Why Evolutionary?

A common question about evolutionary architecture concerns the name itself: why call it *evolutionary* architecture and not something else? Other possible names include *incremental*, *continual*, *agile*, *reactive*, and *emergent*, to name just a few. But each of these terms misses the mark here. 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 capture 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 cover 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* is about 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.

## Summary

An evolutionary architecture consists of three primary aspects: incremental change, fitness functions, and appropriate coupling. In the remainder of the book, we discuss each of these factors separately, then combine them to address what it takes to build and maintain architectures that support constant change.

# CHAPTER 2
# Fitness Functions

> An evolutionary architecture supports *guided*, incremental change across multiple dimensions.
>
> —our definition

As noted, the word *guided* indicates that some objective exists that architecture should move toward or exhibit. The authors borrow a concept from evolutionary computing called "fitness functions," used in genetic algorithm design to define success. Evolutionary computing includes a number of mechanisms that allow a solution to gradually emerge via small changes in each generation of the software. At each generation of the solution, the engineer assesses the current state: Is it closer to or further away from the ultimate goal? For example, when using a genetic algorithm to optimize wing design, the fitness function assess wind resistance, weight, air flow, and other characteristics desirable to good wing design. Architects define a fitness function to explain what better is and to help measure when the goal is met. In software, fitness functions check that developers preserve important architectural characteristics.

We use this concept to define architectural fitness functions:

> An architectural fitness function provides an objective integrity assessment of some architectural characteristic(s).
>
> —our definition

The fitness function protects the various architectural characteristics required for the system. The specific architectural requirements differ greatly across systems and organizations, based on business drivers, technical capabilities, and a host of other factors. Some systems require intense security; others require significant throughput or low latency. Whereas some might need to be more resilient to failure. These considerations form the "-ilities" that architects care about. Conceptually, an architec-

tural fitness function embodies a protection mechanism for the "-ilities" of a given system.

We can also think about the *systemwide fitness function* as a collection of fitness functions with each function corresponding to one or more dimensions of the architecture. Using a systemwide fitness function aids our understanding of necessary tradeoffs when individual elements of the fitness function conflict with each other. As is common with multifunction optimization problems, we might find it impossible to optimize all values simultaneously, forcing us to make choices. For example, in the case of architectural fitness functions, issues like performance might conflict with security due to the cost of encryption. This is a classic example of the bane of architects everywhere—the *tradeoff*. Tradeoffs dominate much of an architect's headaches during the struggle to reconcile opposing forces, such as scalability and performance. However, architects have a perpetual problem of comparing these different characteristics because they fundamentally differ (an apples to oranges comparison) and all stakeholders believe their concern is paramount. Systemwide fitness functions allow architects to think about divergent concerns using the same unifying mechanism of fitness functions, capturing and preserving the important architectural characteristics. The relationship between the systemwide fitness function and its constituent smaller fitness functions is illustrated in Figure 2-1.

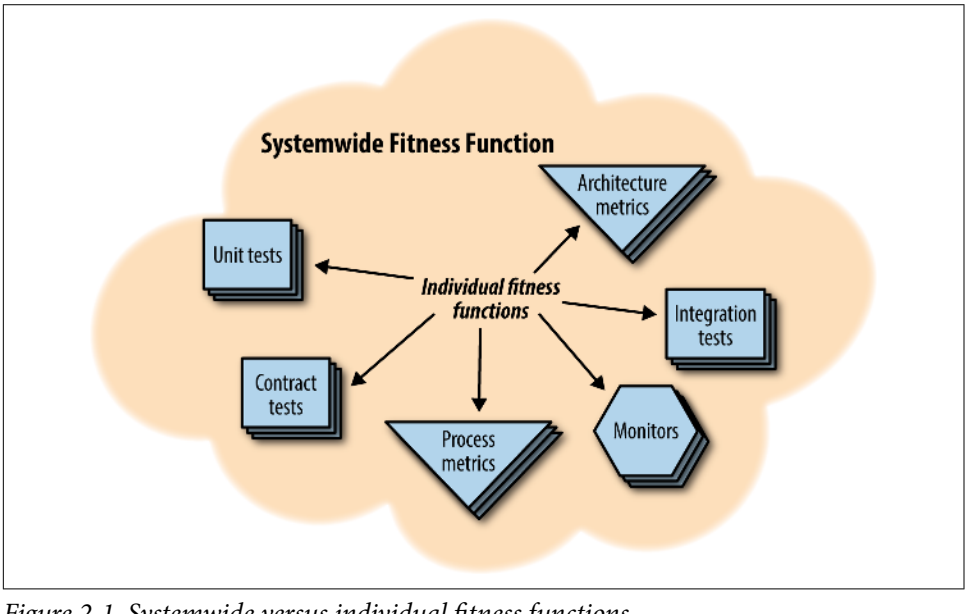



*Figure 2-1. Systemwide versus individual fitness functions*

The systemwide fitness function is crucial for an architecture to be evolutionary, as we need some basis to allow architects to compare and evaluate architectural characteristics against one another. Unlike the more directed fitness functions, architects

will likely never try to "evaluate" this systemwide fitness function. Rather, it provides guidelines for prioritizing decisions about the architecture in the future.

> A system is never the sum of its parts. It is the product of the interactions of its parts.
>
> —Dr. Russel Ackoff

Without guidance, evolutionary architecture becomes simply a reactionary architecture. Thus, a crucial early architectural decision for any system is to define important dimensions such as scalability, performance, security, data schemas, and so on. Conceptually, this allows architects to weigh the importance of a fitness function based on its importance to the system's overall behavior.

We first define fitness functions more rigorously, and then examine conceptually how they guide the evolution of the architecture.

## What is a Fitness Function?

Mathematically speaking, a function takes input from some allowed set of input values and produces an output in some allowed set of output values. In software, we also generally use the term function to refer to something that is actually implementable. However, as with acceptance criteria in agile software development, the fitness functions for evolutionary architecture may not be implementable in software (e.g., a required manual process for regulatory reasons), but architects must still define manual fitness functions to help guide the evolution of the system. While automated checks are preferable, some projects cannot automate all fitness functions. Thus, it is still useful for architects to elucidate architectural verifications explicitly as fitness functions for many reasons that will become evident.

As discussed in Chapter 1, real-world architecture consists of many different dimensions, including requirements around performance, reliability, security, operability, coding standards, and integration, to name a few. We want a fitness function to represent each requirement for the architecture. Developers commonly express fitness functions using different kinds of mechanisms, such as tests or metrics. We'll look at a few examples and then consider the different kinds of functions more broadly.

Performance requirements make good use of fitness functions. Consider a requirement that all service calls must respond within 100ms. We can implement a test (i.e., fitness function) that measures the response to a service request and fails if the result is greater than 100ms. To this end, every new service should have a corresponding performance test added to the suite. Developers writing the tests must decide what level of comprehensiveness of the range and types of inputs establish confidence in the passing test. They must also decide when to run these tests and how to handle test failures. Performance testing should be conducted early and frequently, in particular to pick up inflection points when performance changes radically (usually in the wrong direction) because of an update to code.

Fitness functions can also be used to maintain coding standards. A common code metric is cyclomatic complexity, a measure of function or method complexity. An architect may set a threshold for an upper value, guarded by a unit test running in continuous integration, using one of the many tools to evaluate that metric. In the previous example, architects decide when to run the fitness functions to assess performance. For coding standards, developers want violations to fail the build immediately and to address the problem aggressively.

Despite need, developers cannot always implement some fitness functions completely because of complexity or other constraints. Consider something like a failover for a database from a hard failure. While the recovery itself might be fully automated (and should be), triggering the test itself is likely best done manually. Additionally, it might be far more efficient to determine the success of the test manually, although scripts and automation are still encouraged.

These examples highlight the different myriad forms fitness functions can take, the immediate response to failure of a fitness function, and even when and how developers might run them. While we can't necessarily run a single script and say "our architecture currently has a composite fitness score of 42," we can have precise and unambiguous conversations about the state of the architecture relative to the systemwide fitness function. We can also entertain discussions about the changes that might incur on the architecture's fitness.

Finally, when we say an evolutionary architecture is guided by the fitness function, we mean we evaluate individual architectural choices against the individual and the systemwide fitness function to determine the impact of the change. The fitness functions collectively denote what matters to us in our architecture, allowing us to make the kinds of trade-off decisions that are both crucial and vexing during the development of software systems.

Fitness functions unify many existing concepts into a single mechanism, allowing architects to think in a uniform way about many existing (often ad hoc) "nonfunctional requirements" tests. Collecting important architecture thresholds and requirements as fitness functions allows for a more concrete representation for previously fuzzy, subjective evaluation criteria. We leverage a large number of existing mechanisms to build fitness functions, including traditional testing, monitoring, and other tools. Not all tests are fitness functions, but some tests are—if the test helps verify the integrity of architectural concerns, we consider it a fitness function.

## Categories

Fitness functions exist across a variety of categories related to their scope, frequency, dynamics, and other factors, including combinations of categories where useful.

## Atomic Versus Holistic

*Atomic* fitness functions run against a singular context and exercise one particular aspect of the architecture. An excellent example of an atomic fitness function is a unit test that verifies some architectural characteristic, such as modular coupling (we show an example of this type of fitness function in Chapter 4). Thus, some application-level testing falls under the heading of fitness functions, but not all unit tests serve as fitness functions—only the ones that verify architecture characteristic(s).

For some architectural characteristics, developers must test more than each architectural dimension in isolation. *Holistic* fitness functions run against a shared context and exercise a combination of architectural aspects such as security and scalability. Developers design holistic fitness functions to ensure that combined features that work atomically don't break in real-world combinations. For example, imagine an architecture has fitness functions around both security and scalability. One of the key items the security fitness function checks is staleness of data, and a key item for the scalability tests is number of concurrent users within a certainly latency range. To achieve scalability, developers implement caching, which allows the atomic scalability fitness function to pass. When caching isn't turned on, the security fitness function passes. However, when run holistically, enabling caching makes data too stale to pass the security fitness function, and the holistic test fails.

We obviously cannot test every possible combination of architecture elements, so architects use holistic fitness functions selectively to test important interactions. This selectivity and prioritization also allows architects and developers to assess the difficultly implementing a particular testing scenario (via fitness functions), thus allowing an assessment of how valuable that characteristic is. Frequently, the interactions between architectural concerns determines the quality of the architecture, which holistic fitness functions address.

## Triggered Versus Continual

Execution cadence is another distinguishing factor between fitness functions. *Triggered* fitness functions run based on a particular event, such as a developer executing a unit test, a deployment pipeline running unit tests, or a QA person performing exploratory testing. This encompasses traditional testing such as unit, functional, behavior-driven development (BDD), and other tests developers.

*Continual* tests don't run on a schedule, but instead execute constant verification of architectural aspect(s) such as transaction speed. For example, consider a microservices architecture where the architects want to build a fitness function around transaction time—how long does it take for a transaction to complete on average? Building any kind of triggered test provides sparse information about real-world behavior. Thus, instead of using a triggered test, developers build a fitness function that simu-

lates a transaction in production while all the other real transactions run. This allows developers to verify behavior and gather real data about the system "in the wild."

Monitoring-driven development (MDD) is another testing technique gaining popularity. Rather than relying solely on tests to verify system results, MDD uses monitors in production to assess both technical and business health. These continual fitness functions are more dynamic than standard triggered tests.

## Static Versus Dynamic

*Static* fitness functions have a fixed result, such as the binary *pass/fail* of a unit test. This type encompasses any fitness function that has a predefined desirable value: binary, a number range, set inclusion, and so on. Metrics are often used for fitness functions. For example, an architect may define acceptable ranges for average cyclomatic complexity of methods in the code base, graded upon checkin using a metrics tool wired into the deployment pipeline.

*Dynamic* fitness functions rely on a shifting definition based on extra context. Some values may be contingent on circumstances, and most architects will accept lower performance metrics when operating at high scale. For example, a company might build a sliding value for performance based on scalability—more scale means slower performance is permitted, but only within a range.

## Automated Versus Manual

Clearly, architects like automated things—part of incremental change includes automation, which we delve into deeply in Chapter 3. Thus, it's not surprising that developers will execute most fitness functions within an automated context: continuous integration, deployment pipelines, and so on. Indeed, developers and DevOps have performed a tremendous amount of work under the auspices of Continuous Delivery to automate many parts of the software development ecosystem previous thought impossible. This beneficial trend should continue.

However, as much as we'd like to automate every single aspect of software development, some aspects of software resist automation. Sometimes, a critical dimension within a system, such as legal requirements, defies automation. For example, developers building applications in some problem domains must have manual certification for changes for legal reasons, which cannot be automated away. Similarly, a project may have aspirations to become more evolutionary evolutionary but not yet have appropriate engineering practices in place. For example, perhaps most QA is still manual on a particular project and must remain so for the near future. In both these cases (and others), we need *manual* fitness functions that are verified by a person-based process.

Clearly, the path to better efficiency eliminates as many manual steps as possible, but many projects still require necessary manual procedures. We still define fitness functions for those characteristics and verify them using manuals stages in deployment pipelines (covered in more detail in Chapter 3).

## Temporal

While most fitness functions trigger on change, architects may want to build a time component into assessing fitness. For example, if a project uses an encryption library, the architect may want to create a temporal fitness function as a reminder to check to see if important updates have been performed. Another common use of this type of fitness function is a *break upon upgrade* test. In platforms like Ruby on Rails, some developers can't wait for the tantalizing new features coming in the next release, so they add a feature to the current version via a *back port*, a custom implementation of a future feature. Problems arise when the project finally upgrades to the new version because the back port is often incompatible with the "real" version. Developers use *break upon upgrade* tests to wrap back ported features to force re-evaluation when the upgrade occurs.

## Intentional Over Emergent

While architects will define most fitness functions at project inception as they elucidate the characteristics of the architecture, some fitness functions will emerge during development of the system. Architects never know all important parts of the architecture at the beginning (the classic *unknown unknowns* problem we address in Chapter 6) and thus must identify fitness functions as the system evolves.

## Domain-specific

Some architectures have specific concerns, such as special security or regulatory requirements. For example, a company that handles international fund transfers might design a specific, continuous, holistic fitness function that stress tests security, modeled after the way that the Simian Army (covered in Chapter 3) stresses infrastructure. Many problem domains contain drivers that lead architects toward one or more set of important characteristics. Architects and developers should capture those drivers as fitness functions to ensure that those important characteristics don't degrade over time.

We show examples of combining these dimensions when it comes time to evaluate fitness functions in Chapter 3.

## Identify Fitness Functions Early

Teams should identify fitness functions as part of their initial understanding of the overall architecture concerns that their design must support. They should also identify their *system* fitness function early to help determine the sort of change that they want to support. Discussions comparing the value and difficulty of implementing different architecture characteristics (along with their fitness functions) help prioritize riskier work earlier to understand how to design for change.

Teams that do not identify their fitness functions face the following risks:

- Making the wrong design choices that ultimately lead to building software that fails in its environment
- Making design choices that cost time and/or money but are unnecessary
- Not being able to evolve the system easily in the future when the environment changes

For each software system, teams should focus on identifying and prioritizing the most important fitness functions as early as possible. Early identification of fitness functions help architects plan for breaking a large system into smaller systems, each dealing with a smaller set of fitness functions.

For example, some companies deal with security-sensitive data such as a credit card or payment details. Depending on the industry and/or job, storing these sorts of information imply stronger regulatory requirements, which may shift because of changes in legislation or standards that impact regulations or because of expansion into new states, territories, or countries with different legislative requirements.

If architects determine that security and payment play a significant role in the system-wide fitness function, it may lead the team to design an architecture that keeps these concerns together. Without identifying fitness functions this early, a team may end up with these responsibilities scattered throughout the entire codebase, requiring a broader impact analysis to understand change and driving up the overall cost of modification.

Fitness functions can be classified into three simple categories:

*Key*
: These dimensions are critical in making technology or design choices. More effort should be invested to explore design choices that make change around these elements significantly easier. For example, for a banking application, performance and resiliency are key dimensions.

*Relevant*
: These dimensions need to be considered at a feature level, but are unlikely to guide architecture choices. For example, code metrics around the quality of code base are important but not key.

*Not Relevant*
: Design and technology choices are not impacted by these types of dimensions. For example, process metrics such as cycle time (the amount of time to move from design to implementation, may be important in some ways but is irrelevant to architecture. As a result, fitness functions for it are not necessary.

Keep knowledge of key and relevant fitness functions alive by posting the results of executing fitness functions somewhere visible or in a shared space so that developers remember to consider them in day-to-day coding.

Classifying fitness functions into categories helps prioritize design decisions. If a decision design has specific implications for a key fitness function, it will be worth spending more time and effort conducting spikes (timed-boxed, experimental coding projects) to validate the archtectural aspects of the design. Some teams adopt set-based development, a practice in lean and agile processes for designing several solutions in parallel, leaving options open for future decisions in exchange for the cost of building multiple solutions.

## Review Fitness Functions

A fitness function review is a meeting with key business and technical stakeholders with the goal of updating fitness functions to meet design goals. Events, such as significant market or customer growth, a new area of functionality or business capability, or an overhaul of an existing part of the system can warrant a fitness function review.

A fitness function review generally includes the following:

- Reviewing existing fitness functions
- Checking the relevancy of the current fitness functions
- Determining change in the scale or magnitude of each fitness function
- Deciding if there are better approaches for measuring or testing the system's fitness functions
- Discovering new fitness functions that the system might need to support

Review your fitness functions at least once a year.

### PenultimateWidgets and the Enterprise Architecture Spreadsheet

When the architects for PenultimateWidgets decided to build a new project platform they first created a spreadsheet of all the desirable characteristics: scalability, security, resiliency, and a host of other "-ilities." But then they faced an age-old question: If they built the new architecture to support those features, how can they ensure that it maintains that support? As developers add new features, how would they keep unexpected degradation of these important characteristics from occurring?

The solution was to create fitness functions for each of the concerns in the spreadsheet, reformulating some of them to meet objective evaluation criteria. Rather than occasional, ad hoc verification of their important criteria, they wired the fitness functions into their deployment pipeline (discussed more fully in Chapter 3).

While software architects are interested in exploring evolutionary architectures, we aren't attempting to model biological evolution. Theoretically, we could build an architecture that randomly changed one of its bits (mutation) and redeployed itself. After a few million years, we would likely have a very interesting architecture. However, we don't have millions of years to wait.

We want our architecture to evolve in a guided way, so we place constraints on different aspects of the architecture to reign in undesirable evolutionary directions. A good example is dog breeding: By selecting the characteristics we want, we can create a vast number of different shaped canines in a relatively short amount of time.

We cover more aspects of operationalizing fitness functions in the next chapter. In Chapter 6, we combine fitness functions with all the other architecture dimensions.

# CHAPTER 3
# Engineering Incremental Change

> An evolutionary architecture supports guided, *incremental* change across multiple dimensions.
>
> —our definition

In 2010, Jez Humble and Dave Farley released *Continuous Delivery*, a collection of practices to enhance the engineering efficiency in software projects. They provided the *mechanism* for building and releasing software via automation and tools but not the *structure* of how to design evolvable software. Evolutionary architecture assumes these engineering practices as prerequisites but addresses how to utilize them to help design evolvable software.

Our definition of evolutionary architecture implies *incremental change*, meaning the architecture should facilitate change in small increments. This chapter describes architectures that support incremental change along with some of the engineering practices used to achieve incremental change, an important building block of evolutionary architecture. We discuss two aspects of incremental change: *development*, which covers how developers build software, and *operational*, which covers how teams deploy software.

Here is an example of the operational side of incremental change. We start with the fleshed out example of incremental change from Chapter 1, which includes additional details about the architecture and deployment environment. PenultimateWidgets, our seller of widgets, has a catalog page backed by a microservice architecture and engineering practices, as illustrated in Figure 3-1.

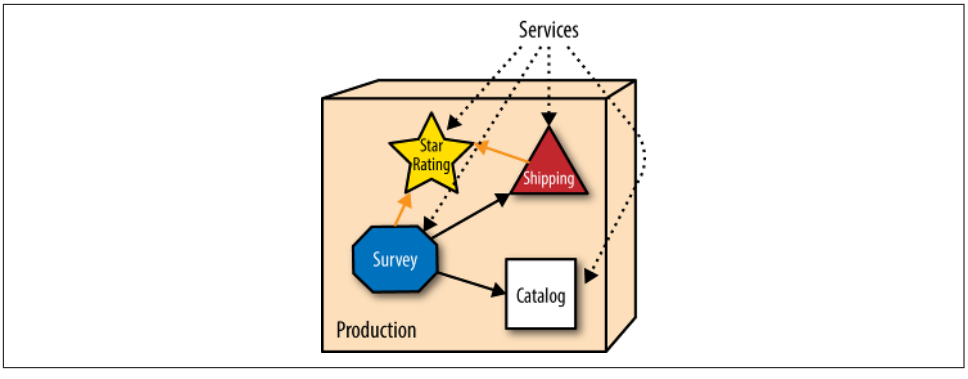



*Figure 3-1. Initial configuration of PenultimateWidgets' component deployment*

PenultimateWidgets' architects have implemented microservices that are operationally isolated from other services. Microservices implement a *share nothing* architecture: Each service is operationally distinct to eliminate technical coupling and therefore promote change at a granular level. PenultimateWidgets deploys all their services in separate containers to trivialize operational changes.

The website allows users to rate different widgets with star ratings. But other parts of the architecture also need ratings (customer service representatives, shipping provider evaluation, and so on), 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 significant upgrade, as shown in Figure 3-2.

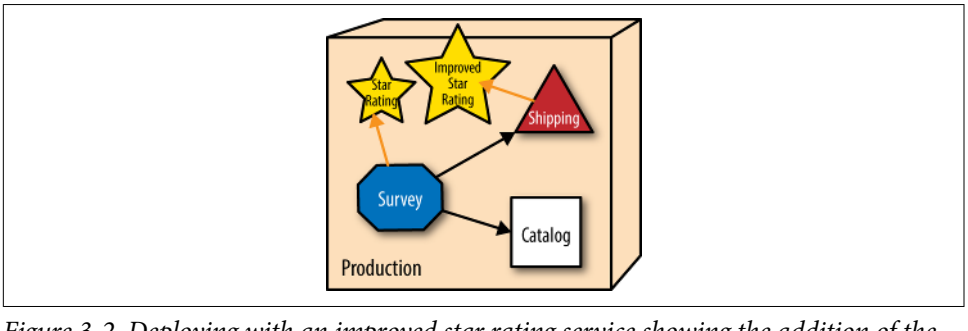



*Figure 3-2. Deploying with an improved star rating service showing the addition of the half-star rating*

The services that utilize ratings aren't required to migrate to the improved rating service but can gradually transition to the better service when convenient. As time progresses, more parts of the ecosystem that need ratings move to the enhanced version. Part of PenultimateWidgets' DevOps practices include architectural monitoring—monitoring 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, as shown in Figure 3-3.

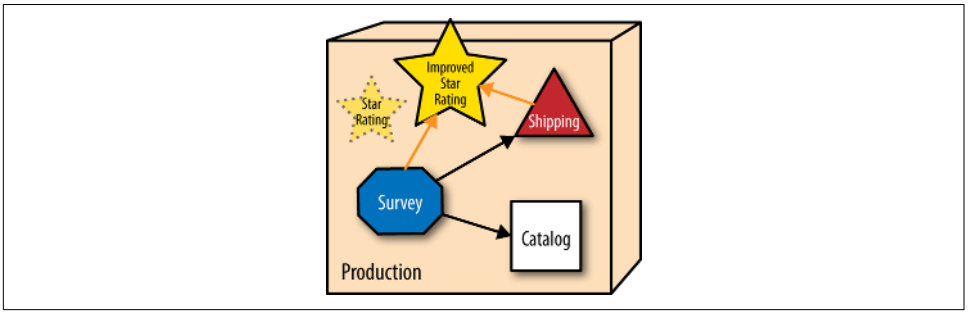



*Figure 3-3. All services now use the improved star rating service*

The mechanical ability to evolve is one the key components of an evolutionary architecture. Let's dig one level deeper in the abstraction above.

PenultimateWidgets has a fine-grained microservices architecture, where each service is deployed using a container (like Docker) and using a service template to handle infrastructure coupling. Applications within PenultimateWidgets consist of routes between instances of services running—a given service may have multiple instances to handle operational concerns like on-demand scalability. This allows architects to host different versions of services in production and control access via routing. When a deployment pipeline deploys a service, it registers itself (location and contract) with a service discovery tool. When a service needs to find another service, it uses the discovery tool to learn the location and version suitability via the contract.

When the new star rating service is deployed, it registers itself with the service discovery tool and publishes its new contract. The new version of the service supports a broader range of values—specifically, half-point values—than the original. That means the service developers don't have to worry about restricting the supported values. If the new version requires a different contract for callers, it is typical to handle that within the service rather than burden callers with resolving which version to call. We cover that contract strategy in "Version Services Internally" on page 119.

When the team deploys the new service, they don't want to force the calling services to upgrade to the new service immediately. Thus, the architect temporarily changes the star-service endpoint into a proxy that checks to see which version of the service is requested and routes to the requested version. No existing services must change to use the rating service as they always have, but new calls can start taking advantage of the new capability. Old services aren't forced to upgrade and can continue to call the original service as long as they need it. As the calling services decide to use the new behavior, they change the version they request from the endpoint. Over time, the original version falls into disuse, and at some point, the architect can remove the old

version from the endpoint when it is no longer needed. Operations is responsible for scanning for services that no other services call anymore (within some reasonable threshold) and garbage collecting the unused services.

All the changes to this architecture, including the provisioning of external components such as the database, happen under the supervision of a deployment pipeline, removing the responsibility of coordinating the disparate moving parts of the deployment from DevOps.

This chapter covers the characteristics, engineering practices, team considerations, and other aspects of building architectures that support incremental change.

## Building Blocks

Many of the building blocks required for agility at the architecture level have become mainstream over the last few years under the umbrella of Continuous Delivery and its engineering practices.

Software architects have to determine how systems fit together, often by creating diagrams, with varying degrees of ceremony. Architects often fall into the trap of seeing software architecture as an *equation* they must solve. Much of the commercial tooling sold to software architects reinforces the mathematical illusion of certainty with boxes, lines, and arrows. While useful, these diagrams offer a 2D view—a snapshot of an ideal world—but we live in a 4D world. To flesh out that 2D diagram, we must add specifics. The `ORM` label Figure 3-4 becomes `JDBC 2.1`, evolving into a 3D view of the world, where architects prove their designs in a real production environment using real software. As Figure 3-4 illustrates, over time, changes in business and technology require architects to adopt a 4D view of architecture, making evolution a first-class concern.

Nothing in software is static. Take a computer, for example. Install an operating system and a nontrivial set of software on it, then lock it in a closet for a year. At the end of the year, retrieve it from the closet and plug it into the wall and Internet…and watch it install updates for a long time. Even though no one changed a single bit on the computer, *the entire world kept moving*; this is the dynamic equilibrium we described earlier. Any reasonable architecture plan must include evolutionary change.

When we know how to put architecture into production *and* upgrade it to incorporate inevitable changes (security patches, new versions of software, evolutions of the architecture, and so on) as needed, we've graduated to a 4D world. Architecture isn't a static equation but rather a snapshot of an ongoing process, as illustrated in Figure 3-4.

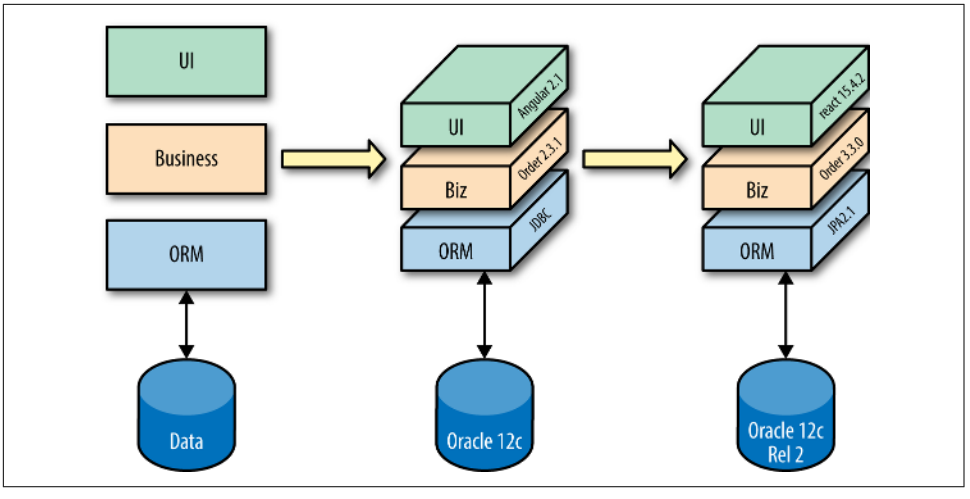



*Figure 3-4. Modern architecture must be deployable and changeable to survive the real world*

Continuous Delivery and the DevOps movement illustrate the need to implement an architecture and keep it current. There is nothing wrong with modeling architecture and capturing those efforts, but the model is merely the first step.

Architecture is abstract until operationalized, when it becomes a living thing.

Figure 3-4 illustrates the natural evolution of version upgrades and new tool choices. Architectures evolve in other ways as well, as we'll see in Chapter 6.

Architects cannot judge the long-term viability of any architecture until *design*, *implementation*, *upgrade*, and *inevitable change* are successful. And perhaps even enabled the architecture to withstand unusual occurrences based on incipient unknown unknowns, which we cover in Chapter 6.

## Testable

One of the oft ignored "-ilities" of software architecture is *testability*—can characteristics of the architecture submit to automated tests to verify veracity? Unfortunately, it is often difficult to test architecture parts due to lack of tool support.

However, some aspects of an architecture do yield to easy testing. For example, developers can test concrete architectural characteristics like coupling, develop guidelines, and eventually automate those tests.

Here is an example of a fitness function defined at the technical architecture dimension to control the directionality of coupling between components. In the Java ecosystem, JDepend is a metrics tool that analyzes the coupling characteristics of packages. Because JDepend is written in Java, it has an API that developers can leverage to build their own analysis via unit tests.

Consider the fitness function in Example 3-1, expressed as a JUnit test:

*Example 3-1. JDepend test to verify the directionality of package imports*

```
public void testMatch() {
    DependencyConstraint constraint = new DependencyConstraint();

    JavaPackage persistence = constraint.addPackage("com.xyz.persistence");
    JavaPackage web = constraint.addPackage("com.xyz.web");
    JavaPackage util = constraint.addPackage("com.xyz.util");

    persistence.dependsUpon(util);
    web.dependsUpon(util);

    jdepend.analyze();

    assertEquals("Dependency mismatch",
            true, jdepend.dependencyMatch(constraint));
    }
```

In Example 3-1, we define the packages in our application and then define the rules about imports. One of the bedeviling problems in component-based systems is component cycles—i.e., when component A references component B, which in turn references component A again. If a developer accidentally writes code that imports into `util` from `persistence`, this unit test will fail before the code is committed. We prefer building unit tests to catch architecture violations over using strict development guidelines (with the attendant bureaucratic scolding): It allows developers to focus more on the domain problem and less on plumbing concerns. More importantly, it allows architects to consolidate rules as executable artifacts.

Fitness functions can have any owner, including shared ownership. In the example shown in Example 3-1, the application team may own the directionality fitness function because it is a particular concern for that project. In the same deployment pipeline, fitness functions common across multiple projects may be owned by the security team. In general, the definition and maintenance of fitness functions is a shared responsibility between architects, developers, and any other role concerned with maintaining architectural integrity.

Many things about architecture are testable. Tools exist to test the structural characteristics of architecture such as JDepend (or a similar tool in the .NET ecosystem NDepend). Tools also exist for performance, scalability, resiliency, and a variety of

other architectural characteristics. Monitoring and logging tools also qualify: Any tool that helps assess some architectural characteristic qualifies as a fitness function.

Once they have defined fitness functions, architects must ensure that they are evaluated in a timely manner. Automation is the key to continual evaluation. A *deployment pipeline* is often used to evaluate tasks like this. Using a deployment pipeline, architects can define which, when, and how often fitness functions execute.

## Deployment Pipelines

*Continuous Delivery* describes the deployment pipeline mechanism. Similar to a continuous integration server, a deployment pipeline "listens" for changes, then runs a series of verification steps, each with increasing sophistication. Continuous Delivery practices encourage using a deployment pipeline as the mechanism to automate common project tasks, such as testing, machine provisioning, deployments, etc. Open source tools such as GoCD facilitate building these deployment pipelines.

### Continuous Integration Versus Deployment Pipelines

Continuous integration is a well-known engineering practice in agile projects that encourages developers to integrate as early and as often as possible. To facilitate continuous integration, tools such as ThoughtWorks CruiseControl and other commercial and open source offerings have emerged. Continuous integration provides an "official" build location, and developers enjoy the concept of a single mechanism to ensure working code. However, a continuous integration server also provides a perfect time and place to perform common project tasks such as unit testing, code coverage, metrics, functional testing, and so on. For many projects, the continuous integration server includes a list of tasks to perform whose successful culmination indicates build success. Large projects eventually build an impressive list of tasks.

Deployment pipelines encourage developers to split individual tasks into *stages*. A deployment pipeline includes the concept of multi-stage builds, allowing developers to model as many post-checkin tasks as necessary. This ability to separate tasks discretely supports the broader mandates expected of a deployment pipeline—to verify production readiness—compared to a continuous integration (CI) server primarily focused on integration. Thus, a deployment pipeline commonly includes application testing at multiple levels, automated environment provisioning, and a host of other verification responsibilities.

Some developers try to "get by" with a continuous integration server but soon find they lack the level of separation of tasks and feedback necessary.

A typical deployment pipeline automatically builds the deployment environment (a container like Docker or a bespoke environment generated by a tool like Puppet or Chef) as shown in Figure 3-5.

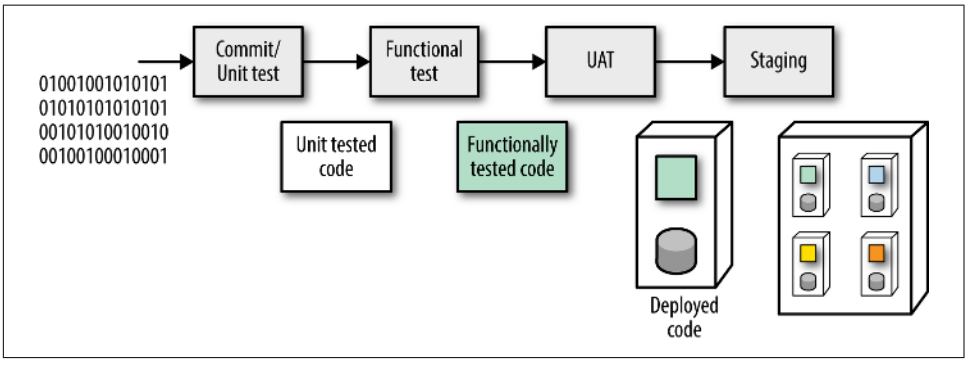



*Figure 3-5. Deployment pipeline stages*

By building the deployment image that the deployment pipeline executes, developers and operations have a high degree of confidence: The host computer (or virtual machine) is declaratively defined, and it's a common practice to rebuild it from nothing.

The deployment pipeline also offers an ideal way to execute the fitness functions defined for an architecture: It applies arbitrary verification criteria, has multiple stages to incorporate differing levels of abstraction and sophistication of tests, and runs every single time the system changes in any way. A deployment pipeline with fitness functions added is shown in Figure 3-6.

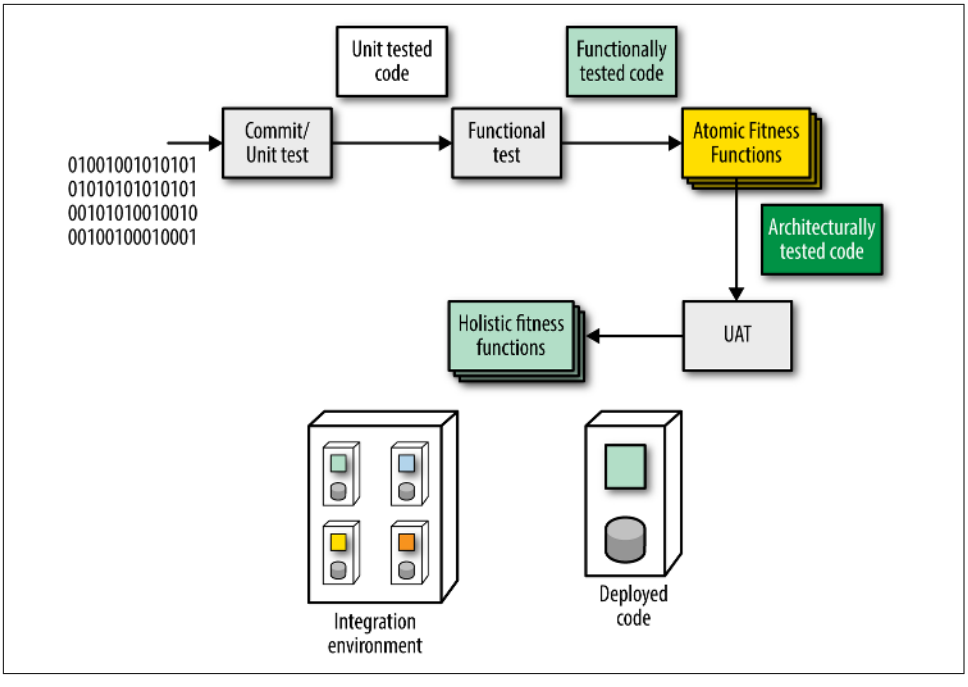



*Figure 3-6. A deployment pipeline with fitness functions added as stages*

Figure 3-6 shows a collection of atomic and holistic fitness functions with the latter in a more complex integration environment. Deployment pipelines can ensure the rules defined to protect architectural dimensions execute each time the system changes.

### PenultimateWidgets Deployment Pipelines

In Chapter 2, we described PenultimateWidgets' spreadsheet of requirements. Once they adopted some of the Continuous Delivery engineering practices, they realized that nonfunctional platform requirements work better in an automated deployment pipeline. To that end, service developers created a deployment pipeline to validate the fitness functions created both by the enterprise architects and by the service team. Now, each time the team makes a change to the service, a barrage of tests validates both the correctness of the code and its overall fitness within the architecture.

Another common practice in evolutionary architecture projects is continuous deployment—using a deployment pipeline to put changes into production contingent on successfully passing the pipeline's gauntlet of tests and other verifications. While continuous deployment is ideal, it requires sophisticated coordination: Developers must ensure changes deployed to production on an ongoing basis don't break things.

To solve this coordination problem, a *fan out* operation is commonly used in deployment pipelines where the pipeline runs several jobs in parallel, as shown in Figure 3-7.

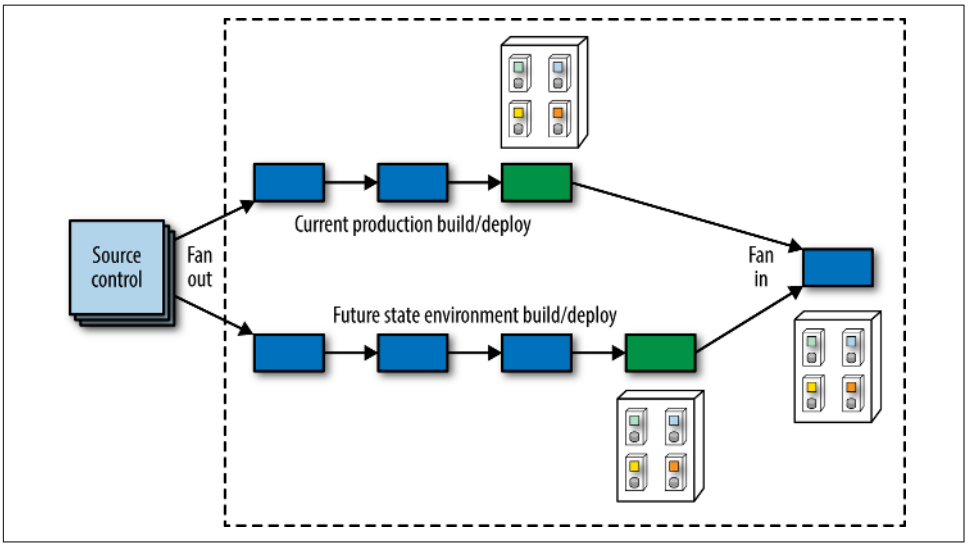



*Figure 3-7. Deployment pipeline fan out to test multiple scenarios*

As shown in Figure 3-7, when a team makes a change, they have to verify two things: They haven't negatively affected the current production state (because a successful deployment pipeline execution will deploy code into production) and their changes were successful (affecting the future state environment). A deployment pipeline fan out allows tasks (testing, deploy, and so on) to execute in parallel, saving time. Once the series of concurrent jobs illustrated in Figure 3-7 completes, the pipeline can evaluate the results and if everything is successful, perform a *fan in*, consolidating to a single thread of action to perform tasks like deployment. Note that the deployment pipeline may perform this combination of *fan out* and *fan in* numerous times whenever the team needs to evaluate a change in multiple contexts.

Another common issue with continuous deployment is business impact. Users don't want a barrage of new features showing up on a regular basis but would rather have them staged in a more traditional way such as a "Big Bang" deployment. A common way to accommodate both continuous deployment and staged releases is to use *feature toggles*. By implementing new features hidden underneath feature toggles, developers can safely deploy the feature to production without worrying about users seeing it prematurely.

### QA in Production

One beneficial side effect of habitually building new features using feature toggles is the ability to perform QA tasks in production. Many companies don't realize they can use their production environment for exploratory testing. Once a team becomes comfortable using feature toggles, they can deploy those changes to production since most feature toggle frameworks allow developers to route users based on wide variety of criteria (IP address, access control list (ACL), etc.). If a team deploys new features within feature toggles to which only the QA department has access, they can test in production.

Using deployment pipelines in engineering practices, architects can easily apply project fitness functions. Figuring out which stages are needed is a common challenge for developers designing a deployment pipeline. Casting the project's architectural concerns (including evolvability) as fitness functions provides many benefits:

- Fitness functions are designed to have objective, quantifiable results
- Capturing all concerns as fitness function creates a consistent enforcement mechanism
- Having a list of fitness functions allows developers to most easily design deployment pipelines

Determining when in the project's build cycle to run fitness functions, which ones to run, and the proper context is a nontrivial undertaking. However, once the fitness functions inside a deployment pipeline are in place, architects and developers have a high level of confidence that evolutionary changes won't violate the project guidelines. Architectural concerns are often poorly elucidated and sparsely evaluated, often subjectively; creating them as fitness functions allows better rigor and therefore better confidence in the engineering practices.

## Combining Fitness Function Categories

Fitness function categories often intersect when implementing them in mechanisms like deployment pipelines. Here are some common mashups of fitness function categories, along with examples.

*atomic + triggered*
: This type of fitness function is exemplified by unit and functional tests run as part of software development. Developers run them to verify changes, and an automation mechanism, such as a deployment pipeline, applies continuous integration to ensure timeliness. A common example of this type of fitness function

is a unit test that verifies some aspect of the architectural integrity of the application architecture, such as circular dependencies or cyclomatic complexity.

*holistic + triggered*
: Holistic, triggered fitness functions are designed to run as part of integration testing via a deployment pipeline. Developers design these tests specifically to test how different aspects of the system interact in well-defined ways. For example, developers may be curious to see what kind of impact tighter security has on scalability. Architects design these tests to intentionally test some integration characteristic in the code base because breakages indicate some architectural shortcoming. Like all triggered tests, developers typically run these fitness functions both during development and as part of a deployment pipeline or continuous integration environment. Generally, these are tests and metrics that have well-known outcomes.

*atomic + continual*
: Continual tests run as part of the architecture, and developers design around their presence. For example, architects might be concerned that all REST endpoints support the proper verbs, exhibit correct error handling, and support metadata properly and therefore build a tool that runs continually to call REST endpoints (just as normal clients would) to verify the results. The *atomic* scope of these fitness functions suggests that they test just one aspect of the architecture, but *continual* indicates that the tests run as part of the overall system.

*holistic + continual*
: Holistic, continual fitness functions test multiple parts of the system all the time. Basically, this mechanism represents an agent (or another client) in a system that constantly assesses a combination of architectural and operational qualities. An outstanding example of a real-world continual holistic fitness function is Netflix's Chaos Monkey. When Netflix designed their distributed architecture, they designed it to run on the Amazon Cloud. But engineers were concerned what sort of odd behavior could occur because they have no direct control over their operations, such as high latency, availability, elasticity, and so on, in the Cloud. To assuage their fears, they created Chaos Monkey, eventually followed by an entire open source Simian Army. Chaos Monkey "infiltrates" an Amazon data center and starts making unexpected things happen: Latency goes up, reliability goes down, and other chaos ensues. By designing with Chaos Monkey in mind, each team must build resilient services. The RESTful verification tool mentioned in in the previous section exists as the Conformity Monkey, which checks each service for architect-defined best practices.

Note that Chaos Monkey isn't a testing tool run on a schedule—it runs continuously within Netflix's ecosystem. Not only does this force developers to build systems that withstand problems, it tests the system's validity continually. Having this constant

verification built into the architecture has allowed Netflix to build one of the the most robust systems in the world. The Simian Army provides an excellent example of a holistic continual operational fitness function. It runs against multiple parts of the architecture at once, ensuring architectural characteristics (resiliency, scalability, etc.) are maintained.

Holistic, continual fitness functions are the most complex fitness functions for developers to implement but can provide great power, as the following case study illustrates.

## Case Study: Architectural Restructuring while Deploying 60 Times/Day

GitHub is a well-known developer-centric website with aggressive engineering practices, deploying on average 60 times a day. They describe a problem in their blog "Move Fast and Fix Things" that will make many architects shudder in horror. It turns out that GitHub has long used a shell script wrapped around command-line Git to handle merges, which works correctly but doesn't scale well enough. The Git engineering team built a replacement library for many command-line Git functions called libgit2 and implemented their merge functionality there, thoroughly testing it locally.

But now they must deploy the new solution into production. This behavior has been part of GitHub since its inception and has worked flawlessly. The last thing the developers want to do is introduce bugs in existing functionality, but they must address technical debt as well.

Fortunately, GitHub developers created and open sourced Scientist, a framework that provides holistic, continual testing to vet changes to code. Example 3-2 gives us the structure of a `Scientist` test.

*Example 3-2. Scientist setup for an experiment*

```
require "scientist"

class MyWidget
  include Scientist

  def allows?(user)
    science "widget-permissions" do |e|
      e.use { model.check_user(user).valid? } # old way
      e.try { user.can?(:read, model) } # new way
    end # returns the control value
  end
end
```

In Example 3-2, the developer takes the existing behavior and encapsulates it with the `use` block (called the *control*) and adds the experimental behavior to the `try` block (called the *candidate*). The `science` block handles the following details during the invocation of the code:

*Decides whether to run the* `try` *block*
: Developers configure Scientist to determine how the experiment runs. For example, in this case study—the goal of which was to update their merge functionality—1% of random users tried the new merge functionality. In either case, Scientist *always* returns the results of the `use` block, ensuring the caller always receives the existing behavior in case of differences.

*Randomizes the order that* `use` *and* `try` *blocks run*
: Scientist does this to prevent accidentally masking bugs due to unknown dependencies. Sometimes the order or other incidental factors can cause false positives; by randomizing their order, the tool makes those faults less likely.

*Measures the durations of all behaviors*
: Part of Scientist's job is A/B performance testing, so monitoring performance is built in. In fact, developers can use the framework piecemeal—for example, they can use it to measure calls without performing experiments.

*Compares the result of* `try` *to the result of* `use`
: Because the goal is refactoring existing behavior, Scientist compares and logs the results of each call to see if differences exist.

*Swallows (but logs) any exceptions raised in the* `try` *block*
: There's always a chance that new code will throw unexpected exceptions. Developers never want end users to see these errors, so the tool makes them invisible to the end user (but logs it for developer analysis).

*Publishes all this information*
: Scientist makes all its data available in a variety of formats.

For the merge refactoring, the GitHub developers used the following invocation to test the new implementation (called `create_merge_commit_rugged`), as shown in Example 3-3.

*Example 3-3. Experimenting with a new merge algorithm*

```
def create_merge_commit(author, base, head, options = {})
  commit_message = options[:commit_message] || "Merge #{head} into #{base}"
  now = Time.current

  science "create_merge_commit" do |e|
    e.context :base => base.to_s, :head => head.to_s, :repo => repository.nwo
```

```
    e.use { create_merge_commit_git(author, now, base, head, commit_message) }
    e.try { create_merge_commit_rugged(author, now, base, head, commit_message) }
  end
end
```

In Example 3-3, the call to `create_merge_commit_rugged` occurred in 1% of invocations, but, as noted in this case study, at GitHub's scale, all edge cases appear quickly.

When this code executes, end users always receive the correct result. If the `try` block returns a different value from `use`, it is logged, and the `use` value is returned. Thus, the worse case for end users is exactly what they would have gotten before the refactoring. After running the experiment for 4 days and experiencing no slow cases or mismatched results for 24 hours, they removed the old merge code and left the new in place.

From our perspective, Scientist is a fitness function. This case study is an outstanding example of the strategic use of a holistic, continous fitness function to allow developers to refactor a critical part of their infrastructure with confidence. They changed a key part of their architecture by running the new version alongside the existing, essentially turning the legacy implementation into a consistency test.

In general, most architectures will have a large number of atomic fitness functions and a few key holistic ones. The determining factor of atomicity comes down to what developers are testing and how broad are the results.

## Conflicting Goals

The agile software development process has taught us that the sooner a developer can detect problems, the less effort is required to fix them. One of the side effects of broadly considering all the dimensions in software architecture is the early identification of goals that conflict across dimensions. For example, developers at an organization may want to support the most aggressive pace of change to support new features. Fast change to code implies fast changes to database schemas, but the database administrators are more concerned about stability because they are building a data warehouse. The two evolution goals conflict across the technical and data architecture.

Obviously, some compromise must occur, taking into account the myriad factors that affect the underlying business. Using architecture dimensions as a technique for identifying portions of concern in architecture (plus fitness functions to evaluate them) allows an apples-to-apples comparison, making the prioritization exercise more informed.

Conflicting goals are inevitable. However, discovering and quantifying those conflicts early allows architects to make better informed decisions and create more clearly defined goals and principles.

## Case Study: Adding Fitness Functions to PenultimateWidgets' Invoicing Service

Our exemplar company, PenultimateWidgets, has an architecture that includes a service to handle invoicing. The invoicing team wants to replace outdated libraries and approaches but wants to ensure these changes don't impact other teams ability to integrate with them.

The invoicing team identified the following needs:

*Scalability*
: While performance isn't a big concern for PenultimateWidgets, they handle invoicing details for several resellers, so the invoicing service must maintain availability service-level agreements.

*Integration with other services*
: Several other services in the PenultimateWidgets ecosystem use invoicing. The team wants to make sure integration points don't break while making internal changes.

*Security*
: Invoicing means money, and security is always an ongoing concern.

*Auditability*
: Some state regulations require that changes to taxation code be verified by an independant accountant.

The invoicing team uses a continuous integration server and recently upgraded to on-demand provisioning of the environment that runs their code. To implement evolutionary architecture fitness functions, they implement a deployment pipeline to replace the continuous integration server, allowing them to create several stages of execution, as shown in Figure 3-8.

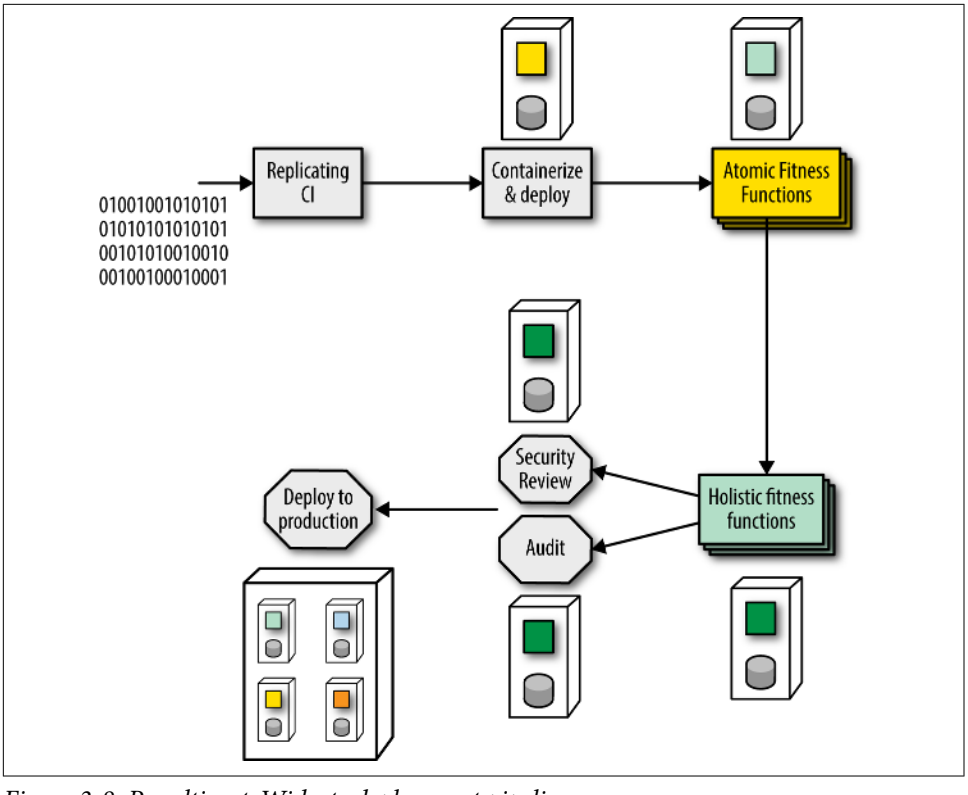



*Figure 3-8. PenultimateWidgets deployment pipeline*

PenultimateWidgets' deployment pipeline consists of six stages.

*Stage 1—Replicating CI*
: The first stage replicates the behavior of the former CI server, running unit, and functional tests.

*Stage 2—Containerize and Deploy*
: Developers use the second stage to build containers for their service, allowing deeper levels of testing, including deploying the containers to a dynamically created test environment.

*Stage 3—Atomic Fitness Functions*
: In the third stage atomic fitness functions, including automated scalability tests and security penetration testing, are executed. This stage also runs a metrics tool that flags any code within a certain package that developers changed, pertaining to auditability. While this tool doesn't make any deteriminations, it assists a later stage in narrowing in on specific code.

*Stage 4—Holistic Fitness Functions*
: The fourth stage focuses on holistic fitness functions, including testing contracts to protect integration points and some further scalability tests.

*Stage 5a—Security Review (manual)*
: This stage includes a manual stage by a specific security group within the organization to review, audit, and assess any security vulnerabilities in the codebase. Deployment pipelines allow the definition of manual stages, triggered on demand by the relevant security specialist.

*Stage 5b—Auditing (manual)*
: PenultimateWidgets is based in Springfield, where the state mandates specific auditing rules. The invoicing team builds this manual stage into their deployment pipeline, which offers several benefits. First, treating auditing as a fitness function allows developers, architects, auditors, and others to think about this behavior in a unified way—a necessary evaluation to deterimine the system's correct function. Second, adding the evaluation to the deployment pipeline allows developers to assess the engineering impact of this behavior compared equally to other automated evaluations within the deployment pipeline.

  For example, if the security review happens weekly but auditing happens only monthly, the bottleneck to faster releases is clearly the auditing stage. By treating both security and audit as stages in the deployment pipeline, decisions concerning both can be addressed more rationally: Is it worth value to the company to increase release cadence by having consultants perform the necessary audit more often?

*Stage 6—Deployment*
: The last stage is deployment into the production environment. This is a automated stage for PenultimateWidgets and is triggered only if the two upstream manual stages (*security review* and *audit*) report success.

Interested architects at PenultimateWidgets receive a weekly automatically generated report about the success/failure rate of the fitness functions, helping them gauge health, cadence, and other factors.

## Hypothesis- and Data-Driven Development

The GitHub example in "Case Study: Architectural Restructuring while Deploying 60 Times/Day" on page 37 using the `Scientist` framework is an example of *data-driven development*—allow data to drive changes and focus efforts on technical change. A similar approach that incorporates the business rather than technical concerns is *hypothesis-driven development*.

In the week between Christmas 2013 and New Year's Day 2014, Facebook encountered a problem: More photos were uploaded to Facebook in that week than all the photos on Flickr, and more than a million of them were flagged as offensive. Facebook allows users to flag photos they believe potentially offensive and then reviews them to determine objectively if they are. But this dramatic increase in photos created a problem: There was not enough staff to review the photos.

Fortunately, Facebook has modern DevOps and the ability to perform experiments on their users. When asked about the chances a typical Facebook user has been involved in an experiment, one Facebook engineer claimed "Oh, one hundred percent —we routinely have more than twenty experiments running at time." They used this experimental capability to ask users follow-up on questions about *why* photos were deemed offensive and discovered many delightful quirks of human behavior. For example, people don't like to admit that they look bad in a photo but will freely admit that the photographer did a poor job. By experimenting with different phrasing and questions, the engineers could query their actual users to determine why they flagged a photo as offensive. In a relatively short amount of time, Facebook shaved off enough false positives to restore offensive photos to a manageable problem by building a platform that allowed for experimentation.

In the book *Lean Enterprise* (O'Reilly, 2014), Barry O'Reilly describes the modern process of *hypothesis-driven development*. Under this process, rather than gathering formal requirements and spending time and resources building features into applications, teams should leverage the scientific method instead. Once teams have created the minimal viable product version of an application (whether as a new product or by performing maintenance work on an existing application), they can build hypotheses during new feature ideation rather than requirements. Hypothesis-driven devlopment hypotheses are couched in terms of the hypothesis to test, what experiments can determine the results, and what validating the hypothesis means to future application development.

For example, rather than change the image size for sales items on a catalog page because a business analyst thought it was a good idea, state it as a hypothesis instead: If we make the sales images bigger, we hypothesize that it will lead to a 5% increase in sales for those items. Once the hypothesis is in place, run experiments via A/B testing —one group with bigger sales images and one without—and tally the results.

Even agile projects with engaged business users incrementally build themselves into a bad spot. An individual decision by a business analyst may make sense in isolation, but when combined with other features may ultimately degrade the overall experience. In an excellent case study, mobile.de followed a logical path of accruing new features haphazardly to the point where sales were diminishing, at least in part because their UI had become so convoluted, as is often the result of development continuing on mature software products. Several different philosophical approaches

were: more listings, better prioritization, or better grouping. To help them make this decision, they built three versions of the UI and allowed their users to decide.

The engine that drives agile software methodologies is the nested feedback loop: testing, continuous integration, iterations, etc. And yet, the part of the feedback loop that incorporates the ultimate users of the application has eluded teams. Using hypothesis-driven development, we can incorporate users in an unprecedented way, learning from behavior and building what users really find valuable.

Hypothesis-driven development requires the coordination of many moving parts: evolutionary architecture, modern DevOps, modified requirements gathering, and the ability to run multiple versions of an application simultaneously. Service-based architectures (like microservices) usually achieve side-by-side versions by intelligent routing of services. For example, one user may execute the application using a particular constellation of services while another request may use an entirely different set of instances of the same services. If most services include many running instances (for scalability, for example), it becomes trivial to make some of those instances slightly different with enhanced functionality, and to route some users to those features.

Experiments should run long enough to yield significant results. Generally, it is preferable to find a measurable way to determine better outcomes rather than annoy users with things like pop-up surveys. For example, does one hypothesized workflow allow the user to complete a task with fewer keystrokes and clicks? By silently incorporating users into the development and design feedback loop, you can build much more functional software.

## Case Study: What to Port?

One particular PenultimateWidgets application has been a workhorse, developed as a Java Swing application over the better part of a decade and continually growing new features. The company decided to port it to the web application. However, now the business analysts face a difficult decision: How much of the existing sprawling functionality should they port? And, more practically, what order should they implement the ported features of the new application to deliver the most functionality quickly?

One of the architects at PenultimateWidgets asked the business analysts what the most popular features were, and they had no idea! Even though they have been specifying the details of the application for years, they had no real understanding of how users used the application. To learn from users, the developers released a new version of the legacy application with logging enabled to track which menu features users actually used.

After a few weeks, they harvested the results, providing an excellent road map of what features to port and in what order. They discovered that the invoicing and customer lookup features were most commonly used. Surprisingly, one subsection of the appli-

cation that had taken great effort to build had very little use, leading the team to decide to leave that functionality out of the new web application.

CHAPTER 4

# Architectural Coupling

Discussions about architecture frequently boil down to coupling: how the pieces of the architecture connect and rely on one another. Many architects decry coupling as a necessary evil, but it's difficult to build complex software without relying on (and coupling with) other components. Evolutionary architecture focuses on appropriate coupling—how to identify which dimensions of the architecture should be coupled to provide maximum benefit with minimal overhead and cost.

## Modularity

First, let's untangle some of the common terms used and overused in discussions about architecture. Different platforms offer different reuse mechanisms for code, but all support some way of grouping related code together into *modules*. *Modularity* describes a logical grouping of related code. Modules in turn may be packaged in different physical ways. *Components* are the physical packaging of modules. Modules imply *logical* grouping, while components imply *physical* partitioning.

Developers find it useful to further subdivide *components* based on engineering practices, including build and deployment considerations. One kind of component is a *library*, which tends to run in the same memory address as the calling code and communicates via language function call mechanisms. Libraries are usually compile-time dependencies. Most concerns around libraries exist in application architecture, as most complex applications consist of a variety of components. The other type of component, called a *service*, tends to run in its own address space and communicates via low-level networking protocols like TCP/IP or higher-level formats like simple object access protocol (SOAP) or representational state transfer (REST). Concerns around services tend to come up most commonly in integration architecture, making these runtime dependencies.

All module mechanisms facilitate code reuse, and it is wise to try to reuse code at all levels, from individual functions all the way up to encapsulated business platforms.

## Architectural Quanta and Granularity

Software systems are bound together in a variety of ways. As software architects, we analyze software using many different perspectives. But component-level coupling isn't the only thing that binds software together. Many business concepts semantically bind parts of the system together, creating *functional cohesion*. To successfully evolve software, developers must consider *all* the coupling points that could break.

As defined in physics, the *quantum* is the minimum amount of any physical entity involved in an interaction. An *architectural quantum* is an independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function properly. In a monolithic architecture, the quantum is the entire application; everything is highly coupled and therefore developers must deploy it en mass.

### Domain-Driven Design's Bounded Context

Eric Evans's book Domain-Driven Design has deeply influenced modern architectural thinking. *Domain-driven design* (DDD) is a modeling technique that allows for organized decomposition of complex problem domains. DDD defines the *bounded context*, where everything related to the domain is visible internally but opaque to other bounded contexts. Before DDD, developers sought holistic reuse across common entities within the organization. Yet, creating common shared artifacts causes a host of problems, such as coupling, more difficult coordination, and increased complexity. The *bounded context* concept recognizes that each entity works best within a localized context. Thus, instead of creating a unified `Customer` class across the entire organization, each problem domain can create their own, and reconcile differences at integration points. DDD influenced several modern architectural styles, along with related factors like team organization (described in "Introducing PenultimateWidgets and Their Inverse Conway Moment" on page 13 in Chapter 1).

In contrast, a microservices architecture defines physical bounded contexts between architectural elements, encapsulating all the parts that might change. This type of architecture is designed to allow incremental change. In a microservices architecture, the bounded context serves as the quantum boundary and includes dependent components such as database servers. It may also include architecture components such as search engines and reporting tools—anything that contributes to the delivered functionality of the service, as shown in Figure 4-1.

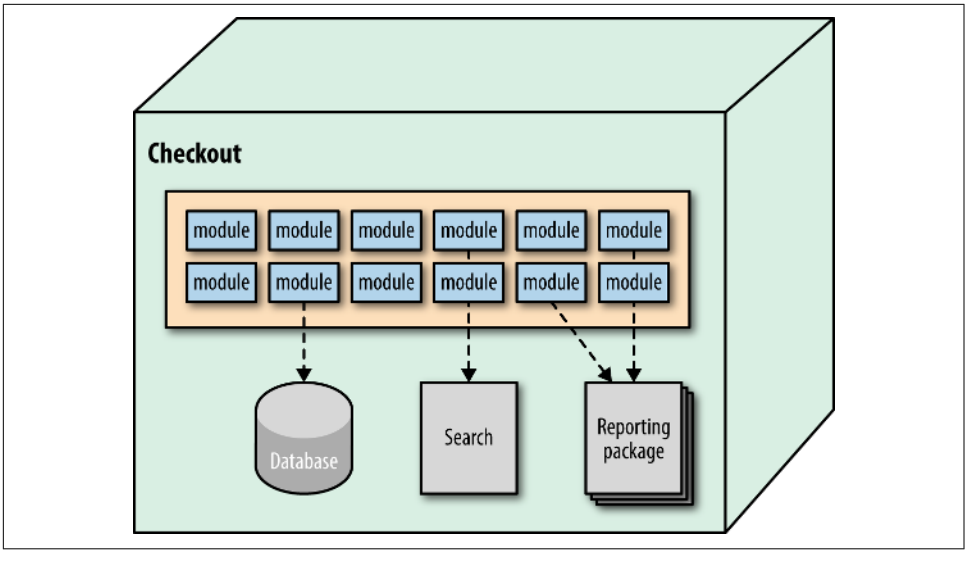



*Figure 4-1. The architectural quantum in microservices encompasses the service and all its dependent parts*

In Figure 4-1, the service includes code components, a database server, and a search engine component. Part of the bounded context philosophy of microservices operationalizes all the pieces of a service together, leaning heavily on modern DevOps practices. In the following section, we investigate some common architectural patterns and their typical quantum boundaries.

Traditionally isolated roles such as architect and operations must coordinate in an evolutionary architecture. Architecture is abstract until operationalized; developers must pay attention to how their components fit together in the real world. Regardless of which architecture pattern developers choose, architects should also explicitly define their quantum size. Small quanta implies faster change because of small scope. Generally, small parts are easier to work with than big ones. Quantum size determines the lower bound of the incremental change possible within an architecture.

As in physics, four fundamental interactions exist in nature: *gravitational*, *electromagnetic*, *strong*, and *weak*. The *strong* nuclear force, which holds atoms (and therefore ordinary matter) together, is notable for its strength. Breaking it unleashes much of the power of nuclear fission. Similarly, some architectural components are extremely difficult to break into smaller pieces. Metaphorically, they exhibit strong nuclear force. One of the keys to building evolutionary architectures lies in determining natural component granularity and coupling bewteen components to fit the capabilities they want to support via the software architecture.

In evolutionary architecture, architects deal with *architectural quanta*, the parts of a system held together by hard-to-break forces. For example, transactions act like a strong nuclear force, binding together otherwise unrelated pieces. While it is possible for developers to break apart a transactional context, it is a complex process and often leads to incidental complications like distributed transactions. Similarly, parts of a business might be highly coupled, and breaking the application into smaller architectural components may not be desirable.

Figure 4-2 summarizes the relationship between these terms.

## Monolithic Listing

We worked on a project for several years centered around automobile auctions. Not surprising, one of the large classes in the system was `Listing`, which grew into a monster. Developers undertook several technical refactoring exercises to find ways to break up the huge class because it was causing coordination problems. Finally, a scheme was hatched to break out one of the key parts, `Vendor`, into its own class. While the technical refactoring was a success, problems emerged between the interactions by developers and business analysts: developers kept talking about changes to `Vendor`, which wasn't a separate entity in their world. Developers violated what Eric Evans in *DDD* calls *ubiquitous language* on the project—make sure that all the terms on the team mean the same thing. While it made a few things more convenient for developers to split the functionality, the semantic coupling that defined the business process was violated, making our job more difficult.

Eventually, we unrefactored the `Listing` class back into a single large entity, because the software project revolved around it. We solved the coordination problem by treating `Listing` differently. Changes to `Listing` caused the continuous integration server to automatically generate a message to interested teams to encourage aggressive integration. Thus, we solved the coordination problem with an engineering practice rather than an architectural structure.

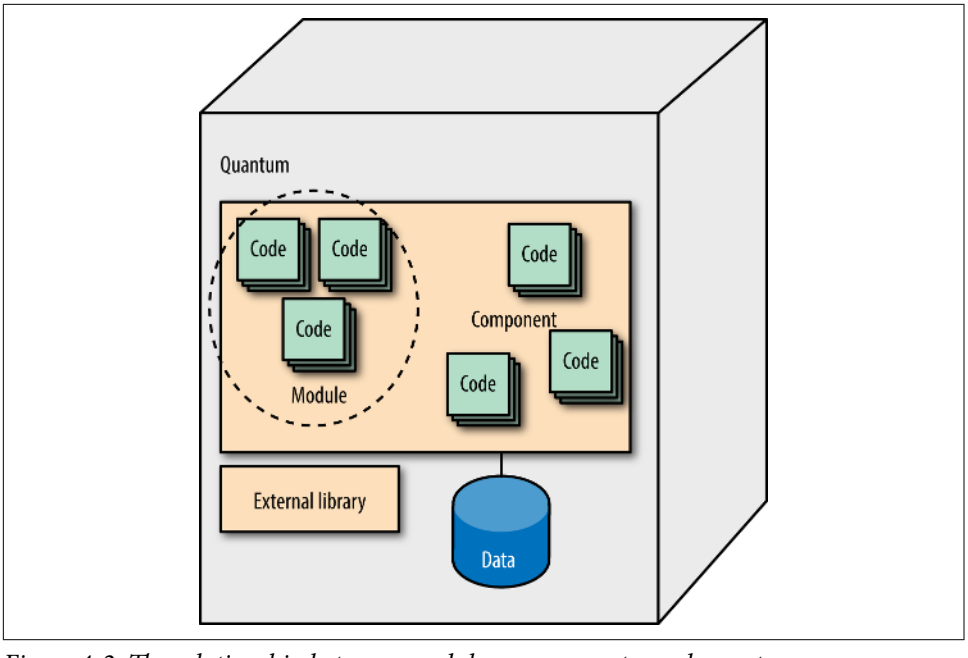



*Figure 4-2. The relationship between modules, components, and quanta*

As shown in Figure 4-2, the outermost container is the *quantum*: the deployable unit that includes all the facilities required for the system to function properly, including data. Within the quantum, several *components* exist, each consisting of code (classes, packages, namespaces, functions, and so on). An external component (from an open source project) also exists as a *library*, a component packaged for reuse within a given platform. Of course, developers can mix and match all possible combinations of these common building blocks.

## Evolvability of Architectural Styles

Software architecture exists at least partially to enable certain types of evolution across specific dimensions—easier change is one of the reasons for architecture patterns. Different architectural patterns have different inherent quantum sizes, which impact their ability to evolve. In this section, we investigate several popular architecture patterns and evaluate their inherent quantum size, along with their impact on the architecture's natural ability to evolve based on our three evolutionary criteria: incremental change, fitness functions, and appropriate coupling.

Note that while the architectural pattern is critical for successful evolution, it isn't the only determining factor. The inherent characteristics of the pattern must be combined with the additional characteristics defined for the system to fully define the dimensions of evolvability.

## Big Ball of Mud

First, consider the degenerate case of a chaotic system with no discernible architecture, colloquially known as the Big Ball of Mud antipattern. While typical architectural elements like frameworks and libraries may exist, developers haven't built structure on purpose. These systems are highly coupled, leading to rippling side effects when changes occur. Developers created highly coupled classes with poor modularity. Database schemas snaked into the UI and other parts of the system, effectively insulating them against change. DBAs spent the last decade avoiding refactoring by stitching together tightly bound join tables. Likely driven by draconian budget constraints, operations crams as many systems together as possible and deals with the operational coupling.

Figure 4-3 shows a class coupling diagram that exemplifies the Big Ball of Mud: each node represents a class, the lines represent coupling (either inward or outward) and the boldness of the line indicates the number of connections.

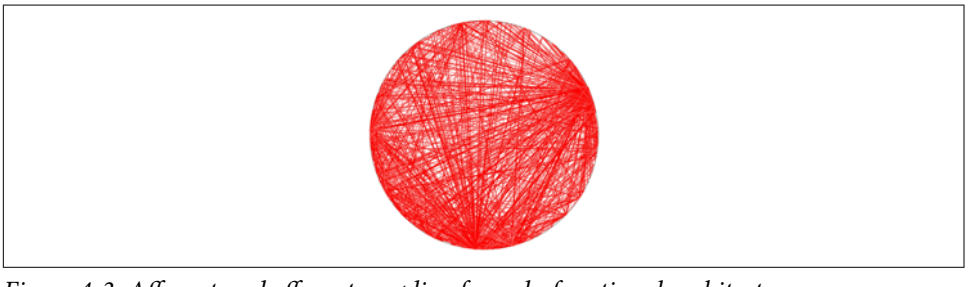


*Figure 4-3. Afferent and efferent coupling for a dysfunctional architecture*

Changing any part of the application depicted in Figure 4-3 (taken from a real project) presents intense challenges. Because so much exuberant coupling exists between classes, it is virtually impossible to modify one part of the application without impacting other parts. Thus, from an evolvability standpoint, this architecture scores extremely low. Developers who need to change data access throughout the application must hunt down all the places it exists and change them, risking missing some places.

From an evolution standpoint, this architecture fails each criteria drastically:

*Incremental change*

Making any change in this architecture is difficult. Related code is scattered throughout the system, meaning changes to one component will cause unexpected breakages in other components. Fixing those breakages will generate more breakage, a rippling effect that never ends.

*Guided change with fitness functions*
: Building fitness functions for this architecture is difficult because no clearly defined partitioning exists. To build protective functions, developers must be able to identify parts to protect, and no structure exists in this architecture outside low-level functions or classes.

*Appropriate coupling*
: This architectural style is a good example of *inappropriate* coupling. No architectural advantages result from building software like this.

In this dire state, change is difficult and expensive. Essentially, because each part of the system is highly coupled to every other part, the quantum is the entire system—no part is easy to change because every part affects every other part.

## Monoliths

Monolithic architectures often contain a large amount of highly coupled code. We investigate several variations of this architectural style, based on organization.

### Unstructured monoliths

This architectural pattern includes several different variations, including systems with essentially independent classes coordinating as seen in Figure 4-4.

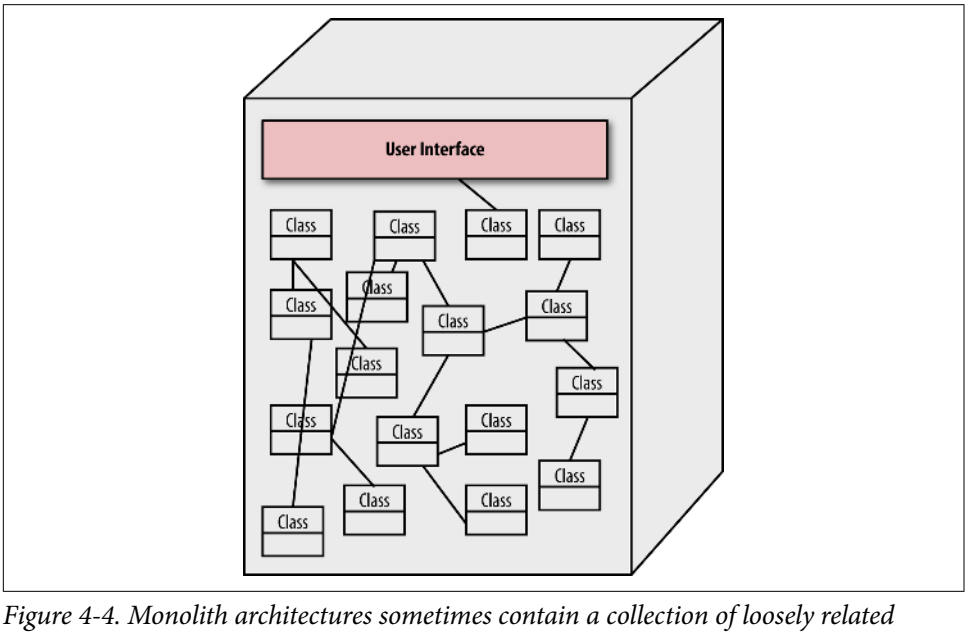



*Figure 4-4. Monolith architectures sometimes contain a collection of loosely related classes*

In Figure 4-4, different modules handle different tasks independently, utilizing shared classes for common functionality. A lack of coherent overarching structure hinders change in this architecture.

*Incremental change*
: Large quantum size hinders incremental change because high coupling requires deploying large chunks of the application. Deploying a single component is difficult because each component is highly coupled to others, requiring change to those components as well.

*Guided change with fitness functions*
: Building fitness functions for monoliths is difficult but not impossible. Because this architectural pattern has existed for a long time, many tools and testing practices have grown around it that can be used to create fitness functions. However, common guided change targets, such as performance and scalability, have traditionally been the Achilles' heel of monolithic architectures. While developers easily understand monoliths, building good scalability and performance is difficult, largely due to inherent coupling.

*Appropriate coupling*
: A monolithic architecture, with little internal structure outside simple classes, exhibits coupling almost as bad as a Big Ball of Mud. Thus, changes in one portion of the code may have unanticipated side effects in sometimes far-reaching parts of the code base.

Though the evolvability of this architecture is a milder version of the "Big Ball of Mud" on page 52. It is quite easy for this architecture to degenerate because there are few structural constraints to prevent it.

### Layered architecture

Other monolith architectures utilize a more structured approach to creating a layered architecture, one variation of which appears in Figure 4-5.

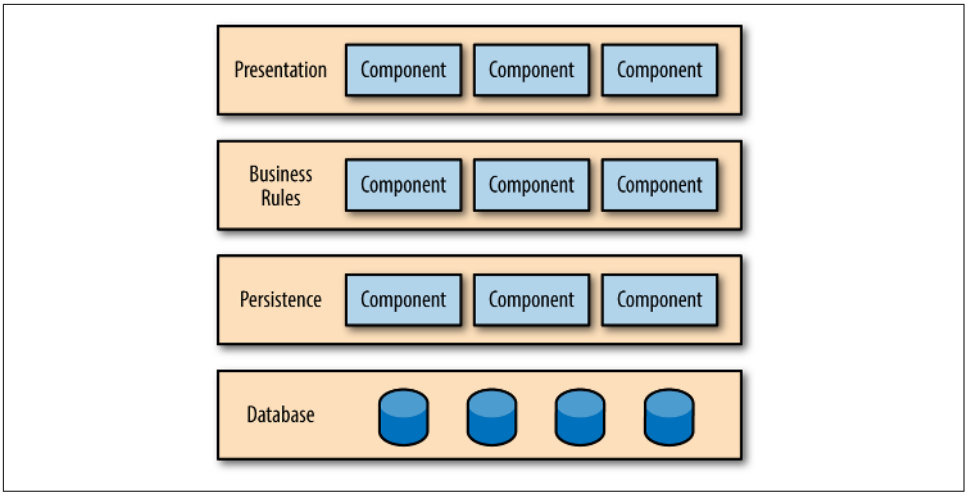



*Figure 4-5. Typical layered monolith architecture*

In Figure 4-5, each layer represents a technical capability, allowing developers to swap out technical architecture functionality easily. The primary design criteria for the layered architecture separates different technical capabilities into layers, each with a distinct responsibility. The primary advantages of this architecture are *isolation* and *separation of concerns*. Each layer is isolated from the others, accessible via a well-defined interface. This allows for implementation changes within the layer without affecting the other layers and grouping of similar code together, making space for specialization and separation within the layer. For example, a persistence layer typically encapsulates all implementation details of how data is saved, allowing other layers to ignore those details.

In all cases of monolith architecture, the quantum is the application, including dependent components like database servers. Evolving systems with large quantum size is difficult:

*Incremental change*

Developers find it easy to make some changes in this architecture, particularly if those changes are isolated to existing layers. Cross-layer changes can cause coordination challenges, especially if the organization's workforce resembles the layers of the architecture (a reflection of "Conway's Law" on page 11). For example, a team can swap one persistence framework for another with little disruption to other teams because they can perform that work behind the well-defined interface. If, on the other hand, the business is required to change something like `ShipToCustomer`, that change will affect all the layers, requiring coordination.

*Guided change with fitness functions*
: Developers find it easier to write fitness functions in a more structured version of a monolith because the structure of the architecture is more apparent. The separation of concerns in layers also allows developers to test more parts in isolation, making it easier to create fitness functions.

*Appropriate coupling*
: One of the virtues of monolith architectures is easy understandability. Developers who understand concepts such as design patterns can easily apply that knowledge to layered architectures. A large portion of understandability is convenient access to all parts of the code. Layered architectures allow for easy evolution of the technical architecture partitions defined by the layers. For example, a well-designed (and implemented) layered architecture makes it easy to swap out the database, business rules, or any other layer with minimal side effects.

Monolith architectures tend to have high coupling, both intentional and unintentional. When developers use layered architectures for separation of concerns (e.g., using a persistence layer to simplify data access), the layer typically exhibits high internal and low external coupling. Within the layer, each component is cooperating towards a single goal, so they tend toward high coupling. In contrast, developers typically define the interfaces between layers more carefully, creating lower coupling between layers.

### Modular monoliths

Many of the benefits architects tout about microservices—isolation, independence, small unit of change—can be achieved in monolithic architectures…if developers are extremely disciplined about coupling. Note that this discipline must extend beyond just technical architecture, encompassing other dimensions (notably data) equally. Modern tools make code reuse so convenient that developers struggle to achieve appropriate coupling in environments where coupling is easy. Fitness functions like the one in Example 4-1 allow architects to build safety nets into their deployment pipelines to keep monolith component dependencies clean.

Most modern languages allow building strict visibility and connection rules. If architects and developers build a modular monolith using those rules, they will have a much more malleable architecture, as demonstrated by a well modularized monolith depicted in Figure 4-6.

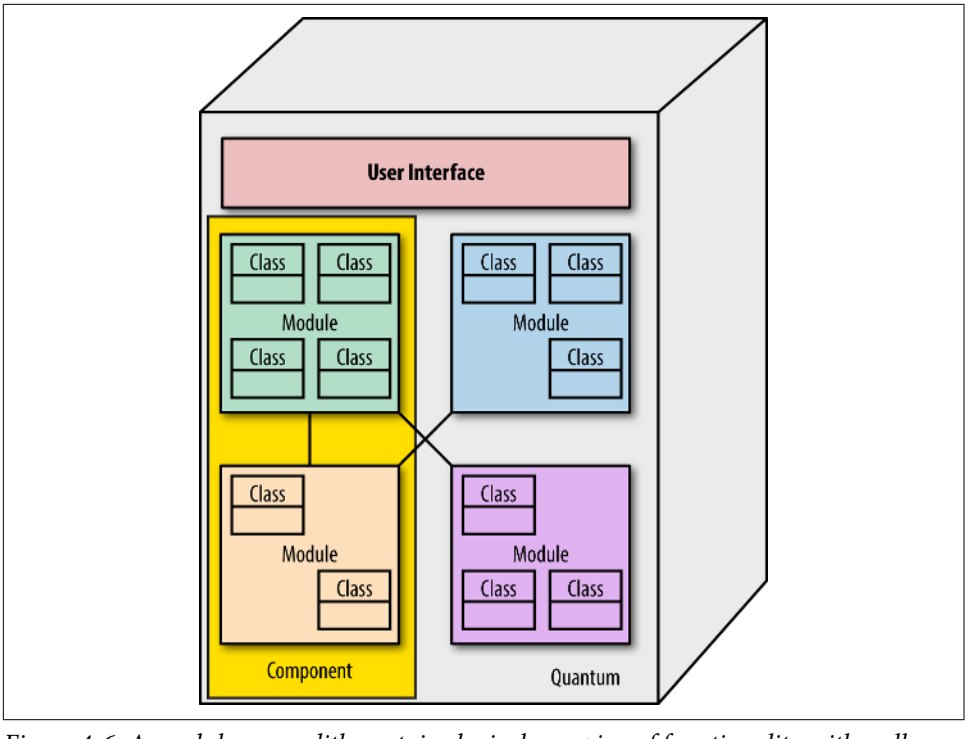



*Figure 4-6. A modular monolith contains logical grouping of functionality with well-defined isolation between modules*

*Incremental change*
: Incremental change is easy in this type of architecture because developers can enforce modularity. However, despite the *logical* separation of functionality into modules, if the components containing the modules are difficult to individually deploy, the quantum size is still large. In a modular monolith, the degree of deployability of the components determines the rate of incremental change.

*Guided change with fitness functions*
: Tests, metrics, and other fitness function mechanisms are easier to design and implement in this architecture because of good separation of components, allowing easier mocking and other testing techniques that rely on isolation layers.

*Appropriate coupling*
: A well-designed modular monolith is a good example of appropriate coupling. Each component is functionally cohesive, with good interfaces between them and low coupling.

Monolithic architectures, particularly layered architectures, are a common choice when starting a project because developers understand the structure easily. However,

many monoliths reach end of life and must be replaced because of decreasing performance, size of code base, and a host of other factors. A current common target for monolith migration is microservices-style architectures, which are more complex than monolithic architectures in areas like service and data granularity, operationalization, coordination, transactions, and so on. If a development team has a hard time building one of the simplest architectures, how will moving to a more complex architecture solve their problems?

> If you can't build a monolith, what makes you think microservices are the answer?
>
> —Simon Brown

Before embarking on an expensive architecture restructuring exercise, architects may benefit from improved modularization of what's already present. If nothing else, it's an excellent starting point for the more serious restructuring that follows.

### Microkernel

Consider another popular monolithic architectural style, the microkernal architecture, commonly found in browsers and integrated development environments (IDEs), as shown in Figure 4-7.

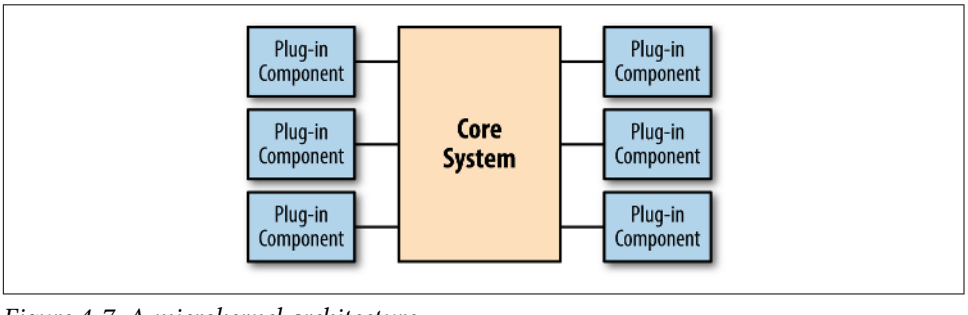



*Figure 4-7. A microkernel architecture*

The microkernel architecture shown in Figure 4-7 defines a core system with an API that allows plug-in enhancements. Two architectural quantum sizes exist for this architecture: one for the core system and another for the plug-ins. Architects typically design the core system as a monolith, creating hooks for well-known extension points for plug-ins. Plug-ins are usually designed to be self-contained and independently deployable. Thus, this architecture supports positive, incremental change, and developers can design for testability and facilitate fitness function definition. From a technical coupling standpoint, architects tend to design these systems with low coupling for practical reasons, which is related to keeping the plug-ins independent from one another to simplify them.

The primary challenge architects face in microkernel architectures revolves around contracts, a form of semantic coupling. To perform useful work, plug-ins must pass

information in and out of the core system. As long as the plug-ins don't need to coordinate between each other, developers can focus on information and versioning with the core system. For example, most browser plug-ins interact only with the browser, not other plug-ins.

More complex microkernel systems, such as the Eclipse Java IDE, must support intraplug-in communication. The core of Eclipse offers no specific language support beyond interacting with text files. All complex behavior comes via plug-ins that pass information between each other. For example, the compiler and debugger must closely coordinate during a debugging session. Because the plug-ins shouldn't rely on other plug-ins to work, the core system must handle the communication, making contract coordination and common tasks like versioning complex. While this level of isolation is desirable because it makes the system less stateful, it is often not possible. For example, in Eclipse, plug-ins often require dependent plug-ins to function, creating another level of transitive dependency management around the architectural quantum of the plug-in.

Typically, microkernel architectures include a registry that tracks installed plug-ins and the contracts they support. Building explicit coupling between plug-ins increases semantic coupling between parts of the system and therefore increases the architectural quantum.

While the microkernel architecture is popular with tools such as IDEs, it is also applicable for a wide variety of business applications. For example, consider an insurance company. The standard business rules for handling claims exist companywide, yet each state may have special rules. Building this system as a microkernel allows developers to add support for new states as needed and to upgrade individual state behavior without affecting any other state because of the inherent isolation of the plug-ins.

Microkernel architectures offer reasonably good if limited opportunities to evolve the technical architecture via plug-ins. Systems with completely isolated plug-ins make evolution easier because no coupling exists between plug-ins; plug-ins that must collaborate increase coupling and therefore hinder evolution. If you design a system with interacting plug-ins, you should also build fitness functions to protect the integration points, modeled after consumer-driven contracts. The core system in microkernel architectures is typically large but stable—most changes in this architecture should occur in plug-ins (otherwise, the architect may have poorly partitioned the application). Thus, incremental change is straightforward: deployment pipelines trigger change to plug-ins to validate changes.

Architects don't traditionally include data dependencies within the technical architecture for microkernels, so developers and DBAs must consider data evolution independently. Treating each plug-in as a bounded context improves the evolvablilty of the architecture because it decreases external coupling. For example, if all the plug-ins use the same database as the core system, developers must worry about coupling

occurring between plug-ins at the data level. If each plug-in is completely independent, this data coupling cannot occur.

From an evolutionary standpoint, microkernels have many desirable characteristics, including the following:

*Incremental change*
: Once the core system is complete, most behavior should come from plug-ins, small units of deployment. If the plug-ins are independent, incremental change becomes even easier.

*Guided change with fitness functions*
: Fitness functions are typically easy to create in this architecture because of the isolation between the core and plug-ins. Developers maintain two sets of fitness functions for systems like this: *core* and *plug-ins*. The core fitness functions guard against changes to the core, including deployment concerns like scalability. Generally, plug-in testing is simpler as the domain behavior is tested in isolation. Developers will want a good mock or stub version of the core to make testing plug-ins easier.

*Appropriate coupling*
: The coupling characteristics in this architecture are well defined by the microkernel pattern. Building independent plug-ins makes change trivial from a coupling standpoint. Dependent plug-ins make coordination more difficult. Developers should use fitness functions to ensure dependent components integrate properly.

These architectures should also include holistic fitness functions to ensure that developers maintain key architectural characteristics. For example, individual plug-ins might affect a systematic property like scalability. Thus, developers should plan to have a suite of integration tests to act as a holistic fitness function. In systems with dependent plug-ins, developers should also have a holistic fitness function to ensure contract and message consistency.

## Event-Driven Architectures

Event-driven architectures (EDA) usually integrate several disparate systems together using message queues. There are two common implementations of this type of architecture: the *broker* and *mediator* patterns. Each pattern has different core capabilities, thus we discuss the pattern and evolution implications separately.

### Brokers

In a *broker* EDA, the architectural components consist of the following elements:

*message queues*
: Message queues implemented via a wide variety of technologies such as JMS (Java Messaging Service).

*initiating event*
: The event that starts the business process.

*intra-process events*
: Events passed between event processors to fulfill a business process.

*event processors*
: The active architecture components, which perform actual business processing. When two processors need to coordinate, they pass messages via queues.

A typical broker EDA workflow is illustrated in Figure 4-8, in which a customer of an insurance company changes their address.

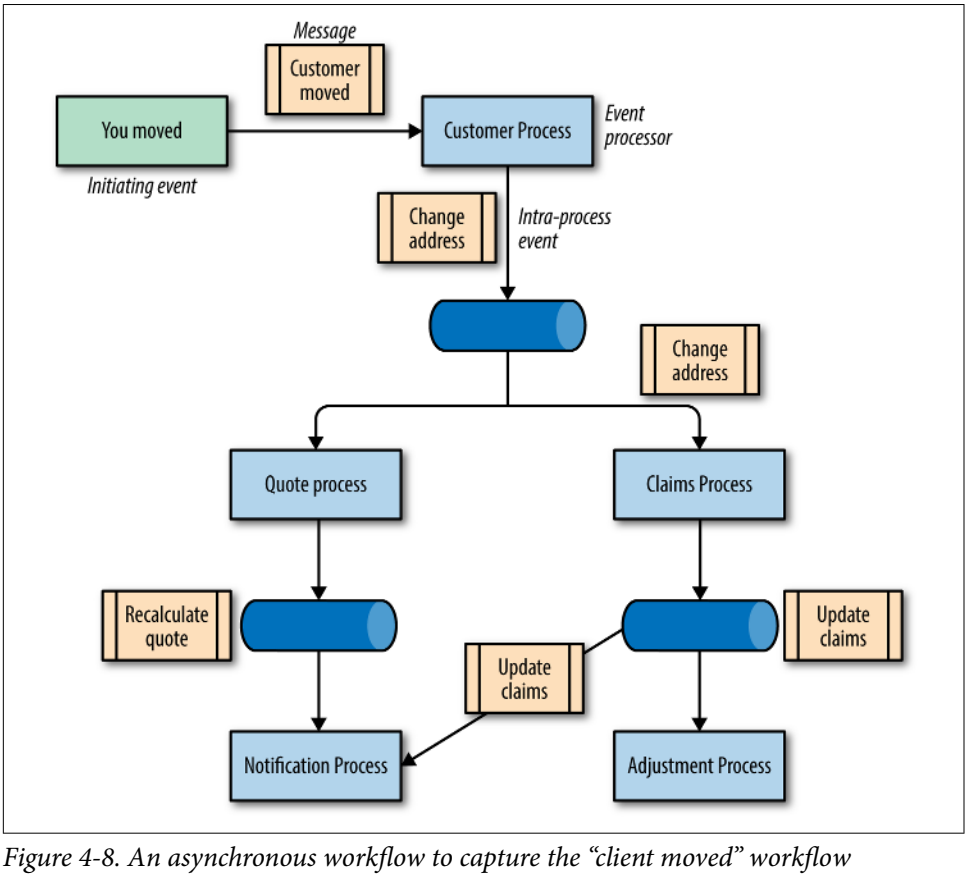



*Figure 4-8. An asynchronous workflow to capture the "client moved" workflow*

As seen in Figure 4-8, the initiating event is `client moved`. The first interested event processor is the `Customer process`, which updates the internal address record. Upon completion, it posts a message to a `address changed` message queue. Both the `Quote` and `Claims` processes respond to this event, updating their respective characteristics. Note that because the services need no coordination, these operations can occur in parallel, a key benefit of this architecture. Once completed, each processor posts to relevant queues, such as `Notification`.

Broker EDAs offer some design challenges when building robust asynchronous systems. For example, coordination and error handling are difficult because of a lack of a centralized mediator. Because the architectural parts are highly decoupled, developers must restore the functional cohesion of business processing to this architecture. Thus, behaviors such as transactions are more difficult.

Despite the implementation challenges, these are extremely evolvable architectures. Developers can add new behaviors to the system by adding new listeners on existing event queues, without affecting existing behavior. For example, let's say the insurance company wanted to add auditing to all claims updates. Developers can add an `Audit` listener on the `Claims` event queue without affecting the existing workflow.

*Incremental change*

Broker EDAs allow incremental change in multiple forms. Developers typically design services to be loosely coupled, making independent deployment easier. Decoupling in turn makes it easier for developers to make nonbreaking changes in the architecture. Building deployment pipelines for broker EDAs can be challenging because the essence of the architecture is asynchronous communication, which is notoriously difficult to test.

*Guided change with fitness functions*

Atomic fitness functions should be easy for developers to write in this architecture because the individual behaviors of event processors is simple. However, holistic fitness functions are both necessary and complex in this architecture. Much of the behavior of the overall system relies on the communication between loosely coupled services, making testing multifaceted workflows difficult. Consider the workflow in Figure 4-8. Developers can easily test the individual parts of the workflow by unit testing the event processors, but testing all processes is more challenging. There are a variety of ways to mitigate testing challenges in architectures like this. For example, correlation IDS, where each request is tagged with a unique identifier, helps track cross-service behavior. Similarly, synthetic transactions allow developers to test coordination logic without actually, for example, ordering washing machines.

*Appropriate coupling*

Broker EDAs exhibit a low degree of coupling, enhancing the ability to make evolutionary change. For example, to add new behavior to this architecture, new listeners are added to existing endpoints without affecting existing listeners. The coupling that does occur in this architecture is between services and the message contracts they maintain, a form of functional cohesion. Fitness functions using techniques like consumer-driven contracts help manage integration points and avoid breakages.

In business processes that lend themselves toward broker EDAs, the event processors are typically stateless, decoupled, and own their own data, making evolution easier because of fewer external coupling issues such as with databases, discussed in Chapter 5.

### Mediators

The other common EDA pattern is the *mediator*, where an additional component appears: a hub that acts as a coordinator, shown in Figure 4-9.

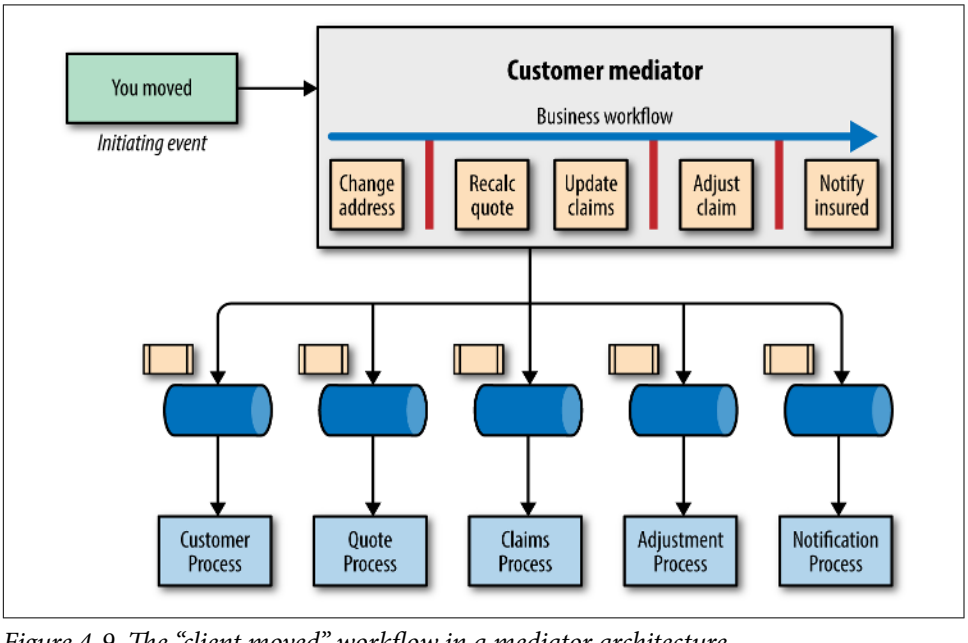



*Figure 4-9. The "client moved" workflow in a mediator architecture*

The mediator handles the initiating "client moved" event in Figure 4-9 and has the workflow defined within: `change address`, `recalc quotes`, `update claims`, `adjust claims`, and `notify insured`. The mediator posts messages to queues which in turn trigger the appropriate event processors. While the mediator handles coordination,

this is still an EDA, enabling most of the processing to occur in parallel. For example, the `recalc quotes` and `update claims` processes run in parallel. Once all the tasks are complete, the mediator generates a message to the `notify insured` queue to generate a single status message. Any event process that needs to communicate with another processor does so via the mediator. Generally, event processors do not call one another in this type of architecture because the mediator defines important workflow information direct communication would bypass. Notice the vertical bars depicted in the mediator in Figure 4-9, indicating both parallel execution and coordination of service requests and responses.

This transactional coordination is the primary advantage of the mediator architecture. The mediator can ensure errors don't occur during the process, and generate a single status message to the insured. In a broker EDA, this type of coordination is more difficult. To generate a single notification message, for example, the coordination would occur at the `Notification` event processor or via an explicit message queue to handle this aggregation. While asynchronous architectures create challenges around coordination and transactional behavior they offer fantastic parallel scale.

*Incremental change*
: Similar to broker EDAs, the services in a mediator EDA are typically small and self-contained. Thus, this architecture shares many of the operational advantages of the broker version.

*Guided change with fitness functions*
: Developers find it easier to build fitness functions for the mediator than for the broker EDA. The tests for individual event processors don't differ much from the broker version. However, holistic fitness functions are easier to build because developers can rely on the mediator to handle coordination. For example, in the insurance workflow, a developer can write a test and easily tell if the entire process was successful because the mediator coordinates it.

*Appropriate coupling*
: While many testing scenarios become easier with mediators, coupling increases, harming evolution. The mediator includes important domain logic, increasing the size of the architectural quantum to encompass it, which in turn couples each service to one another. In this architecture, when a developer makes a change, other developers must consider the side effects for the other services in the workflow, increasing coupling.

From an evolutionary standpoint, the broker architecture has clear advantages because of reduced coupling. In the mediator pattern, the coordinator acts as a coupling point, binding all the affected services together. In a broker topology, behavior can evolve by adding new processors to existing message queues without affecting the others (except in cases of overburdening the queue with traffic, which is solvable by a

variety of architectural patterns and/or fitness functions). Because broker topologies are inherently decoupled, evolution is easier.

This is a classic example of an architectural tradeoff. Broker EDAs offer many advantages in terms of evolvability, asynchronicity, scale, and a host of other desirable characteristics. However, common tasks like transactional coordination become more difficult.

## Service-Oriented Architectures

There are a variety of service-oriented architectures (SOAs) in existence, including many hybrids. Here are some common architectural patterns.

### ESB-driven SOA

A particular manner of creating SOAs became popular several years ago, building an architecture based around services and coordination via a *service bus*—typically called an *Enterprise Service Bus* (ESB). The service bus acts as a mediator for complex event interactions and handles various other typical integration architecture chores such as message transformation, choreography, and so on.

While ESB architectures typically use the same building blocks as EDAs, the organization of services differs, and is based on a strictly defined service taxonomy. The ESB style differs from organization to organization, but all are based on segregating services based on reusability, shared concepts, and scope. A representative ESB SOA is shown in Figure 4-10.

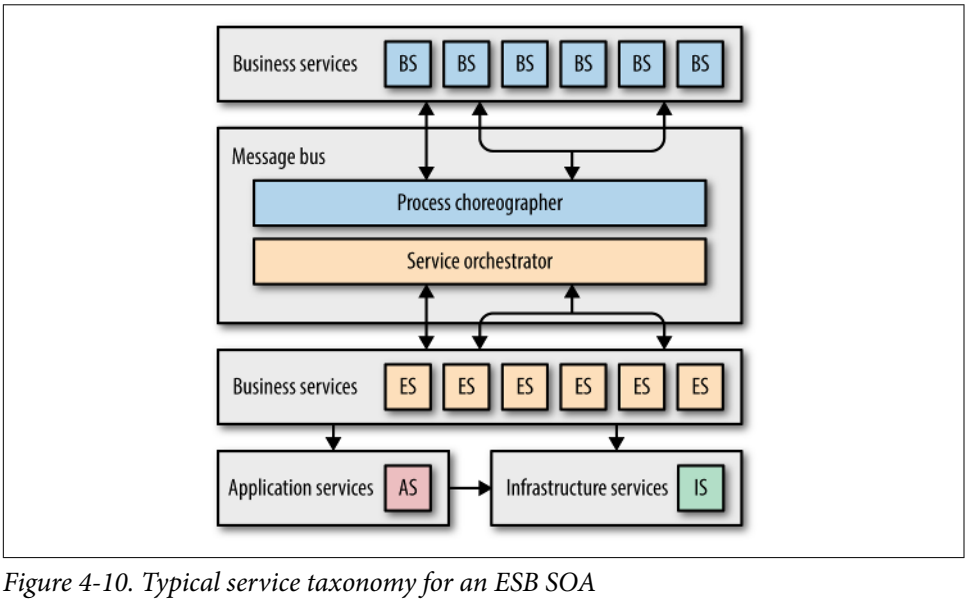



*Figure 4-10. Typical service taxonomy for an ESB SOA*

In Figure 4-10, each layer of the architecture has specific responsibilities. The business services define the coarse-grained functionality of the business as abstract definitions, sometimes defined by business users using standards like BPEL (Business Processing Execution Language). The goal of business services is to capture what the company does in an abstract way. To determine whether you have defined these services at the correct level of abstraction, ask yourself, "Are we in the business of…" affirmatively for each of the service names (like `CreateQuote` or `ExecuteTrade`). While a developer might invoke a `CreateCustomer` service in order to create a quote, that isn't the main thrust of the business but rather a necessary intermediate step.

The abstract business services must call code to implement their behavior, which are *enterprise services*: concrete implementations meant for sharing, owned by a distinct service teams. The goal of this team is to create reusable services integration architects can "stitch together" using choreography to form business implementations. Developers aim for high reuse and design enterprise services accordingly (to understand why this often fails, see "Antipattern: Code Reuse Abuse" on page 128).

Some services don't need a high degree of reuse. For example, one part of the system may need geolocation, but it isn't important enough to devote resources to make it a full blown enterprise service. The *Application Services* at the bottom left in Figure 4-10 handle these cases. These are bound to a specific application context and not meant for reuse, and are typically owned by a specific application team.

*Infrastructure services* are shared services owned by an infrastructure team to handle nonfunctional requirements such as monitoring, logging, authentication/authorization, and so on.

The defining characteristic of an ESB-driven SOA is the message bus architectural component, responsible for a wide variety of tasks as follows:

*Mediation and routing*
: The message bus knows how to locate and communicate with services. Typically, it maintains a registry of physical location, protocols, and other information needed to invoke services.

*Process choreography and orchestration*
: The message bus composes enterprise services together and manages tasks like invocation order.

*Message enhancement and transformation*
: One of the benefits of an integration hub is its ability to handle protocol and other transformations on behalf of applications. For example, `ServiceA` may "speak" `HTTP` and needs to call `ServiceB`, which only "speaks" `RMI/IIOP`. Developers can configure the message bus to handle this transformation invisibly anytime this conversion is needed.

The architectural quantum for ESB-driven SOA is massive! It basically encompasses the entire system, much like a monolith, but is much more complex because it is a distributed architecture. Making singular evolutionary change is extraordinarily difficult in ESB-driven SOA because the taxonomy, while assisting reuse, harms common change. For example, consider the `CatalogCheckout` domain concept within an SOA—it is smeared throughout the technical architecture. Making a change to only `CatalogCheckout` requires coordination between the parts of the architecture, commonly owned by different teams, generating a tremendous amount of coordination friction.

Contrast this representation of `CatalogCheckout` with the bounded context partitioning of microservices. In a microservices architecture, each bounded context represents a business process or workflow. Thus, developers would build a bounded context around something like `CatalogCheckout`. It is likely that `CatalogCheckout` will need details about `Customer`, but each bounded context "owns" their own entities. If other bounded contexts also have the notion of `Customer`, developers make no attempt to unify around a single, shared `Customer` class, which would be the preferred approach in an ESB-driven SOA. If the `CatalogCheckout` and `ShipOrder` bounded contexts need to share information about their customers, they do so via messaging rather than trying to unify around a single representation.

ESB-driven SOA was never designed to exhibit evolutionary properties, so it's no surprise that none of the evolutionary facets score well here:

*Incremental change*
: While having a well-established technical service taxonomy allows for reuse and segregation of resources, it greatly hampers making the most common types of change to business domains. Most SOA teams are as partitioned as the architecture, requiring herculean amounts of coordination for common changes. ESB-driven SOA is notoriously difficult to operationalize as well. It typically consists of multiple physical deployment units, making coordination and automation challenging. No one chooses ESBs for agility and operational ease of use.

*Guided change with fitness functions*
: Testing in general is difficult within ESB-driven SOA. No one piece is complete—every piece is part of a larger workflow and isn't typically designed for isolated testing. For example, an enterprise service is designed for reuse, but testing its core behavior is challenging because it is only a portion of potentially a variety of workflows. Building atomic fitness functions is virtually impossible, leaving most verification chores to large-scale holistic fitness functions that do end-to-end testing.

*Appropriate coupling*

From a potential enterprise reuse standpoint, extravagant taxonomy makes sense. If developers can manage to capture the reusable essence of each workflow, they will eventually write all the company's behavior once and for all, and future application development consists of connecting existing services. However, in the real world this isn't always possible. ESB-driven SOA isn't built to allow independent evolvable parts, so it has extremely poor support for it. Designing for categorical reuse harms the ability to make evolutionary change at the architectural level.

Software architectures aren't created in a vacuum—they always reflect the ecosystem in which they were defined. For example, when SOA was a popular architectural style, companies didn't use tools like open-source operating systems—all infrastructure was commercial, licensed, and expensive. A decade ago, a developer proposing a microservices architecture, where every service runs on its own instance of an operating system and machine, would be laughed out of the operations center because the architecture would have been ludicrously expensive. Because of the dynamic equilibrium of the software development ecosystem, new architectures arise because of a literal new environment.

While architects may still choose ESB-driven SOA for integration heavy environments, scale, taxonomy, or other legitimate reasons, they choose it for those features rather than evolvability, for which it is spectacularly unsuited.

### Microservices

Combining the engineering practices of Continuous Delivery with the logical partitioning of bounded context forms the philosophical basis for the microservice style of architecture, along with our architectural quantum concept.

In a layered architecture, the focus is on the *technical* dimension, or how the mechanics of the application work: persistence, UI, business rules, etc. Most software architectures focus primarily on these technical dimensions. However, an additional perspective exists. Suppose that one of the key bounded contexts in an application is *Checkout*. Where does it live in the layered architecture? Domain concepts like *Checkout* smear across the layers in this architecture. Because the architecture is segregated via technical layers, there is no clear concept of the *domain* dimension in this architecture, as can be seen in Figure 4-11.

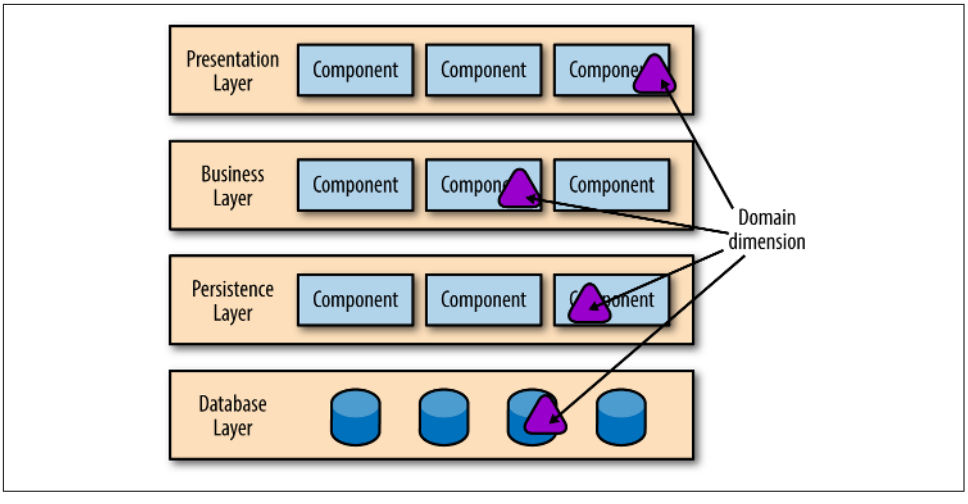



*Figure 4-11. The domain dimension is embedded within technical architecture*

In Figure 4-11, some portion of *Checkout* exists in the UI, another portion lives in the business rules, and persistence is handled by the bottom layers. Because layered architecture isn't designed to accomodate domain concepts, developers must modify each layer to make changes to domains. From a domain perspective, a layered architecture has zero evolvability. In highly coupled architectures, change is difficult because coupling between the parts developers want to change is high. Yet, in most projects, the common unit of change revolves around domain concepts. If a software development team is organized into silos resembling their role in the layered architecture, then changes to *Checkout* require coordination across many teams.

In contrast, consider an architecture where the *domain* dimension is the primary segregation of the architecture, as shown in Figure 4-12.

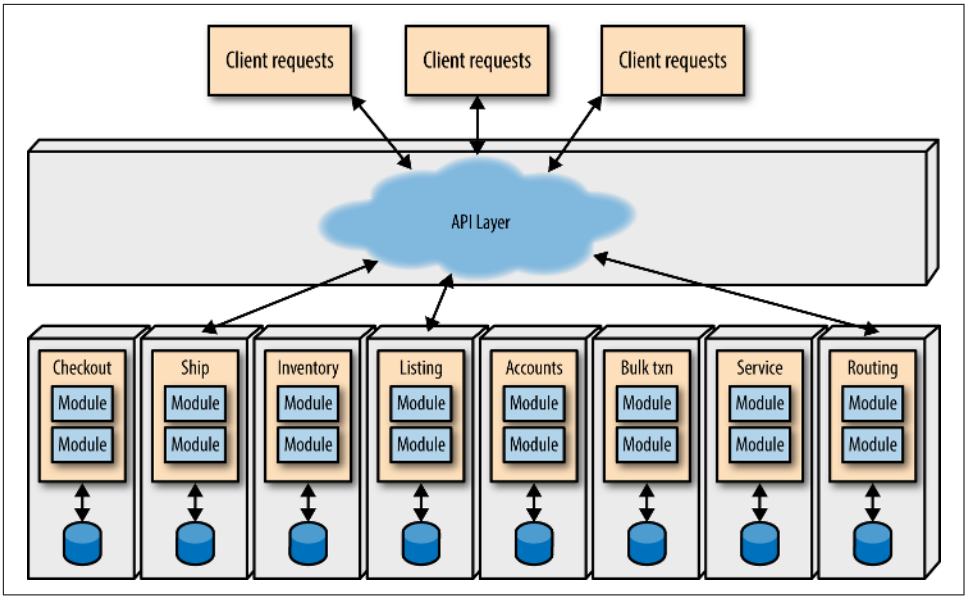



*Figure 4-12. Microservices architectures partition across domain lines, embedding the technical architecture*

As shown in Figure 4-12, each service is defined around DDD domain concept, encapsulating the technical architecture and all other dependent components (like databases) into a bounded context creating a highly decoupled architecture. Each service "owns" all parts of its bounded context, and communicates with other bounded contexts via messaging (such as REST or message queues). Thus, no service is allowed to know the implementation details of another service (such as database schemas), preventing inappropriate coupling. The operational goal of this architecture is to replace one service with another without disrupting other services.

Microservices architectures generally follow seven principles, as discussed in Building Microservices Architectures:

*Modeled around the business domain*
: The emphasis in microservices design is on the business domain, not technical architecture. Thus, the quantum reflects the bounded context. Some developers make the mistaken association that a bounded context represents a single entity such as `Customer`; instead, it represents a business context and/or workflow such as CatalogCheckout. The goal in microservices isn't to see how small developers can make each service but rather to create a useful bounded context.

*Hide implementation details*
: The technical architecture in microservices is encapsulated within the service boundary, which is based on the business domain. Each domain forms a physical

bounded context. Services integrate with each other by passing messages or resources, not by exposing details like database schemas.

*Culture of automation*
: Microservices architectures embrace Continuous Delivery, by using deployment pipelines to rigorously test code and automate tasks like machine provisioning and deployment. Automated testing in particular is extremely useful in fast-changing environments.

*Highly decentralized*
: Microservices form a *shared nothing* architecture—the goal is to decrease coupling as much as possible. Generally, duplication is preferable to coupling. For example, both the `CatalogCheckout` and `ShipToCustomer` services have a concept called `Item`. Because both teams have the same name and similar properties, developers try to reuse it across both services, thinking it will save time and effort. Instead, it increases effort because changes must now propagate between all the teams that share the component. And, whenever a service changes, developers must worry about changes to the shared component. If, on the other hand, each service has their own `Item` and passes information they need from `CatalogCheckout` to `ShipToCustomer` without coupling to the component, they can each change independently.

*Deployed independently*
: Developers and operations expect that each service component will be deployed independently from other services (and other infrastructure), reflecting the physical manifestation of the bounded context. The ability for developers to deploy one service without affecting any other service is one of the defining benefits of this architectural style. Moreover, developers typically automate all deployment and operations tasks, including parallel testing and Continuous Delivery.

*Isolate failure*
: Developers isolate failure both within the context of a microservices and in the coordination of services. Each service is expected to handle reasonable error scenarios and recover if possible. Many DevOps best practices (such as the circuit breaker pattern, bulkheads, and so on) commonly appear in these architectures. Many microservices architectures adhere to the Reactive Manifesto, a list of operational and coordination principles that lead to more robust systems.

*Highly observable*
: Developers cannot hope to manually monitor hundreds or thousands of services (how many multicast SSH terminal sessions can one developer observe?). Thus, monitoring and logging become first-class concerns in this architecture. If operations cannot monitor one of these services, it might as well not exist.

The main goals of microservices are isolation of domains via physical bounded context and emphasis on understanding the problem domain. Therefore, the architectural quantum is the service, making this an excellent example of an evolutionary architecture. If one service needs to evolve to change its database, no other service is affected because no other service is allowed to know implementation details like schemas. Of course, the developers of the changing service will have to deliver the same information via the integration point between the services (hopefully protected by a fitness function like *consumer-driven contracts*), allowing the calling service developers the bliss of never knowing the change occurred.

Given that microservices is our exemplar for an evolutionary architecture, it is unsurprising that it scores well from an evolutionary standpoint.

*Incremental change*
: Both aspects of incremental change are easy in microservices architectures. Each service forms a bounded context around a domain concept, making it easy to make changes that only affect that context. Microservices architectures rely heavily on automation practices from *Continuous Delivery*, utilizing deployment pipelines and modern DevOps practices.

*Guided change with fitness functions*
: Developers can easily build both atomic and holistic fitness functions for microservices architectures. Each service has a well-defined boundary, allowing a variety of levels of testing within the service components. Services must coordinate via integration, which also requires testing. Fortunately, sophisticated testing techniques grew alongside the development of microservices.

*Appropriate coupling*
: Microservices architectures typically have two kinds of coupling: *integration* and *service template*. Integration coupling is obvious—services need to call each other to pass information. The other type of coupling, service templates, prevents harmful duplication. Developers and operations benefit if a variety of facilities are consistent and managed within microservices. For example, each service needs to include monitoring, logging, authentication/authorization, and other "plumbing" capabilities. If left to the responsibility of each service team, ensuring compliance and lifecycle management like upgrades will likely suffer. By defining the appropriate technical architecture coupling points in service templates, an infrastructure team can manage that coupling while freeing individual service teams from worrying about it. Domain teams merely extend the template and write their behavior. When upgrades to infrastructure changes, the template picks it up automatically during the next deployment pipeline execution.

The physical bounded context in microservices correlates exactly to our concept of architectural quantum—it is a physically decoupled deployable component with high functional cohesion.

One of the key principles of the microservices style of architecture is strict partitioning across domain-bounded contexts. The technical architecture is embedded within the domain parts, honoring DDD's bounded context principle by making each service physically separate, which leads to a *share nothing* architecture from the technical perspective. Physical separation is expected for each service, allowing easy replacement and evolution. Because each microservice embeds the technical architecture within the bounded context, any service may evolve in any way necessary. Thus, the dimensions of evolvability for microservices corresponds to the number of services, each of which developers can treat independently because each service is highly decoupled.

### "Share Nothing" and Appropriate Coupling

Architects often call microservices a "share nothing" architecture. The primary advantage of this architecture style is no coupling at the technical architecture layer. But people who decry coupling are usually talking about "inappropriate coupling." After all, a software system with no coupling isn't very capable. "Share nothing" really means "no entangling coupling points." Even in microservices, some things need to be shared and coordinated, such as tools, frameworks, libraries, and so on. For instance, logging, monitoring, service discovery, etc. A service team forgetting to add monitoring capabilities to their service is a disaster at deployment time. In a microservices architecture, if a service can't be monitored, it disappears into a black hole.

*Service templates* (such as DropWizard and Spring Boot) are common solutions to this problem in microservices. These frameworks allow a DevOps team to build consistent tools, frameworks, versions, etc., into the service template. Service teams use the template to "snap in" their business behavior. When the monitoring tool updates, the service team can coordinate the update to the service template without bothering other teams.

If there are clear benefits, then why haven't developers embraced this style before? A decade ago, automatic provisioning of machines wasn't possible. Operating systems were commercial and licensed, with little support for automation. Real-world constraints like budgets impact architectures, which is one of the reasons developers build more and more elaborate shared resources architectures, segregated at the technical layers. If operations is expensive and cumbersome, architects build around it, as they did in ESB-SOAs.

The Continuous Delivery and DevOps movements added a new factor into the dynamic equilibrium. Now, machine definitions live in version control and support extreme automation. Deployment pipelines spin up multiple test environments in parallel to support safe continuous deployment. Because much of the software stack is open source, licensing and other concerns no longer impact architectures. The com-

munity reacted to the new capabilities emergent in the software development ecosystem to build more domain-centric architectural styles.

In microservices architecture, the domain encapsulates technical and other architectures, making evolution across domain dimensions easy. No one perspective on architecture is "correct," but rather a reflection on the goals developers build into their projects. If the focus is entirely on technical architecture, then making changes across that dimension is easier. However, if the domain perspective is ignored, then evolving across that dimension is no better than the Big Ball of Mud.

One of the major factors that impacts the ability to evolve an application at the architectural level is how *unintentionally coupled* each part of the system is. For example, in a layered architecture, architects specifically couple layers together in an intentional way. However, the domain dimension is unintentionally coupled, making evolution in that dimension difficult, because the architecture is designed around technical architecture layers, not the domain. Thus, one of the important aspects of an evolvable architecture is *appropriate coupling* across dimensions. We discuss how to identify and utilize quantum boundaries for practical purposes in Chapter 8.

#### Service-based architectures

A more commonly used architectural style for migration is a *service-based architecture*, which is similar to but differs from microservices in three important ways: service granularity, database scope, and integration middleware. Service-based architectures are still domain-centric but address some challenges developers face when restructuring existing applications toward more evolutionary architectures.

*Larger service granularity*
: The services in this architecture tend to be larger, more "portion of a monolith" granularity than purely around domain concepts. While they are still domain-centric, the larger size makes the unit of change (development, deployment, coupling, and a host of other factors) larger, diminishing the ability to make change easily. When architects evaluate a monolithic application, they often see coarse-grained divisions around common domain concepts such as CatalogCheckout or Shippping, which form a good first-pass at partitioning the architecture. The goals of operational isolation are the same in service-based architectures as in microservices but are more difficult to achieve. Because the service size is bigger, developers must consider more coupling points and the complications inherent in larger chunks of code. Ideally, the architecture should support the same kind of deployment pipeline and small unit of change as microservices: when a developer changes a service, it should trigger the deployment pipeline to rebuild the dependent services, including the application.

*Database scope*

Service-based architectures tend towards a monolithic database, regardless of how well-factored the services are. In many applications, it isn't feasible or possible to restructure years (or decades) of intractable database schemas into atomic-sized chunks for microservices. While the inability to atomize the data may be inconvenient in some situations, it is impossible in some problem domains. Heavily transactional systems are a poor match for microservices because coordination between services, transactional behavior is too costly. Systems with complex transactional requirements map more cleanly to service-based architectures because of less stringent database requirements.

While the database remains unpartitioned, the components that rely on the database will likely change, becoming more granular. Thus, while the mapping between the services and the underlying data may change, it requires less restructuring. We cover evolutionary database design in Chapter 5.

*Integration middleware*

The third difference between microservices and service-based architectures concerns externalized coordination via a mediator like a service bus. Building greenfield microservices applications allows developers to not worry about old integration points, but those horrors describe many environments rife with legacy systems that still perform useful work. Integration hubs, like enterprise service buses, excel at forming glue between disparate services with different protocols and message formats. If architects find themselves in environments where integration architecture is the top priority, using an integration hub makes adding and changing dependent services easier.

Using an integration hub is a classic architectural tradeoff: by using a hub, developers need to write less code to glue applications together, and may use it to mimic transaction coordination between services. However, using a hub increases the architectural coupling between components—developers can no longer make changes independently without coordinating with other teams. Fitness functions can mitigate some of this coordination cost, but the more developers increase coupling, the harder the system is to evolve.

Here is how a service-based architecture measures against our evolutionary architecture evaluation:

*Incremental change*

Incremental change is relatively functional in this architecture because each service is domain centric. Most changes in software projects occur around domains, providing alignment between unit of change and quantum of deployment. While not as agile as microservices because the service size tends to be larger, many of the advantages of microservices is preserved.

*Guided change with fitness functions*
: Developers typically find it more difficult to write fitness functions in service-based architectures than in microservices because of increased coupling (typically at the database) and a larger bounded context. Increased code coupling often makes writing tests more difficult, and increased data coupling creates its own host of problems. The larger bounded context of service-based architectures creates more opportunities for developers to create internal coupling points, complicating testing and other diagnostics.

*Appropriate coupling*
: Coupling is often the reason developers pursue a service-based architecture rather than microservices: difficulties deconstructing database schemas, high degree of coupling within a monolith targeted for restructuring, and so on. Creating domain-centric services helps ensure appropriate coupling, and service templates help create the appropriate level of technical architecture coupling.

Serverless BaaS architectures allow limited evolution but attractive operational characteristics. Service-based architectures are certainly more inherently evolvable than ESB SOA architectures. The degree to which developers have deviated from bounded context largely determines the quantum size and how much damaging coupling appears.

Service-based architectures are a good compromise between the philosophical purity of microservices and the pragmatic realities of many projects. By loosening the strictures on service size, database independence, and incidental but useful coupling, this architecture solves the most painful aspects of microservices while preserving many of the benefits.

## "Serverless" Architectures

"Serverless" architectures are a recent shift in the software development equilibrium, with two broad meanings, both applicable to evolutionary architecture.

Applications that significantly or primarily depend on third-party applications and/or services in "the cloud" are called BaaS (Backend as a Service). For example, consider the simplified example shown in Figure 4-13.

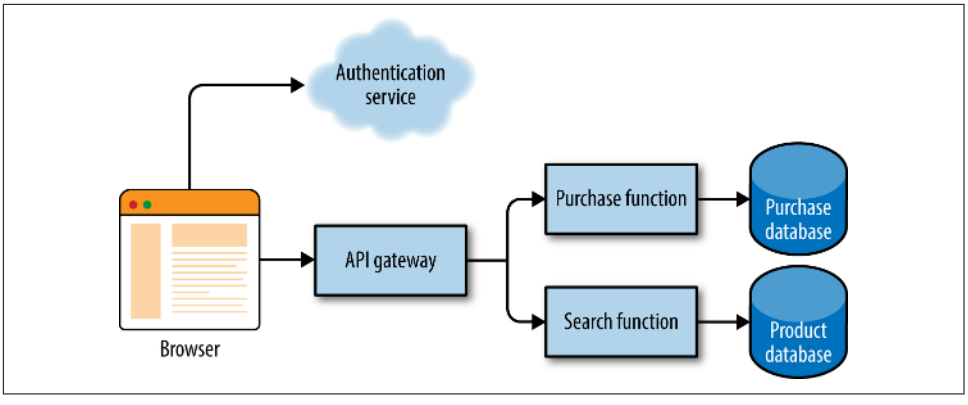



*Figure 4-13. Serverless BaaS*

In Figure 4-13, the developer writes little or no code. Instead, the architecture consists of wiring together services, including things like authentication, data transfer, and other integration architecture pieces. This type of architecture is appealing because, the fewer lines of code an organization writes, the fewer it must maintain. However, integration-heavy architectures come with their own challenges.

The other type of serverless architectures is FaaS (Function as a Service), which eschews infrastructure entirely (at least from the developer's standpoint), provisioning infrastructure per request, automatically handling scaling, provisioning, and a host of other management duties. Functions in FaaS are triggered by event types defined by the service provider. For example, Amaazon Web Services (AWS) is a common FaaS provider, supplying events triggered by file updates (on S3), time (scheduled tasks), and messages added to a message bus (e.g., Kinesis). Providers often limit the amount of time a request may take along with other restrictions, primarily around state. The general assumption is that FaaS functions are stateless, placing the burden of managing state on the caller.

*Incremental change*

Incremental change in serverless architectures should consist of redeploying code—all the infrastructure concerns exist behind the abstraction of "serverless." These types of architecture are natural fits for deployment pipelines, to handle testing and incremental deployment as developers make changes.

*Guided change via fitness functions*

Fitness functions are critical in this type of architecture to ensure integration points stay consistent. Because coordination between services is key, developers can expect to write a larger percentage of holistic fitness functions, which must run in the context of several integration points, to ensure third-party APIs haven't drifted. Architects frequently build anticorruption layers between integra-

tion points to avoid the Vendor King antipattern, discussed in "Antipattern: Vendor King" on page 123.

*Appropriate coupling*
: From an evolutionary architecture standpoint, FaaS is attractive because it eliminates several different dimensions from consideration: technical architecture, operational concerns, and security issues, among others. While this architecture may be easy to evolve, it suffers from serious constraints around practical considerations, offloading much of the complexity to the invoker. For example, while FaaS will handle elastic scalability, the caller must handle any transactional behavior and other complex coordination. In a traditional application, transactional coordination is typically handled by the back-end. However, if the BaaS doesn't support that behavior, coordination must move to the user interface (the invoker of the service).

Architects shouldn't choose an architecture without evaluating it against the real problems they must solve.

Make sure your architecture matches the problem domain. Don't try to force fit an unsuitable architecture.

While serverless architectures have many appealing features, they also have limitations. In particular, all-encompassing solutions often suffer from the "Antipattern: Last 10% Trap" as discussed on on page 127. Most of what a team needs to build is quick and easy, but other times, building a complete solution can be frustrating.

## Controlling Quantum Size

The quantum size of an architecture largely determines how easy it will be for developers to make evolutionary changes. Large quanta like monoliths and ESB SOA are difficult to evolve because of the coordination required for each change. More decoupled architectures like broker event-driven and microservices offer many more avenues for easy evolution.

The structural constraints on evolving architecture depend on how well developers have handled coupling and functional cohesion. Evolution is easier if developers have created a modular component system with well-defined integration points. For example, if developers build a monolith, but are diligent about good modularity and component isolation, that architecture will offer more opportunities to evolve because the size of the architectural quantum is smaller due to decoupling.

The smaller your architectural quanta, the more evolvable your architecture will be.

## Case Study: Guarding Against Component Cycles

PenultimateWidgets has several monolithic applications under active development. When designing components, one of the architect's goals is to create self-contained components—the more isolated the code, the easier it is to make changes. A common problem in many languages with powerful IDEs is the *package dependency cycle*, which describes the common scenario illustrated in Figure 4-14.

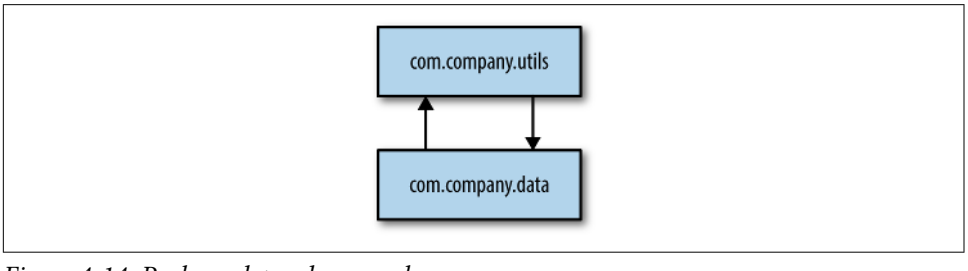



*Figure 4-14. Package dependency cycle*

In Figure 4-14, the package `com.company.data` imports from `com.company.utils`, and `com.company.utils` imports from `com.company.data`—neither component can be used without dragging the other along, creating a component cycle. Obviously, cycles hurt changeability because an elaborate network of cycles makes incremental change difficult. Languages like Java or C# have development environments that assist developers (via code insight helpers built into the IDE) by suggesting missing imports. Because developers implicitly import so many things in the course of daily coding with the help of the IDE, preventing package dependency cycles is difficult because the tooling fights against this effort.

The PenultimateWidgets architects on these systems worry about developers accidentally introducing cycles between components. Fortunately, they have a mechanism to help guard against factors that harm the evolvability of applications—fitness functions. Rather than abandon the benefits of IDEs because they encourage bad habits, an engineering safety net via fitness functions can be built instead. Both commercial and open source tools exist for many popular platforms to help untangle cycles. Many take the form of a static code analysis tool that looks for cycles, while others provide "to-do" lists of refactorings to assist developers in fixing them.

After the cycles have been removed, how can you prevent a developers's idle habits from introducing new ones? Coding standards don't help for this type of problem

because developers have a hard time remembering bureaucratic policies in the heat of coding. Instead, they prefer to establish tests and other verification mechanisms to guard against too-helpful tools.

The PenultimateWidgets developers use a popular open source tool for the Java platform called JDepend, which includes both textual and graphical interfaces to help analyze dependencies. Because JDepend is written in Java, developers can utilize its API to write structural tests of their own. Consider the test case in Example 4-1.

*Example 4-1. Using JDepend to identify cycles programmatically*

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class CycleTest extends TestCase {
    private JDepend jdepend;

    protected void setUp() throws IOException {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/util/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    /**
     * Tests that a single package does not contain
     * any package dependency cycles.
     */
    public void testOnePackage() {
        jdepend.analyze();
        JavaPackage p = jdepend.getPackage("com.xyz.thirdpartyjars");
        assertEquals("Cycle exists: " + p.getName(),
               false, p.containsCycle());
    }

    /**
     * Tests that a package dependency cycle does not
     * exist for any of the analyzed packages.
     */
    public void testAllPackages() {
        Collection packages = jdepend.analyze();
        assertEquals("Cycles exist",
                    false, jdepend.containsCycles());
    }
}
```

In Example 4-1, the developer adds the directories containing packages to `jdepend`. Then, the developer can test either a single package for cycles or the entire codebase, as shown in the unit test `testAllPackages()`. Once the project has gone through the

laborious task of identifying and removing cycles, put the `testAllPackages()` unit test in place as an application architecture fitness function to guard against future cycle occurrence.

CHAPTER 5

# Evolutionary Data

*with contributions from Pramod Sadalage*

Relational and other types of data stores are ubiquitous in modern software projects, a form of coupling that is often more problematic than architectural coupling. *Data* is an important dimension to consider when creating an evolvable architecture. It is beyond the scope of this book to cover all the aspects of evolutionary database design. Fortunately, our colleage Pramod Sadalage, along with Scott Ambler, wrote *Refactoring Databases*, subtitled *Evolutionary Database Design*. We cover only the parts of database design that impact evolutionary architecture and encourage readers to read this book.

When we refer to the *DBA*, we mean anyone who designs the data structures, writes code to access the data and use the data in an application, writes code that executes in the database, maintains and performance tunes the databases, and ensures proper backup and recovery procedures in the event of disaster. DBAs and developers are often the core builders of applications, and should coordinate closely.

## Evolutionary Database Design

Evolutionary design in databases occurs when developers can build and evolve the structure of the database as requirements change over time. Database schemas are abstractions, similar to class hierarchies. As the underlying real world changes, those changes must be reflected in the abstractions developers and DBAs build. Otherwise, the abstractions gradually fall out of synchronization with the real world.

### Evolving Schemas

How can architects build systems that support evolution but still use traditional tools like relational databases? The key to evolving database design lies in evolving schemas alongside code. Continuous Delivery addresses the problem of how to fit the tradi-

tional data silo into the continuous feedback loop of modern software projects. Developers must treat changes to database structure the same way they treat source code: tested, versioned, and incremental.

*Tested*
: DBAs and developers should rigorously test changes to database schemas to ensure stability. If developers use a data mapping tool like an object-relational mapper (ORM), they should consider adding fitness functions to ensure the mappings stay in sync with the schemas.

*Versioned*
: Developers and DBAs should version database schemas alongside the code that utilizes it. Source code and database schemas are symbiotic—neither functions without the other. Engineering practices that artificially separate these two necessarily coupled things cause needless inefficiencies.

*Incremental*
: Changes to the database schemas should accrue just as source code changes build up: incrementally as the system evolves. Modern engineering practices eschew manual updates of database schemas, preferring automated migration tools instead.

Database migration tools are utilities that allow developers (or DBAs) to make small, incremental changes to a database that are automatically applied as part of a deployment pipeline. They exist along a wide spectrum of capabilities from simple command-line tools to sophisticated proto-IDEs. When developers need to make a change to a schema, they write small delta scripts, as illustrated in Example 5-1.

*Example 5-1. A simple database migration*

```
CREATE TABLE customer (
       id BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1) PRIMARY KEY,
       firstname VARCHAR(60),
       lastname VARCHAR(60)
);
```

The migration tool takes the SQL snippet shown in Example 5-1 and automatically applies it to the developer's instance of the database. If the developer later realizes they forgot to add date of birth rather than change the original migration, they can create a new one that modifies the original structure, as shown in Example 5-2.

*Example 5-2. Adding date of birth to existing table using a migration*

```
ALTER TABLE customer ADD COLUMN dateofbirth DATETIME;

--//@UNDO
```

```
ALTER TABLE customer DROP COLUMN dateofbirth;
```

In Example 5-2, the developer modifies the existing schema to add a new column. Some migration tools support *undo* capabilities as well. Supporting *undo* allows developers to easily move forward and backward through the schema versions. For example, suppose a project is on version 101 in the source code repository and needs to return to version 95. For the source code, developers merely check out version 95 from version control. But how can they ensure the database schema is correct for version 95 of the code? If they use migrations with *undo* capabilities, they can "undo" their way backwards to version 95 of the schema, applying each migration in turn to regress back to the desired version.

However, most teams have moved away from building *undo* capabilities for three reasons. First, if all the migrations exist, developers can build the database just up to the point they need without backing up to a previous version. In our example, developers would build from 1 to 95 to restore version 95. Second, why maintain two versions of correctness, both forward and backward? To confidently support *undo*, developers must test the code, sometimes doubling the testing burden. Third, building comprehensive *undo* sometimes presents daunting challenges. For example, imagine that the migration dropped a table—how would the migration script preserve all data in the case of an *undo* operation?

Once developers have run migrations, they are considered immutable—changes are modeled after double-entry bookkeeping. For example, suppose that Danielle the developer ran the migration in Example 5-2 as the 24th migration on the project. Later, she realizes `dateofbirth` isn't needed after all. She could just remove the 24th migration, and the end result on the table is no column. However, any code written between the time Danielle ran the migration and now assumes the presence of the `dateofbirth` column, and will no longer work if for some reason the project needs to back up to an intermediate point (e.g., to fix a bug). Instead, to remove the no-longer needed column, she runs a new migration that removes the column.

Database migrations allow both database admins and developers to manage changes to schema and code incrementally, by treating each as parts of a whole. By incorporating database changes into the deployment pipeline feedback loop, developers have more opportunities to incorporate automation and earlier verification into the project's build cadence.

## Shared Database Integration

A common integration pattern highlighted here is Shared Database Integration, which uses a relational database as a sharing mechanism for data, as illustrated in Figure 5-1.

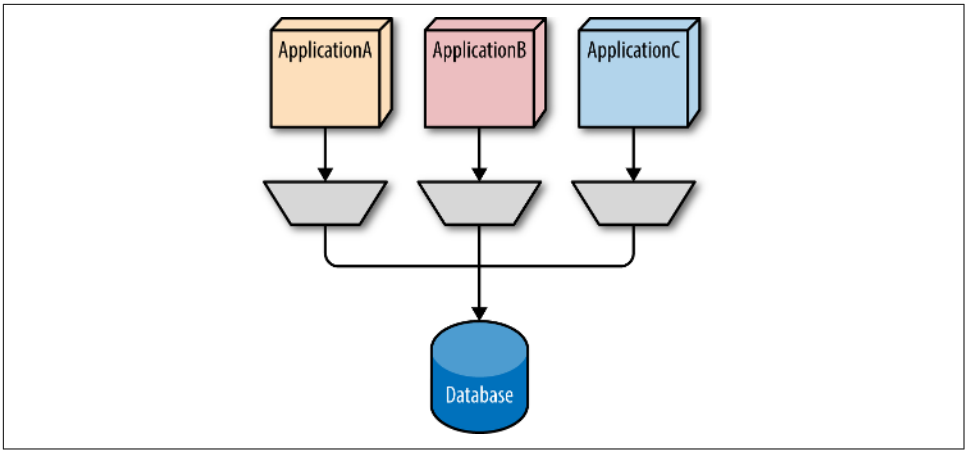



*Figure 5-1. Using the database as an integration point*

In Figure 5-1, each of the three applications share the same relational database. Projects frequently default to this integration style—every project is using the same relational database because of governance, so why not share data across projects? Architects quickly discover, however, that using the database as an integration point fossilizes the database schema across all sharing projects.

What happens when one of the coupled applications needs to evolve capabilities via a schema change? If ApplicationA makes changes to the schema, this could potentially break the other two applications. Fortunately, as discussed in the aforementioned *Refactoring Databases* book, a commonly utilized refactoring pattern is used to untangle this kind of coupling called the *expand/contract pattern*. Many database refactoring techniques avoid timing problems by building a transition phase into the refactoring, as illustrated in Figure 5-2.

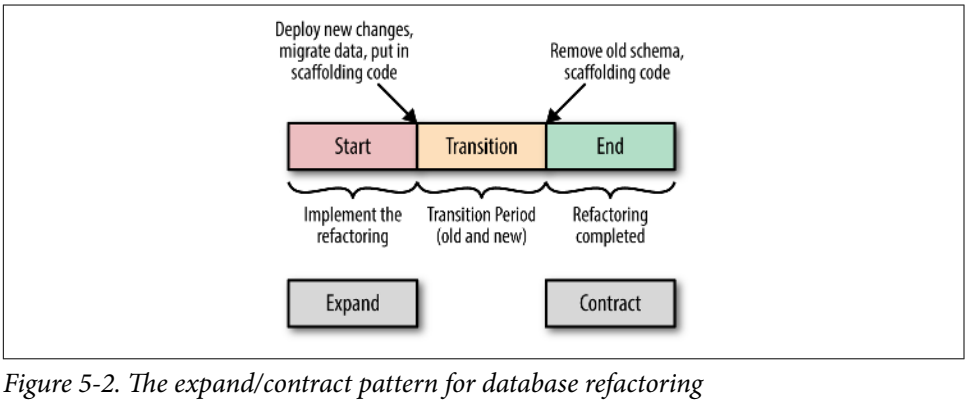



*Figure 5-2. The expand/contract pattern for database refactoring*

Using this pattern, developers have a starting state and an end state, maintaining both the *old* and *new* states during the transition. This transition state allows for backwards compatibility and also gives other systems in the enterprise enough time to catch up with the change. For some organizations, the transition state can last from a few days to months.

Here is an example of *expand/contract* in action. Consider the common evolutionary change of splitting a `name` column into `firstname` and `lastname`, which PenultimateWidgets needs to do for marketing purposes. For this change, developers have the start state, the expand state, and the final state, as shown in Figure 5-3.

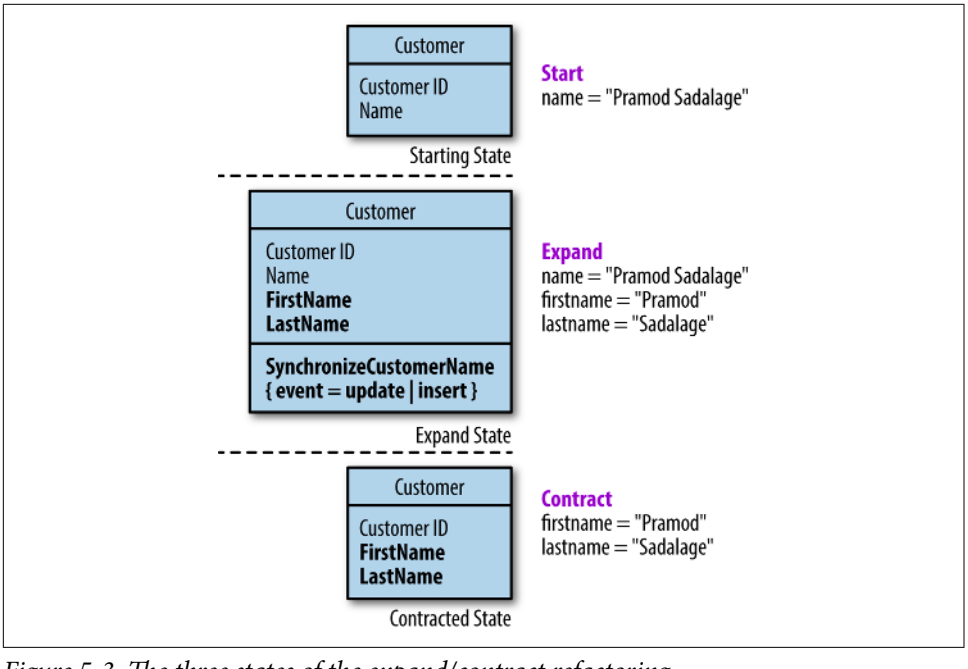



*Figure 5-3. The three states of the expand/contract refactoring*

In Figure 5-3, the full name appears as a single column. During the transition, PenultimateWidgets DBAs must maintain both versions to prevent breaking possible integration points in the database. They have several options on how we proceed to split the `name` column into `firstname` and `lastname`.

### Option 1: No integration points, no legacy data

In this case, the developers have no other systems to think about and no existing data to manage, so they can add the new columns and drop the old column, as shown in Example 5-3.

*Example 5-3. The simple case with no integration points and no legacy data*

```
ALTER TABLE customer ADD firstname VARCHAR2(60);
ALTER TABLE customer ADD lastname VARCHAR2(60);
ALTER TABLE customer DROP COLUMN name;
```

For *Option 1*, the refactoring is straightforward—DBAs make the relevant change and get on with life.

#### Option 2: Legacy data, but no integration points

In this scenario, developers assume existing data to migrate to new columns but they have no external systems to worry about. They must create a function to extract the pertinent information from the existing column to handle migrating the data, as shown in Example 5-4.

*Example 5-4. Legacy data but no integrators*

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);
UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);
ALTER TABLE customer DROP COLUMN name;
```

This scenario requires DBAs to extract and migrate the existing data but is otherwise straightforward.

#### Option 3: Existing data and integration points

This is the most complex and, unfortunately, most common scenario. Companies need to migrate existing data to new columns while external systems depend on the `name` column, which their developers cannot migrate to use the new columns in the desired timeframe. The required SQL appears in Example 5-5.

*Example 5-5. Complex case with legacy data and integrators*

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);

UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);

CREATE OR REPLACE TRIGGER SynchronizeName
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN
```

```
  IF :NEW.Name IS NULL THEN
    :NEW.Name := :NEW.firstname||' '||:NEW.lastname;
  END IF;
  IF :NEW.name IS NOT NULL THEN
    :NEW.firstname := extractfirstname(:NEW.name);
    :NEW.lastname := extractlastname(:NEW.name);
  END IF;
END;
```

To build the transition phase in Example 5-5, DBAs add a trigger in the database that moves data from the old `name` column to the new `firstname` and `lastname` columns when the other systems are inserting data into the database, allowing the new system to access the same data. Similarly, developers or DBAs concatenate the `firstname` and `lastname` into a `name` column when the new system inserts data so that the other systems have access to their properly formatted data.

Once the other systems modify their access to use the new structure (with separate first and last names), the *contraction* phase can be executed and the old column dropped:

```
ALTER TABLE Customer DROP COLUMN name;
```

If a lot of data exists and dropping the column will be time consuming, DBAs can sometimes set the column to "not used" (if the database supports this feature):

```
ALTER TABLE Customer SET UNUSED name;
```

After dropping the legacy column, if a read-only version of the previous schema is needed, DBAs can add a functional column so that read access to the database is preserved.

```
ALTER TABLE CUSTOMER ADD (name AS
            (generatename (firstname,lastname)));
```

As illustrated in each scenario, DBAs and developers can utilize the native facilities of databases to build evolvable systems.

Expand/contract is a subset of a pattern called parallel change, a broad pattern used to safely implement backward-incompatible changes to an interface.

# Inappropriate Data Coupling

Data and databases form an integral part of most modern software architectures—developers who ignore this key aspect when trying to evolve their architecture suffer.

Databases and DBAs form a particular challenge in many organizations because, for whatever reason, their tools and engineer practices are antiquated compared to the traditional development world. For example, the tools DBAs use on a daily basis are extremely primitive compared to any developer's IDE. Features that are common for

developers don't exist for DBAs: refactoring support, out-of-container testing, unit testing, mocking and stubbing, and so on.

### DBAs, Vendors, and Tool Choices

Why has the data world lagged so far behind the engineering practices of the software development world? DBAs have many of the same needs as developers: testing, refactoring, and so on. Yet, while developer tools continue to advance, the same level of innovation hasn't penetrated the data world. It's not like tools aren't available—several third-party tools now exist to add better engineering support—but they don't sell well. Why?

Database vendors have created an interesting relationship between themselves and their consumers. For example, a DBA for *DatabaseVendorX* has an almost irrational level of dedication to that vendor, because the DBA's next job comes at least in part from the fact they are a certified *DatabaseVendorX* DBA, not necessarily from their existing job. Thus, database vendors have secreted armies within enterprises all over the world, where loyalties lie with the vendor rather than the company. DBAs in this situation ignore tools and other development artifacts that don't come from the mothership. The result is stagnation at the innovation level for engineering practices.

DBAs view their database vendors as the source of all heat and light in the universe, and don't care what comes from other dark matter in their universe. The unfortunate side effect of this phenomenon is stagnation in tool advancement compared to developer tools. Consequently, the impedance mismatch between developers and DBAs grows even bigger, as they don't share common engineering techniques. Convincing DBAs to adopt Continuous Delivery practices forces them to use new tools, distancing them from the mothership, which they try to avoid.

Fortunately, the popularity of open source and NoSQL databases has started breaking the hegemony of database vendors.

## Two-Phase Commit Transactions

When architects discuss coupling, the conversation usually revolves around classes, libraries, and other aspects of the technical architecture. However, other avenues of coupling exist in most projects, including transactions.

Transactions are a special form of coupling because transactional behavior doesn't appear in traditional technical architecture-centric tools. Architects can easily determine the afferent and efferent coupling between classes with a variety of tools. They have a much harder time determining the extent of transactional contexts. Just as coupling between schemas harms evolution, transactional coupling binds the constituent parts together in concrete ways, making evolution more difficult.

Transactions appear in business systems for a variety of reasons. First, business analysts love the idea of transactions—an operation that *stops the world* for some context briefly—regardless of the technical challenges. Global coordination in complex systems is difficult, and transactions represent a form of it. Second, transactional boundaries often tell how business concepts are really coupled together in their implementation. Third, DBAs may own the transactional contexts, making it hard to coordinate breaking the data apart to resemble the coupling found in the technical architecture.

Developers encounter transactions as coupling points when attempting to translate heavily transactional systems to inappropriate architectural patterns like microservices, which impose heavy decoupling burdens. Service-based architectures, with much less strict service boundary and data partitioning requirements, fit transactional systems better. We discuss pertinent differences between these architectural styles in "Service-Oriented Architectures" on page 65.

In Chapters 1 and 4, we discussed the architectural quantum boundary concept definition: the smallest architectural deployable unit, which differs from traditional thinking about cohesion by encompassing dependent components like databases. The binding created by databases is more imposing than traditional coupling because of transactional boundaries, which often define how business processes work. Architects sometimes err in trying to build an architecture with a smaller level of granularity than is natural for the business. For example, microservices architectures aren't particularly well suited for heavily transactional systems because the goal service quantum is so small. Service-based architectures tend to work better because of less strict quantum size requirements.

Architects must consider all the coupling characteristics of their application: classes, package/namespace, library and framework, data schemas, and transactional contexts. Ignoring any of these dimensions (or their interactions) creates problems when trying to evolve an architecture. In physics, the *strong nuclear force* that binds atoms together is one of the strongest forces yet identified. Transactional contexts act like a strong nuclear force for architecture quanta.

Database transactions act as a strong nuclear force, binding quanta together.

While systems often cannot avoid transactions, architects should try to limit transactional contexts as much as possible because they form a tight coupling knot, hampering the ability to change components or services without affecting others. More

importantly, architects should take aspects like transactional boundaries into account when thinking about architectural changes.

As discussed in Chapter 8, when migrating a monolithic architectural style to a more granular one, start with a small number of larger services first. When building a greenfield microservices architecture, developers should be diligent about restricting the size of service and data contexts. However, don't take the name *microservices* too literally—each service doesn't have to be small, but rather capture a useful bounded context.

When restructuring an existing database schema, it is often difficult to achieve appropriate granularity. Many enterprise DBAs spend decades stitching a database schema together and have no interest in performing the reverse operation. Often, the necessary transactional contexts to support the business define the smallest granularity developers can make into services. While architects may aspire to create a smaller level of granularity, their efforts slip into inappropriate coupling if it creates a mismatch with data concerns. Building an architecture that structurally conflicts with the problem developers are trying to solve represents a damaging version of meta-work, described in "Migrating Architectures" on page 100.

## Age and Quality of Data

Another dysfunction that manifests in large companies is the fetishization of data and databases. We have heard more than one CTO say, "I don't really care that much about applications because they have a short lifespan, *but my data schemas are precious because they live forever!*" While it's true that schemas change less frequently than code, database schemas still represent an abstraction of the real world. While inconvenient, the real world has a habit of changing over time. DBAs who believe that schemas never change ignore reality.

But if DBAs never refactor the database to make schema changes, how do they make changes to accommodate new abstractions? Unfortunately, *adding another join table* is a common process DBAs use to expand schema definitions. Rather than make a schema change and risk breaking existing systems, they instead just add a new table, joining it to the original using relational database primitives. While this works in the short term, it obfuscates the real underlying abstraction—in the real world, one entity is represented by multiple things. Over time, DBAs who rarely genuinely restructure schemas build an increasingly fossilized world, with byzantine grouping and bunching strategies. When DBAs don't restructure the database, they're not preserving a precious enterprise resource, they're instead creating the concretized remains of every version of the schema, all overlaid upon one another via join tables.

Legacy data quality presents another huge problem. Often, the data has survived many generations of software, each with their own persistence quirks, resulting in data that is inconsistent at best, and garbage at worst. In many ways, trying to keep

every scrap of data couples the architecture to the past, forcing elaborate workarounds to make things operate successfully.

Before trying to build an evolutionary architecture, make sure developers can evolve the data as well, both in terms of schema and quality. Poor structure requires refactoring, and DBAs should perform whatever actions are necessary to baseline the quality of data. We prefer fixing these problems early rather than building elaborate, ongoing mechanisms to handle these problems in perpetuity.

Legacy schemas and data have value, but they also represent a tax on the ability to evolve. Architects, DBAs, and business representatives need to have frank conversations about what represents *value* to the organization—keeping legacy data forever or the ability to make evolutionary change. Look at the data that has true value and preserve it, and make the older data available for reference but out of the mainstream of evolutionary development.

Refusing to refactor schemas or eliminate old data couples your architecture to the past, which is difficult to refactor.

## Case Study: Evolving PenultimateWidgets' Routing

PenultimateWidgets has decided to implement a new routing scheme between pages, providing a navigational breadcrumb trail to users. Doing so means changing the way routing between pages has been done (using an in-house framework). Pages that implement the new routing mechanism require more context (origin page, workflow state, and so on), and thus require more data.

Within the routing service quantum, PenultimateWidgets currently has a single table to handle routes. For the new version, developers need more information, so the table structure will be more complex. Consider the starting point illustrated in Figure 5-4.

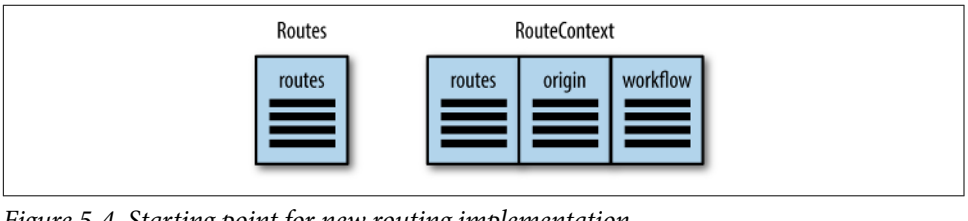



*Figure 5-4. Starting point for new routing implementation*

Not all pages at PenultimateWidgets will implement the new routing at the same time because different business units work at different speeds. Thus, the routing service

must support both old and new versions. We will see how that is handled via routing in Chapter 6. In this case, we must handle the same scenario at the data level.

Using the expand/contract pattern, a developer can create the new routing structure and make it available via the service call. Internally, both routing tables have a trigger associated with the `route` column, so that changes to one are automatically replicated to the other, as shown in Figure 5-5.

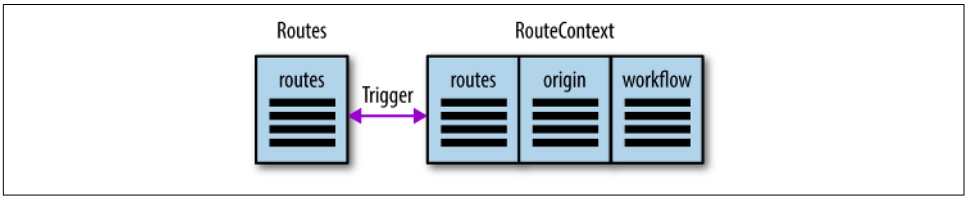



*Figure 5-5. The transitional state, where the service supports both versions of routing*

As seen in Figure 5-5, the service can support both APIs as long as developers need the old routing service. In essence, the application now supports two versions of routing information.

When the old service is no longer needed, the routing service developers can remove the old table and the trigger, as shown in Figure 5-6.

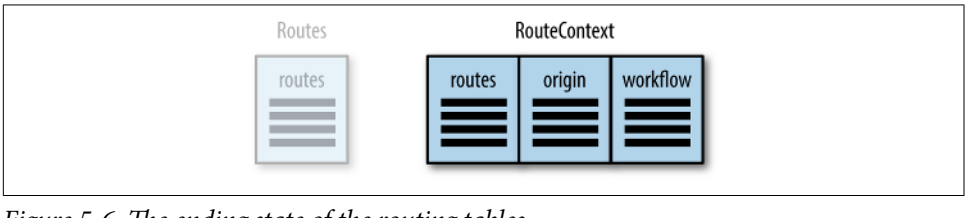



*Figure 5-6. The ending state of the routing tables*

In Figure 5-6, all services have migrated to the new routing capability, allowing the old service to be removed. This matches the workflow shown in Figure 5-2.

The database can evolve right alongside the architecture as long as developers apply proper engineering practices such as continuous integration, source control, and so on. This ability to easily change the database schema is critical: a database represents an abstraction based on the real world, which can change unexpectedly. While data abstractions resist change better than behavior, they must still evolve. Architects must treat data as a primary concern when building an evolutionary architecture.

Refactoring databases is an important skill and craft for DBAs and developers to hone. Data is fundamental to many applications. To build evolvable systems, developers and DBAs must embrace effective data practices alongside other modern engineering practices.

CHAPTER 6

# Building Evolvable Architectures

Until now, we've addressed the three primary aspects of evolutionary architecture—fitness functions, incremental change, and appropriate coupling—separately. Now we have enough context to tie them together.

Many of the concepts we discussed aren't new ideas, but rather viewed through a new lens. For example, testing has existed for years, but not with the fitness function emphasis on architectural verification. *Continuous Delivery* defined the idea of deployment pipelines. Evolutionary architecture shows architects the real utility of that capability.

Many organizations pursue Continuous Delivery practices as a way to increase engineering efficiency for software development, a worthy goal in itself. However, we're taking the next step, using those capabilities to create something more sophisticated—architectures that evolve with the real world.

So how can developers take advantage of these techniques on projects, both existing and new?

## Mechanics

Architects can operationalize these techniques for building an evolutionary architecture in three steps:

### 1. Identify Dimensions Affected by Evolution

First, architects must identify which dimensions of the architecture they want to protect as it evolves. This always includes technical architecture, and usually things like data design, security, scalability, and the other "-ilities" architects have deemed important. This must involve other interested teams within the organization, includ-

ing business, operations, security, and other affected parties. The *Inverse Conway Maneuver* (described in "Conway's Law" on page 11) is helpful here because it encourages multirole teams. Basically, this is the the common behavior of architects at the onset of projects when identifying the architectural characteristics they want to support.

## 2. Define Fitness Function(s) for Each Dimension

A single dimension often contains numerous fitness functions. For example, architects commonly wire a collection of code metrics into the deployment pipeline to ensure architectural characteristics of the code base, such as preventing component dependency cycles. Architects document decisions about which dimensions deserve ongoing attention in a lightweight format such as a wiki. Then, for each dimension, they decide what parts may exhibit undesirable behavior when evolving, eventually defining fitness functions. Fitness functions may be automated or manual, and ingenuity will be necessary in some cases.

## 3. Use Deployment Pipelines to Automate Fitness Functions

Lastly, architects must encourage incremental change on the project, defining stages in a deployment pipeline to apply fitness functions and managing deployment practices like machine provisioning, testing, and other DevOps concerns. Incremental change is the engine of evolutionary architecture, allowing aggressive verification of fitness functions via deployment pipelines and a high degree of automation to make mundane tasks like deployment invisible. Cycle time is the Continuous Delivery measure of engineering efficiency. Part of the responsibility of developers on projects that support evolutionary architecture is to maintain good cycle time. Cycle time is an important aspect of incremental change because many other metrics derive from it. For example, the velocity of new generations appearing in an architecture is proportional to its cycle time. In other words, if a project's cycle time lengthens, it slows down how fast the project can deliver new generations, which affects evolvability.

While the identification of dimensions and fitness functions occurs at the beginning of a new project, it is also an ongoing activity for both new and existing projects. Software suffers from the *unknown unknowns* problem: developers cannot anticipate everything. During construction, some part of the architecture often shows troubling signs, and building fitness functions can prevent this dysfunction from growing. While some fitness functions will naturally come to light at the beginning of a project, many won't reveal themselves until an architectural stress point appears. Architects must vigilantly watch for situations where nonfunctional requirements break and retrofit the architecture with fitness functions to prevent future problems.

# Greenfield Projects

Building evolvability into new projects is much easier than retrofitting existing ones. First, developers have the opportunity to utilize incremental change right away, building a deployment pipeline at project inception. Fitness functions are easier to identify and plan before any code exists, making it easier to accommodate complex fitness functions because scaffolding has existed since inception. Second, architects don't have to untangle any undesirable coupling points that creep into existing projects. The architect can also put metrics and other verifications in place to ensure architectural integrity as the project changes.

Building new projects that handle unexpected change is easier if a developer chooses the correct architectural patterns and engineering practices to facilitate evolutionary architecture. For example, microservices architectures offer extremely low coupling and a high degree of incremental change, making that style an obvious candidate (and another contributing factor to its popularity).

# Retrofitting Existing Architectures

Adding evolvability to existing architectures depends on three factors: component coupling, engineering practice maturity, and developer ease in crafting fitness functions.

## Appropriate Coupling and Cohesion

Component coupling largely determines the evolvability of the technical architecture. Yet the best possible evolvable technical architecture is doomed if the data schema is rigid and fossilized. Cleanly decoupled systems make evolution easy; nests of exuberant coupling harm it. To build truly evolvable systems, architects must consider all affected dimensions of an architecture.

Beyond the technical aspects of coupling, architects must also consider and defend the functional cohesion of the components of their system. When migrating from one architecture to another, the functional cohesion determines the ultimate granularity of restructured components. That doesn't mean architects can't decompose components to a ridiculous level, but rather that components should have an appropriate size based on the problem context. For example, some business problems are more coupled than others, such as in the case of heavily transactional systems. Trying to build an extremely decoupled architecture that is counter to the problem is unproductive.

Understand the business problem before choosing an architecture.

While this advice seems obvious, we constantly see teams that have chosen the shiniest new architectural pattern rather than the most appropriate one suffer. Part of choosing an architecture lies in understanding where the problem and physical architecture come together.

## Engineering Practices

Engineering practices matter when defining how evolvable an architecture can be. While Continuous Delivery practices don't guarantee evolutionary architecture, it is almost impossible without them.

Many teams embark on improved engineering practices for the sake of efficiency. However, once those practices cement, they become building blocks for advanced capabilities such as evolutionary architecture. Thus, the ability to build an evolutionary architecture is an incentive to improving efficiency.

Many companies reside in the transition zone between older practices and new. They may have solved low hanging fruit like continuous integration but still have largely manual testing. While it slows cycle time, it is important to include manual stages in deployment pipelines. First, it treats each stage of an application's build the same—as a stage in the pipeline. Second, as teams slowly automate more pieces of deployment, manual stages may become automated ones with no disruption. Third, elucidating each stage brings awareness about the mechanical parts of the build, creating a better feedback loop and encouraging improvements.

The biggest single common impediment to building evolutionary architecture is intractable operations. If developers cannot easily deploy changes, all parts of the feedback cycle are hampered.

## Fitness Functions

Fitness functions form the protective substrate of an evolutionary architecture. If architects design a system around particular characteristics, those features may be orthogonal to testability. Fortunately, modern engineering practices have vastly improved around testability, making formerly difficult architectural characteristics automatically verifiable. This is the area of evolutionary architecture that requires the most work, but fitness functions allows equal treatment for formerly disparate concerns.

We encourage architects to start thinking of all kinds of architectural verification mechanisms as fitness functions, including things they have previously considered ad hocly. For example, many architectures have a service-level agreement around scalability and corresponding tests. They also have rules around security requirements, with accompanying verification mechanisms. Architects often think of these as separate categories, but both intents are the same: verify some feature of the architecture. By thinking of all architectural verification as fitness functions, there is more consistency when automation and other beneficial synergistic interactions are defined.

### Refactoring Versus Restructuring

Developers sometimes co-opt terms that sound cool and make them into broader synonyms, as is the case for *refactoring*. As defined by Martin Fowler, refactoring is the process of restructuring existing computer code without changing its external behavior. For many developers, *refactoring* has become synonymous with *change*, but there are key differences.

It is exceedingly rare that a team refactors an architecture; rather, they *restructure* it, making substantive changes to both structure and behavior. Architecture patterns exist in part to make certain architectural characteristics primary in an application. Switching patterns entails switching priorities, which isn't refactoring. For example, architects might choose an EDA for scalability. If the team switches to a different architectural pattern, it likely won't support the same level of scalability.

## COTS Implications

In many organizations, developers don't own all the parts that make up their ecosystem. COTS (Commercial off-the-shelf) and package software is prevalent in large companies, creating challenges for architects building evolvable systems.

COTS systems must evolve alongside other applications within an enterprise. Unfortunately, these systems don't support evolution well.

*Incremental change*
: Most commercial software falls woefully short of industry standards for automation and testing. Architects and developers must often ring fence integration points and build whatever testing is possible, frequently treating the entire system as a black box. Enforcing agility in terms of deployment pipelines, DevOps, and other modern practices offers challenges to development teams.

*Appropriate coupling*
: Package software often commits the worst sins in terms of coupling. Generally, the system is opaque, with a defined API developers use to integrate. Inevitably, that API suffers from the problem described in "Antipattern: Last 10% Trap" on

page 127, allowing almost (but not quite) enough flexibility for developers to get useful work done.

*Fitness functions*
: Adding fitness functions to package software is perhaps the biggest hurdle to enable evolvability. Generally, tools of this ilk don't expose enough internals to allow unit or component testing, making behavioral integration testing the last resort. These tests are less desirable because they are necessarily coarse grained, must run in a complex environment, and must test a large swath of behavior of the system.

Work diligently to hold integration points to your level of maturity. If that isn't possible, realize that some parts of the system will be easier for developers to evolve than others.

Another worrisome coupling point introduced by many package software vendors is opaque database ecosystems. In the best-case scenarios, the package software manages the state of the database entirely, exposing selected appropriate values via integration points. In the worst case, the vendor database *is* the integration point to the rest of the system, vastly complicating changes on either side of the API. In this case, architects and DBAs must wrestle control of the database away from the package software for any hope of evolvability.

If trapped with necessary package software, architects should build as robust a set of fitness functions as possible and automate their running at every possible opportunity. Lack of access to internals relegates testing to less desirable techniques.

## Migrating Architectures

Many companies end up migrating from one architectural style to another. For example, architects choose simple-to-understand architecture patterns at the beginning of a company's IT history, often layered architecture monoliths. As the company grows, the architecture comes under stress. One of the most common paths of migration is from monolith to some kind of service-based architecture, for reasons of the general domain-centric shift in architectural thinking, covered in "Microservices" on page 68. Many architects are tempted by the highly evolutionary microservices architecture as a target for migration, but this is often quite difficult, primarily because of existing coupling.

When architects think of migrating architecture, they typically think of the coupling characteristics of classes and components, but ignore many other dimensions affected by evolution, such as data. Transactional coupling is as real as coupling between

classes, and just as insidious to eliminate when restructuring architecture. These extra-class coupling points become a huge burden when trying to break the existing modules into too-small pieces.

Many senior developers build the same types of applications year after year, and become bored with the monotony. Most developers would rather *write* a framework than *use* a framework to create something useful: *Meta-work is more interesting than work*. Work is boring, mundane, and repetitive, whereas building new stuff is exciting.

This manifests in two ways. First, many senior developers start writing the infrastructure that other developers use, rather than using existing (often open source) software. We once worked with a client who had once been on the cutting edge of technology. They built their own application server, web framework in Java, and just about every other bit of infrastructure. At one point, we asked if they had built their own operating system, too, and when they said, "No," we asked, "Why not?!? You built everything else from scratch!"

Upon reflection, the company needed capabilities that weren't available. However, when open-source tools became available, they already owned their lovingly handcrafted infrastructure. Rather than cut over to the more standard stack, they opted to keep their own because of minor differences in approach. A decade later, their best developers worked in full-time maintenance mode, fixing their application server, adding features to their web framework, and other mundane chores. Rather than applying innovation on building better applications, they permanently slaved away on plumbing.

Architects aren't immune to the "meta-work is more interesting than work" syndrome, which manifests in choosing inappropriate but buzz-worthy architectural styles like microservices.

Don't build an architecture just because it will be fun meta-work.

## Migration Steps

Many architects find themselves faced with the challenge of migrating an outdated monolithic application to a more modern service-based approach. Experienced architects realize that a host of coupling points exist in applications, and one of the first tasks when untangling a code base is understanding how things are joined. When decomposing a monolith, the architect must take coupling and cohesion into account to find the appropriate balance. For example, one of the most stringent constraints of

the microservices architectural style is the insistance that the database reside inside the service's bounded context. When decomposing a monolith, even if it is possible to break the classes into small enough pieces, breaking the transactional contexts into similar pieces may present an unsurmountable hurdle.

When restructuring architecture, consider *all* the affected dimensions.

Many architects end up migrating from monolithic applications to service-based architectures. Consider the starting point architecture shown in Figure 6-1.

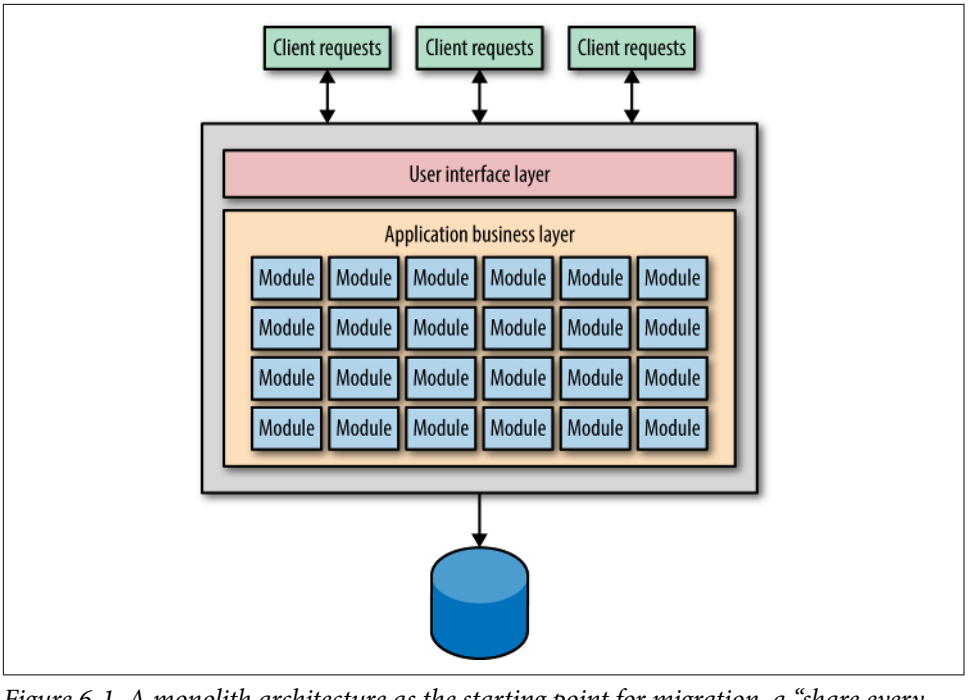



*Figure 6-1. A monolith architecture as the starting point for migration, a "share everything" architecture*

Building extremely granular services is easier in new projects but difficult in existing migrations. So, how can we migrate the architecture in Figure 6-1 to the service-based architecture shown in Figure 6-2?

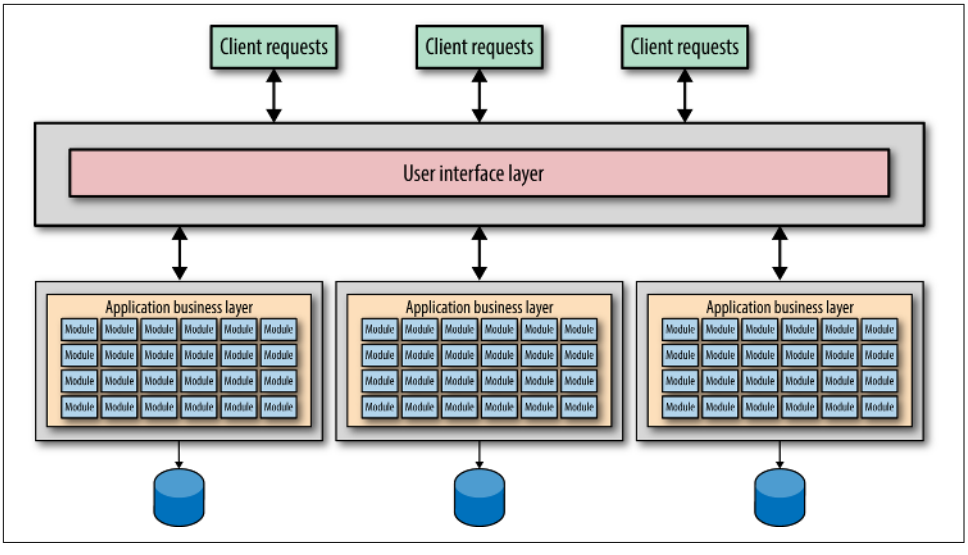



*Figure 6-2. The service-based, "share as little as possible" end result of the migration*

Performing the kind of migration shown in Figures 6-1 and 6-2 comes with a host of challenges: service granularity, transactional boundaries, database issues, and things like how to handle shared libraries. Architects must understand why they want to perform this migration, and it must be a better reason than "it's the current trend." Splitting the architecture into domains, along with better team structure and operational isolation, allows for easier incremental change, one of the building blocks of evolutionary architecture, because the focus of work matches the physical work artifacts.

When decomposing a monolithic architecture, finding the correct service granularity is key. Creating large services alleviates problems like transactional contexts and orchestration, but does little to break the monolith into smaller pieces. Too-fine-grained components lead to too much orchestration, communication overhead, and interdependency between components.

For the first step in migrating architecture, developers identify new service boundaries. Teams may decide to break monoliths into services via a variety of partitioning as follows:

*Business functionality groups*
: A business may have clear partitions that mirror IT capabilities directly. Building software that mimics the existing business communication hierarchy falls distinctly into an applicable use of Conway's Law (see "Conway's Law" on page 11).

*Transactional boundaries*
: Many businesses have extensive transaction boundaries they must adhere to. When decomposing a monolith, architects often find that transactional coupling is the hardest to break apart, as discussed in "Two-Phase Commit Transactions" on page 90.

*Deployment goals*
: Incremental change allows developers to selectively release code on different schedules. For example, the marketing department might want a much higher cadence of updates than inventory. Partitioning services around operational concerns like speed to release makes sense if that criteria is highly important. Similarly, a portion of the system may have extreme operational characteristics (like scalability). Partitioning services around operational goals allows developers to track (via fitness functions) health and other operational metrics of the service.

Coarser service granularity means many of the coordination problems inherent in microservices go away because more of the business context resides inside a single service. However, the larger the service, the more operational difficulties tend to escalate (another architectural tradeoff).

## Evolving Module Interactions

Migrating shared modules (including components) is another common challenge faced by developers. Consider the structure shown in Figure 6-3.

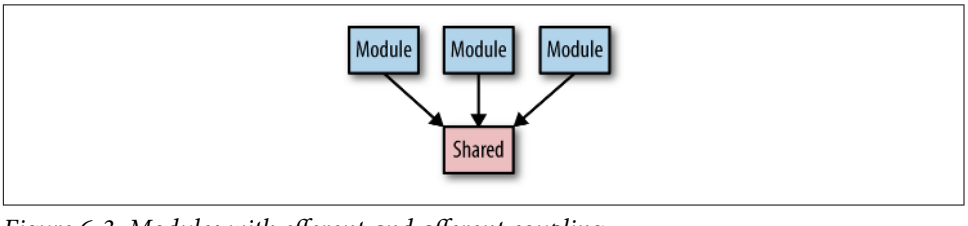



*Figure 6-3. Modules with efferent and afferent coupling*

In Figure 6-3, all three modules share the same library. However, the architect needs to split these modules into separate services. How can she maintain this dependency?

Sometimes, the library may be split cleanly, preserving the separate functionality each module needs. Consider the situtation shown in Figure 6-4.

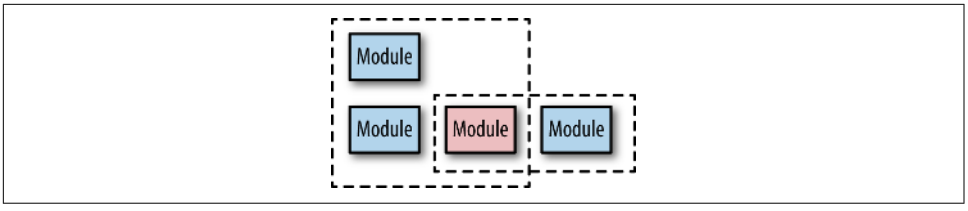



*Figure 6-4. Modules with a common dependency*

In Figure 6-4, both modules need the conflicting one shown in red. If developers are lucky, the functionality may be cleanly split down the middle, partitioning the shared library into the relevant parts needed by each dependent, as shown in Figure 6-5.

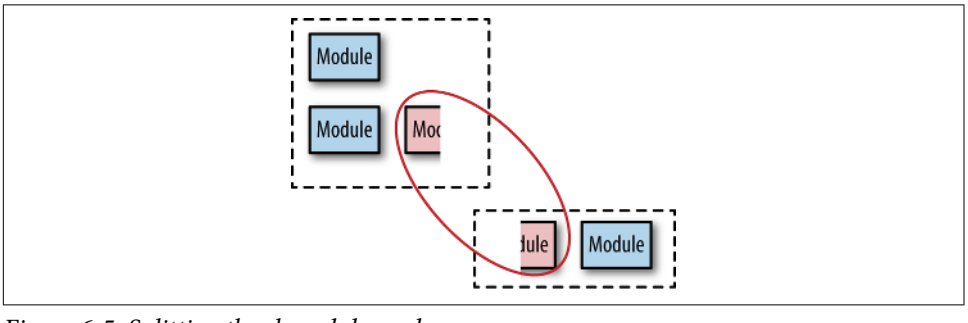



*Figure 6-5. Splitting the shared dependency*

However, it's more likely the shared library won't split that easily. In that case, developers can extract the module into a shared library (such as a JAR, DLL, gem, or some other component mechanism) and use it from both locations, as shown in Figure 6-6.

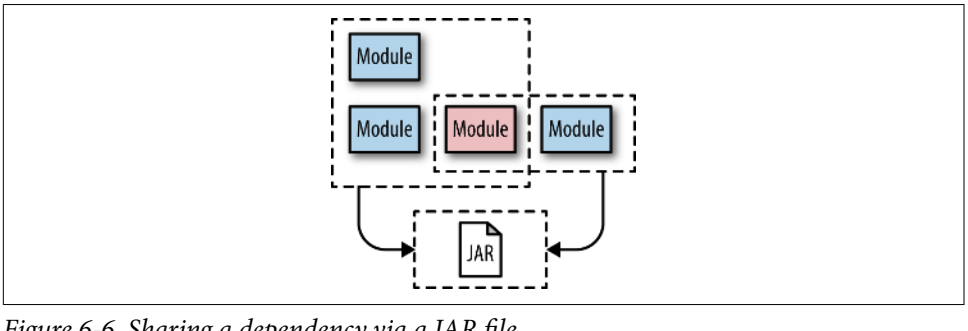



*Figure 6-6. Sharing a dependency via a JAR file*

Sharing is a form of coupling, which is highly discouraged in architectures like microservices. An alternative to sharing a library is replication, as illustrated in Figure 6-7.

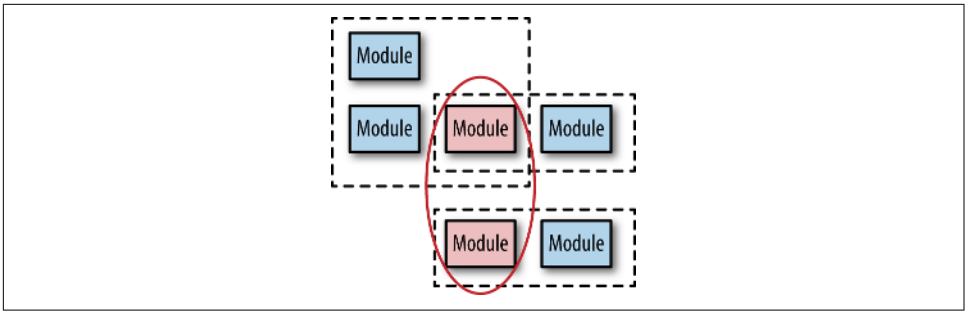



*Figure 6-7. Duplicating a shared library to eliminate a coupling point*

In a distributed environment, developers may achieve the same kind of sharing using messaging or service invocation.

When developers have identified the correct service partitioning, the next step is *separation* of the business layers from the UI. Even in microservices architectures, the UIs often resolve back to a monolith—after all, developers must show a unified UI at some point. Thus, developers commonly separate the UIs early in the migration, creating a mapping proxy layer between UI components and the back-end services they call. Separating the UI also creates an anticorruption layer, insulating UI changes from architecture changes.

The next step is *service discovery*, allowing services to find and call one another. Eventually, the architecture will consist of services that must coordinate. By building the discovery mechanism early, developers can slowly migrate parts of the system that are ready to change. Developers often implement service discovery as a simple proxy layer: each component calls the proxy, which in turn maps to the specific implementation.

> All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.
>
> —Dave Wheeler and Kevlin Henney

Of course, the more levels of indirection developers add, the more confusing navigating the services becomes.

When migrating an application from a monolithic application architecture to a more services-based one, the architect must pay close attention to how modules are connected in the existing application. Naive partitioning introduces serious performance problems. The connection points in application become integration architecture connections, with the attendant latency, availability, and other concerns. Rather than tackle the entire migration at once, a more pragmatic approach is to gradually decompose the monolithic into services, looking at factors like transaction boundaries, structural coupling, and other inherent characteristics to create several restruc-

turing iterations. At first, break the monolith into a few large "portions of the application" chunks, fix up the integration points, and rinse and repeat. Gradual migration is preferred in the microservices world.

> When migrating from a monolith, build a small number of larger services first.
>
> —Sam Newman, *Building Microservices*

Next, developers choose and detach the chosen service from the monolith, fixing any calling points. Fitness functions play a critical role here—developers should build fitness functions to make sure the newly introduced integration points don't change, and add consumer-driven contracts.

# Guidelines for Building Evolutionary Architectures

We've used a few biology metaphors throughout the course of the book, and here is another. Our brains evolved not in a nice, pristine environment where each capability was carefully built. Instead, each layer is based on primeval layers beneath. Much of our core autonomic behavior (like breathing, hunger, and so on) resides in parts of our brain not very different from reptilian brains. Instead of wholesale replacement of core mechanisms, evolution builds new layers on top.

Software architecture in large enterprises follows a similar pattern. Rather than rebuild each capability anew, most companies try to adapt whatever is present. As much as we like to talk about architecture in pristine, idealized settings, the real world often exhibits a contrary mess of technical debt, conflicting priorities, and limited budgets. Architecture in large companies is built like the human brain—lower-level systems still handle critical plumbing details but have some old baggage. Companies hate to decommission something that works, leading to escalating integration architecture challenges.

Retrofitting evolvability into an existing architecture is challenging—if developers never built easy change into the architecture, it is unlikely to appear spontaneously. No architect, now matter how talented, can transform a Big Ball of Mud into a modern microservices architecture without immense effort. Fortunately, projects can receive benefits without changing their entire architecture by building some flexibility points into the existing one.

## Remove Needless Variability

One of the goals of Continuous Delivery is stability—building on known good parts. A common manifestation of this goal is the modern DevOps perspective on building immutable infrastructure. We discussed the dynamic equilibrium of the software development ecosystem in Chapter 1—nowhere is that more apparent in how much the foundation shifts around software dependencies. Software systems undergo constant change, as developers update capabilities, issue service packs, and generally

tweak their software. Operating systems are a great example, as they endure constant change.

Modern DevOps has solved the dynamic equilibrium problem locally by replacing *snowflakes* with *immutable infrastructure*. *Snowflake* computers are ones that have been manually crafted by an operations person, and all future maintenance is done by hand. Chad Fowler coined the term *immutable infrastructure* in his blog post, "Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components". Immutable infastructure refers to systems defined entirely programmatically. All changes to the system must occur via the source code, not by modifying the running operating system. Thus, the entire system is immutable from an operational standpoint—once the system is bootstrapped, no other changes occur.

While immutability may sound like the opposite of evolvability, quite the opposite is true. Software systems comprise thousands of moving parts, all interlocking in tight dependencies. Unfortunately, developers still struggle with unanticipated side effects of changes to one of those parts. By locking down the possibility of unanticipated change, we control more of the factors that make systems fragile. Developers strive to replace variables in code with constants to reduce vectors of change. DevOps introduced this concept to operations, making it more declarative.

Immutable infrastructure follows our advice to *remove needless variables*. Building software systems that evolve means controlling as many unknown factors as possible. It is virtually impossible to build fitness functions that can anticipate how the latest service pack of the operating system might affect the application. Instead, developers build the infrastructure anew each time the deployment pipeline executes, catching breaking changes as aggressively as possible. If developers can remove known foundational, changeable parts such as the operating system as a possibility, they have less ongoing testing burden to carry.

Architects can find all sorts of avenues to convert changeable things to constants. Many teams extend the immutable infrastructure advice to the development environment as well. How many times has some team member exclaimed, "But it works on my machine!"? By ensuring every developer has the exact same image, a host of needless variables disappear. For example, most development teams automate the update of development libraries through repositories, but what about updates to tools like IDEs? By capturing the development environment as immutable infrastructure, developers always work on the same foundation.

Building an immutable development environment also allows useful tools to spread throughout projects. Pair programming is a common practice in many agile engineering-focused development teams, including pair rotation, where each team member changes on a regular basis, from every few hours to every few days. However, it's frustrating when a tool appears on the computer a developer used yesterday

that isn't present today. By building a single source for developer systems, it becomes easy to add useful tools to all systems at once.

### The Hazards of Snowflakes

A story in a popular blog called, "Knightmare: A DevOps Cautionary Tale" serves as a cautionary tale of snowflake servers. A financial services company previously had an algorithm called PowerPeg that handled trading details, but that code hadn't been used in a number of years. However, the developers never removed the code. It resided underneath a feature toggle that remained off. Because of regulatory changes, developers implemented a new trading algorithm called SMARS. Because they were lazy, they decided to reuse the old PowerPeg feature flag to implement the new SMARS code. On August 1, 2012, developers deployed the new code to seven servers. Unfortunately, their system ran on eight servers and one of them wasn't updated. When they enabled the PowerPeg feature toggle, seven servers started selling…and the other started buying! Developers had accidentally set up the worst market scenario—they had automated selling low and buying high. Convinced that the new code was the culprit, developers rolled back the new code on the seven servers, but left the feature toggle on, meaning the PowerPeg code now ran on all servers. It took them 45 minutes to reign in the chaos, with a loss of over $400 million. Luckily, an angel investor saved them, as that was more than the company was worth.

This story highlights the problems with unknown variability. Reusing an old feature flag is reckless—the best practice for feature flags is removing them aggressively as soon as their purpose is fulfilled. Not automating deploying critical software to servers is also considered reckless in modern DevOps environments.

Identify and remove needless variability.

## Make Decisions Reversible

Inevitably, systems that aggressively evolve will fail in unanticipated ways. When these failures occur, developers need to craft new fitness functions to prevent future occurrences. But how do you recover from a failure?

Many DevOps practices exist to allow *reversible decisions*—decisions that need to be undone. For example *blue/green deployments*, where operations have two identical (probably virtual) ecosystems—*blue* and *green* ones—common in DevOps. If the current production system is running on *blue*, *green* is the staging area for the next release. When the *green* release is ready, it becomes the production system and *blue*

temporarily shifts to backup status. If something goes awry with *green*, operations can go back to *blue* without too much pain. If *green* is fine, *blue* becomes the staging area for the next release.

*Feature toggles* are another common way developers make decisions reversible. By deploying changes underneath feature toggles, developers can release them to a small subset of users (called canary releasing) to vet the change. If a feature behaves unexpectedly, developers can switch the toggle back to the original and correct the fault before trying again. Make sure you remove the outdated ones!

Using feature toggles greatly reduces risk in these scenarios. Service routing—routing to a particular instance of a service based on request context—is another common method to canary release in microservices ecosystems.

Make as many decisions as possible reversible (without over-engineering).

## Prefer Evolvable over Predictable

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

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

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

While no architecture can survive the unknown, we know that dynamic equilibrium renders predictability useless in software. Instead, we prefer to build *evolvability* into software: if projects can easily incorporate changes, architects don't need a crystal ball. Architecture is not a solely upfront activity—projects constantly change in both explicit and unexpected ways throughout their life. One technique commonly used by developers to insulate themselves from change is to use an *anticorruption layer*.

## Build Anticorruption Layers

Projects often need to couple themselves to libraries that provide incidental plumbing: message queues, search engines, and so on. The *abstraction distraction antipattern* describes the scenario where a project "wires" itself too much to an external library, either commercial or open source. Once it becomes time for developers to upgrade or switch the library, much of the application code utilizing the library has baked-in assumptions based on the previous library abstractions. Domain-driven design includes a safeguard against this phenomenon called an *anticorruption layer*. Here is an example.

Agile architects prize the *last responsible moment* principle when making decisions, which is used to counter the common hazard in projects of buying complexity too early. We worked intermittently on a Ruby on Rails project for a client who managed wholesale car sales. After the application went live, an unexpected workflow arose. It turned out that used car dealers tended to upload new cars to the auction site in large batches, both in number of cars and number of pictures per car. We realized that, as much as the general public doesn't trust used-car dealers, dealers *really* don't trust each other; thus, each car must include a photo covering essentially every molecule of the car. Users wanted a way to begin an upload, then either get progress via some UI mechanism like a progress bar or check back later to see if the batch was done. Translated to technical terms, they wanted asynchronous upload.

A message queue is one traditional architectural solution to this problem, and the team discussed whether to add an open source queue to the architecture. A common trap at this juncture for many projects is the attitude of, "We know we'll need a message queue for lots of stuff eventually, so let's get the fanciest one we can now and grow into it later." The problem with this approach is *technical debt*: stuff that's part of your project that isn't supposed to be there and is in the way of stuff that is supposed to be there. Most developers treat crufty old code as the only form of technical debt, but projects can inadvertently *buy* technical debt as well via premature complexity.

For the project, the architect encouraged developers to find a simpler way. One developer discovered BackgrounDRb, an extraordinarily simple open source library that simulates a single message queue backed by a relational database. The architect knew this simple tool would probably never scale to other future problems, but she didn't have other objections. Rather than try to predict future usage, she instead made it relatively easy to replace by placing it behind an API. In the *last responsible moment* answer questions such as "Do I have to make this decision now?", "Is there a way to safely defer this decision without slowing any work?", and "What can I put in place now that will suffice but I can easily change later if needed?"

Around the one-year anniversary, a second request for asynchronicity appeared in the form of timed events around sales. The architect evaluated the situation and decided that a second instance of BackgrounDRb would suffice, put it in place, and

moved on. At around the two-year anniversary, a third request appeared for constantly updating values like caches and summaries. The team realized that the current solution couldn't handle the new workload. However, they now had a good idea about what kind of asynchronous behavior the application needed. At that point, the project switched over to Starling, a simple but more traditional message queue. Because the original solution was isolated behind an interface, it took one pair of developers less than one iteration (one week on that project) to complete the transition—without disrupting other developers' work on the project.

Because the architect put an anticorruption layer in place with an interface, replacing one piece of functionality became a mechanical exercise. Building an anticorruption layer encourages the architect to think about the *semantics* of what they need from the library, not the *syntax* of the particular API. But this is not an excuse to *abstract all the things!* Some development communities love preemptive layers of abstraction to a distracting degree but understanding suffers when you must call a `Factory` to get a `proxy` to a remote interface to a `Thing`. Fortunately, most modern languages and IDEs allow developers to be *just in time* when extracting interfacs. If a project finds themselves bound to an out-of-date library in need of change, the IDE can *extract interface* on behalf of the developer, making a Just In Time (JIT) anticorruption layer.

Build just-in-time anticorruption layers to insulate against library changes.

Controlling the coupling points in an application, especially to external resources, is one of the key responsibilities of an architect. Try to find the pragmatic time to add dependencies. As an architect, remember dependencies provide benefits but also impose constraints. Make sure the benefits outweigh the cost in updates, dependency management, and so on.

> Developers understand the benefits of everything and the tradeoffs of nothing!
>
> —Rich Hickey, *creator of Clojure*

Architects must understand both benefits and tradeoffs and build engineering practices accordingly.

Using anticorruption layers encourages evolvability. While architects can't predict the future, we can at least lower the cost of change so that it doesn't impact us so negatively.

## Case Study: Service Templates

Microservices architectures are designed to be *share nothing* architectures—each component is as decoupled as possible from other components, adhering to the bounded context principle. However, the shunning of coupling between services pertains primarily to domain classes, database schemas, and other coupling points that harm the ability to evolve. Development teams often want to manage some aspects of the technical coupling uniformly—adhering to our *remove needless variables* advice—to ensure uniformity. For example, monitoring, logging, and other diagnostics are critical in this architectural style due to the profusion of moving parts. When operations must manage thousands of services, when service teams forget to add monitoring capabilities to their service the results can be disasterous. Upon deployment, the service will disappear into a black hole, because in these environments, if it can't be monitored, it is invisible. Yet, in a highly decoupled environment, how can teams enforce consistency?

*Service templates* are one common solution for ensuring consistency. These are preconfigured sets of common infrastructure libraries like service discovery, monitoring, logging, metrics, authentication/authorization, and so on. In large organizations, a shared infrastructure team manages the service templates. Service implementation teams use the template as scaffolding, writing their behavior within. If the logging tool requires an upgrade, the shared infrastructure team can mange it orthogonally from the service teams—they should never know (or care) that the change occurred. If a breaking change occurs, it fails during the provisioning phase of the deployment pipeline, alerting developers to the problem as soon as possible.

This is a good example of what we mean when we espouse *appropriate coupling*. Duplicating technical architecture functionality across services creates a slew of well-known problems. By finding exactly the level of coupling we need, we can free evolution without creating new problems.

Use service templates to couple just the appropriate parts of architecture together—the infrastructure elements that allow teams to benefit from coupling.

Service templates exemplify adaptability. Eliminating the technical architecture as the primary structure of the system makes it easier to target changes to just that dimension of architecture. When developers build a layered architecture, change is easy within each layer but highly coupled across layers. While a layered architecture partitions the technical architecture parts together, it entangles other concerns like domain, security, operations, etc. By building a part of the architecture solely for technical architecture concerns (e.g., service templates), developers can isolate and

unify change to that entire dimension. We discuss how to think about architectural elements as deployable units in Chapter 4.

## Build Sacrificial Architectures

In his book *Mythical Man Month*, Fred Brooks says to Plan to Throw One Away when building a new software system.

> The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. […] Hence plan to throw one away; you will, anyhow.
>
> —Fred Brooks

His point was that once a team has built a system, they know all the unknown unknowns and proper architecture decisions that are never clear from the outset—the next version will profit from all those lessons. At an architectural level, developers struggle to anticipate radically changing requirements and characteristics. One way to learn enough to choose a correct architecture is build a proof of concept. Martin Fowler defines a sacrificial architecture as an architecture designed to be thrown away if the concept proves successful. For example, eBay started as a set of Perl scripts in 1995, migrated to C++ in 1997, and then to Java in 2002. Obviously, eBay has been a resounding success in spite of rearchitecting their system several times. Twitter is another good example of successful utilization of this approach. When Twitter released, it was written in Ruby on Rails to achieve fast time-to-market. However, as Twitter became popular, the platform couldn't support the scale, resulting in frequent crashes and limited availability. Many early users became all too familiar with their failure beacon, shown in Figure 6-8.

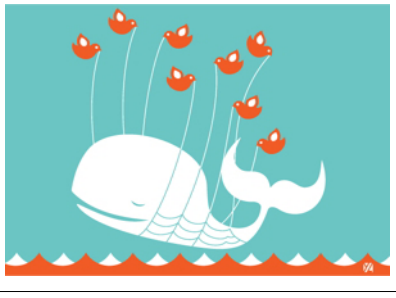


*Figure 6-8. Twitter's famous Fail Whale*

Thus, Twitter restructured their architecture to replace the backend with something more robust. However, it could be argued that this tactic is the reason the company survived. If the Twitter engineers had built the final, robust platform from the beginning, it would have delayed their entry into the market long enough for *Snitter* or some alternative short-form messaging service to beat them to market. Despite the growing pains, starting with a sacrificial architecture eventually paid off.

Cloud environments make sacrificial architecture more attractive. If developers have a project they want to test, building the initial version in the cloud greatly reduces the resources required to release the software. If the project is successful, architects can take the time to build a more suitable architecture. If developers are careful about anticorruption layers and other evolutionary architecture practices, they can mitigate some of the pains of the migration.

Many companies build a sacrificial architecture to achieve a minimum viable product to prove a market exists. While this is a good strategy, the team must eventually allocate time and resources to build a more robust architecture, hopefully less visibly than Twitter.

One other aspect of technical debt impacts many initially successful projects, elucidated again by Fred Brooks, when he refers to the *second system syndrome*—the tendency of small, elegant, and successful systems to evolve into giant, feature-laden monstrosities due to inflated expectations. Business people hate to throw away functioning code, so architecture tends toward always adding, never removing, or decommissioning.

Technical debt works effectively as a metaphor because it resonates with project experience, and represents faults in design, regardless of the driving forces behind them. Technical debt aggravates inappropriate coupling on projects—poor design frequently manifests as pathological coupling and other antipatterns that make restructuring code difficult. As developers restructure architecture, their first step should be to remove the historical design compromises that manifest as technical debt.

## Mitigate External Change

A common feature of every development platform is *external dependencies*: tools, frameworks, libraries, and other assets provided by and (more importantly) updated via the Internet. Software development sits on a towering stack of abstractions, each built on the abstractions before. For example, operating systems are an external dependency outside the developer's control. Unless companies want to write their own operating system and all other supporting code, they must rely on external dependencies.

Most projects rely on a dizzying array of third-party components, applied via build tools. Developers like dependencies because they provide benefits, but many developers ignore the fact that they come with a cost as well. When relying on code from a third party, developers must create their own safeguards against unexpected occur-

rences: breaking changes, unannounced removal, and so on. Managing these external parts of projects is critical to creating evolutionary architecture.

### The Eleven Lines of Code that Broke the Internet

In early 2016, JavaScript developers learned a harsh lesson about the hazards of depending on trivial things. A developer who had created a large number of small utilities became disgruntled because one of his modules clashed with the name of a commercial software project, which asked him to rename his module. Rather than comply, he removed more than 250 of his modules, including one library called `left pad.io`, eleven lines of code to pad strings with zeros or spaces (if 11 lines of code can be called a "library"). Unfortunately, many major JavaScript projects (including `node.js`) relied on this dependency. When it disappeared, everyone's JavaScript deployments broke.

The repository administrator for JavaScript packages took the unprecedented move of restoring the code to restore the ecosystem, but it spawned a deeper conversation in the community about the wisdom of the trends around dependency management.

This story contains two valuable lessons for architects. First, remember external libraries provide *both* benefits and cost. Make sure the benefits justify the cost. Second, don't allow external forces to affect the stability of your builds. If an upstream required dependency suddenly disappears, you should reject that change.

Edsger Dijkstra, a legendary figure in computer science, famously observed in 1968 that "Go To Statement Considered Harmful," where he punctured the existing best practice of unstructured coding, leading eventually to the structured programming revolution. Since that time, "considered harmful" has become a trope in software development.

> Transitive dependency management is our "considered harmful" moment.
>
> —Chris Ford (no relation to Neal)

Chris' point is that, until we recognize the severity of the problem, we cannot determine a solution. While we're not offering a solution to the problem, we need to highlight it because it critically affects evolutionary architecture. Stability is one of the foundations of both Continuous Delivery and evolutionary architecture. Developers cannot build repeatable engineering practices atop uncertainty. Allowing third parties to make changes to core dependencies defies this principle.

We recommend that developers take a more proactive approach to dependency management. A good start on dependency management models external dependencies using a *pull* model. For example, set up an internal version-control repository to act as a third-party component store, and treat changes from the outside world as pull

requests to that repository. If a beneficial change occurs, allow it into the ecosystem. However, if a core dependency disappears suddenly, reject that pull request as a destabilizing force.

Using a Continuous Delivery mindset, the third-party component repository utilizes its own deployment pipeline. When an update occurs, the deployment pipeline incorporates the change, then performs a build and smoke test on the affected applications. If successful, the change is allowed into the ecosystem. Thus, third-party dependencies use the same engineering practices and mechanisms of internal development, usefully blurring the lines across this often unimportant distinction between in-house written code and dependencies from third parties—at the end of the day, it's all code in a project.

## Updating Libraries Versus Frameworks

Architects make a common distinction between *libraries* and *frameworks*, with the colloquial definition of "a developer's code calls library whereas the framework calls a developer's code." Generally, developers subclass from frameworks (which in turn calls those derived classes), thus the distinction that the framework calls code. Conversely, library code generally comes as a collection of related classes and/or functions developers call as needed. Because the framework calls the developer's code, it creates a high degree of coupling to the framework. Contrast that with library code, which is generally more utilitarian code (like XML parsers, network libraries, etc.) and has a lower degree of coupling.

We prefer libraries because they introduce less coupling to your application, making them easier to swap out when the technical architecture needs to evolve.

One reason to treat libraries and frameworks differently comes down to engineering practices. Frameworks include capabilities such as UI, object-relational mapper, scaffolding like model-view-controller, and so on. Because the framework forms the scaffolding for the remainder of the application, all the code in the application is subject to impact by changes to the framework. Many of us have felt this pain viscerally—any time a team allows a fundamental framework to become outdated by more than two major versions, the effort (and pain) to finally update it is excruciating.

Because frameworks are a fundamental part of applications, teams must be aggressive about pursuing updates. Libraries generally form less brittle coupling points than frameworks do, allowing teams to be more casual about upgrades. One informal governance model treats framework updates as *push* updates and library updates as *pull* updates. When a fundamental framework (one whose afferent/efferent coupling numbers are above a certain threshold) updates, teams should apply the update as soon as the new version is stable and the team can allocate time for the change. Even though it will take time and effort, the time spent early is a fraction of the cost if the team perpetually procrastinates on the update.

Because most libraries provide utilitarian functionality, teams can afford to update them only when new desired functionality appears, using more of an "update when needed" model.

Update framework dependencies aggressively; update libraries passively.

## Prefer Continuous Delivery to Snapshots

Many dependency management tools use a mechanism called *snapshots* to model in-flight development. A snapshot build was originally meant to indicate a component almost ready for release but still under development, the implication being that the code might change on a regular basis. Once a component is "blessed" with a version number, the -SNAPSHOT moniker drops away.

Developers use snapshots because of the historical assumption that testing is difficult and time consuming, leading developers to try to segregate things changing from things not changing.

In evolutionary architecture, we expect all things to change all the time, and build engineering practices and fitness functions to accommodate change. For example, when a project has excellent test coverage and a deployment pipeline, developers test every change to every component via the automated deployment pipeline. Developers have no reason to keep a "special" repository for each part of the project.

Prefer Continuous Delivery over snapshots for (external) dependencies.

Snapshots are an artifact from a development era where comprehensive testing wasn't common, storage was expensive, and verification was difficult. Today's updated engineering practices avoid inefficient handling of component dependencies.

*Continuous Delivery* suggested a more nuanced way to think about dependencies, repeated here. Currently, developers only have *static* dependencies, linked via version numbers captured as metadata in a build file somewhere. However, this isn't sufficient for modern projects, which need a mechanism to indicate *speculative updating*. Thus, as the book suggests, developers should introduce two new designations for externmal dependencies: *fluid* and *guarded*. *Fluid* dependencies try to automatically update themselves to the next version, using mechanisms like deployment pipelines. For

example, say that `order` fluidly relies on version 1.2 of `framework`. When `framework` updates itself to version 1.3, `order` tries to incorporate that change via its deployment pipeline, which is set up to rebuild the project anytime any part of it changes. If the deployment pipeline runs to completion, the fluid dependency between the components is updated. However, if something prevents successful completion—failed test, broken diamond dependency, or some other problem—the dependency is updated to a *guarded* reliance on `framework1.2`, which means the developer should try to determine and fix the problem, restoring the fluid dependency. If the component is truly incompatible, developers create a permanent static reference to the old version, eschewing future automatic updates.

None of the popular build tools support this level of functionality yet—developers must build this intelligence atop existing build tools. However, this model of dependencies works extremely well in evolutionary architectures, where cycle time is a critical foundational value, being proportional to many other key metrics.

### Version Services Internally

In any integration architecture, developers inevitably must version service endpoints as the behavior evolves. Developers use two common patterns to version endpoints, *version numbering* or *internal resolution*. For version numbering, developers create a new endpoint name, often including the version number, when a breaking change occurs. This allows older integration points to call the legacy version while newer ones call the newer version. The alternative is internal resolution, where callers never change the endpoint—instead, developers build logic into the endpoint to determine the context of the caller, returning the correct version. The advantage of retaining the name forever is less coupling to specific version numbers in calling applications.

In either case, severely limit the number of supported versions. The more versions, the more testing and other engineering burdens. Strive to support only two versions at a time, and only temporarily.

When versioning services, prefer internal versioning to numbering; support only two versions at a time.

## Case Study: Evolving PenultimateWidgets' Ratings

PenultimateWidgets has a microservices architecture so developers can make small changes. Let's look closer at details of one of those changes, switching star ratings, as outlined in Chapter 3. Currently, PenultimateWidgets has a star rating service, whose parts are shown in Figure 6-9.

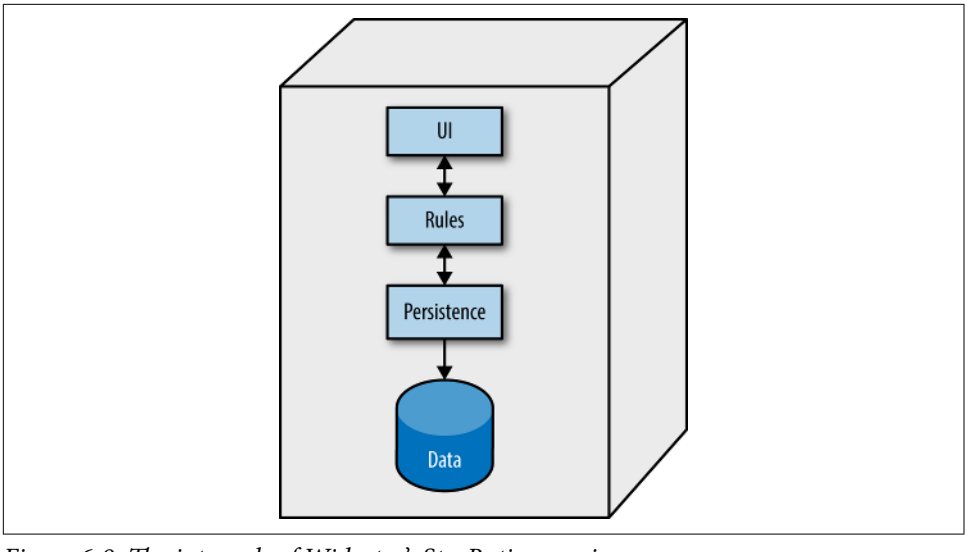



*Figure 6-9. The internals of Widgetco's StarRating service*

As shown in Figure 6-9, the star rating service consists of a database and a layered architecture, with persistence, business rules, and a UI. Not all of PenultimateWidgets' microservices include the UI. Some services are primarily informational, whereas others have UIs tightly coupled to the service's behavior, as is the case with star ratings. The database is a traditional relational database that includes a column to track ratings for a particular item ID.

When the team decided to update the service to support half-star ratings, they modified the original service as shown in Figure 6-10.

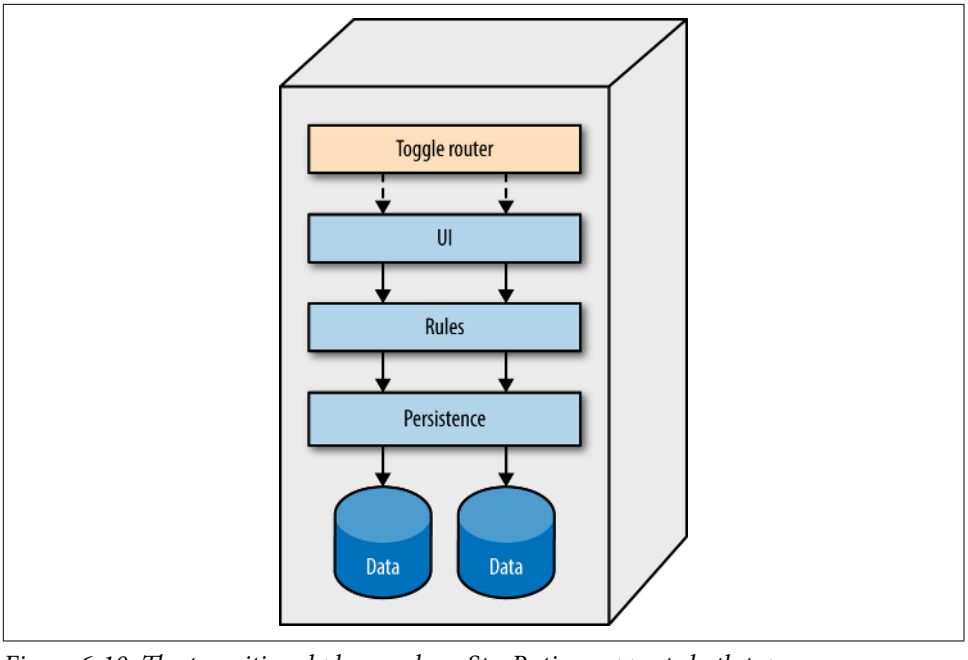



*Figure 6-10. The transitional phase, where StarRating supports both types*

In Figure 6-10, they added a new column to the database to handle the additional data—whether a rating has an additional half star. The architects also added a proxy component to our service to resolve the return differences at the service boundary. Rather than force calling services to "understand" the version numbers of this service, the star rating service resolves the request type, sending back whichever format is requested. This is an example of using *routing* as an evolutionary mechanism. The star rating service can exist in this state as long as some services still want star ratings.

Once the last dependent service has evolved away from whole-star ratings, developers can remove the old code path, as shown in Figure 6-11.

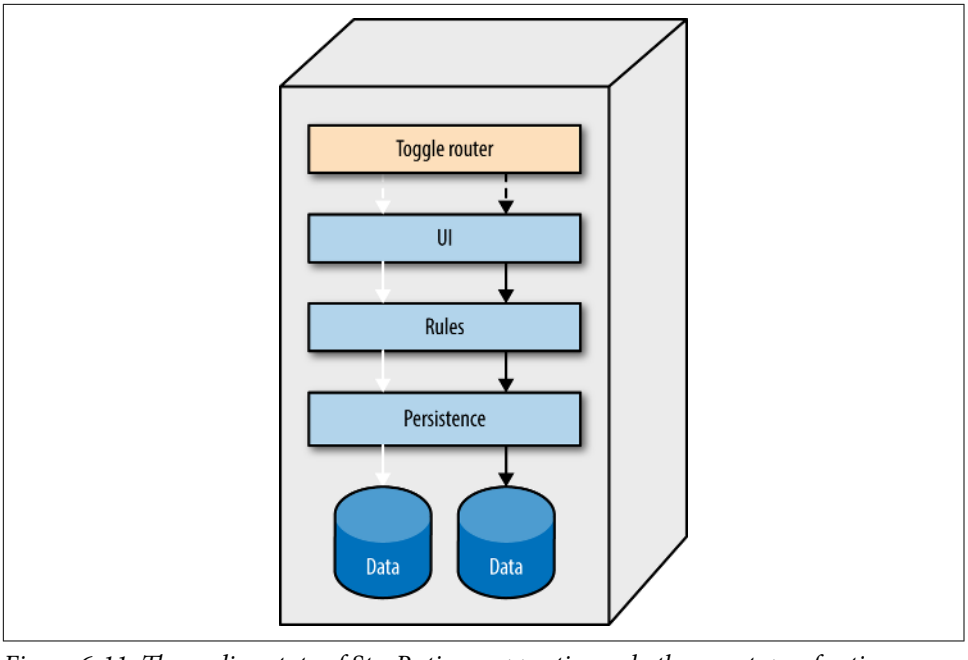



*Figure 6-11. The ending state of StarRating, supporting only the new type of rating*

As shown in Figure 6-11, developers can remove the old code path, and perhaps remove the proxy layer to handle version differences (or perhaps leave it to support future evolution).

In this case, PenultimateWidgets' change wasn't difficult from a data evolution standpoint because the developers were able to make an additive change, meaning they can add to the database schema rather than change it. What about the case where the database must change as well because of a new feature? Refer back to the discussion on evolutionary data design in Chapter 5.

CHAPTER 7

# Evolutionary Architecture Pitfalls and Antipatterns

We've spent a lof of time discussing appropriate levels of coupling in architectures. However, we also live in the real world, and see lots of coupling that *harms* a project's ability to evolve.

We identify two kinds of bad engineering practices that manifest in software projects —*pitfalls* and *antipatterns*. Many developers use the word *antipattern* as jargon for "bad," but the real meaning is more subtle. A software antipattern has two parts. First, an antipattern is a practice that initially looks like a good idea, but turns out to be a mistake. Second, better alternatives exist for most antipatterns. Architects notice many antipatterns only in hindsight, so they are hard to avoid. A *pitfall* looks superficially like a good idea but immediately reveals itself to be a bad path. We cover both pitfalls and antipatterns in this chapter.

## Technical Architecture

In this section, we focus on common practices in the industry that specifically harm a team's ability to evolve the architecture.

### Antipattern: Vendor King

Some large enterprises buy Enterprise Resource Planning (ERP) software to handle common business tasks like accounting, inventory management, and other common chores. This works if companies are willing to bend their business processes and other decisions to accommodate the tool, and can be used strategically when architects understand limitations as well as benefits.

However, many organizations become overambitious with this category of software, leading to the *vendor king antipattern*, an architecture built entirely around a vendor product that pathologically couples the organization to a tool. Companies who buy vendor software plan to augment the package via its plug-ins to flesh out the core functionality to match their business. However, a lot of the time ERP tools can't be customized enough to fully implement what is needed, and developers find themselves hamstrung by the limitations of the tool *and* the fact that they have centered the architectural universe around it. In other words, architects have made the vendor the king of the architecture, dictating future decisions.

To escape this antipattern, treat all software as just another integration point, even if it initially has broad responsibilities. By assuming integration at the outset, developers can more easily replace behavior that isn't useful with other integration points, dethroning the king.

By placing an external tool or framework at the heart of the architecture, developers severely restrict their ability to evolve in two key ways, both technically and from a business process standpoint. Developers are technically constrained by choices the vendor makes in terms of persistence, supported infrastructure, and a host of other constraints. From a business standpoint, large encapsulating tools ultimately suffer from the "Antipattern: Last 10% Trap" on page 127. From a business process standpoint, the tool simply can't support the optimal workflow; this is a side effect of the Last 10% Trap. Most companies end up knuckling under the framework, modifying their processes rather than trying to customize the tool. The more companies do that, the less differentiators exist between companies, which is fine as long as that differentiation isn't a competitive advantage.

The *Let's Stop Working and Call It A Success* principle is one developers commonly encounter when dealing with ERP packages in the real world. Because they require huge investments of both time and money, companies are reluctant to admit when they don't work. No CTO wants to admit they wasted millions of dollars, and the tool vendor doesn't want to admit to a bad multiyear implementation. Thus, each side agrees to stop working and call it a success, with much of the promised functionality unimplemented.

Don't couple your architecture to a vendor king.

Rather than fall victim to the vendor king antipattern, treat vendor products as just another integration point. Developers can insulate vendor tool changes from impacting their architecture by building anticorruption layers between integration points.

## Pitfall: Leaky Abstractions

> All non-trivial abstractions, to some degree, are leaky.
>
> —Joel Spolsky

Modern software resides on a tower of abstractions: operating systems, frameworks, dependencies, and a host of other pieces. As developers, we build abstractions so that we don't have to perpetually think at the lowest levels. If developers were required to translate the binary digits that come from hard drives into text to program, they would never get anything done! One of the triumphs of modern software is how well we can build effective abstractions.

But abstractions come at a cost because no abstraction is perfect—if it was, it wouldn't be an abstraction, it would be the real thing. As Joel Spolsky put it, all non-trivial abstractions leak. This is a problem for developers because we come to trust that abstractions are always accurate, but they often break in surprising ways.

Increased tech stack complexity has made the abstraction distraction problem worse recently. Consider the typical technology stack, circa 2005, shown in Figure 7-1.

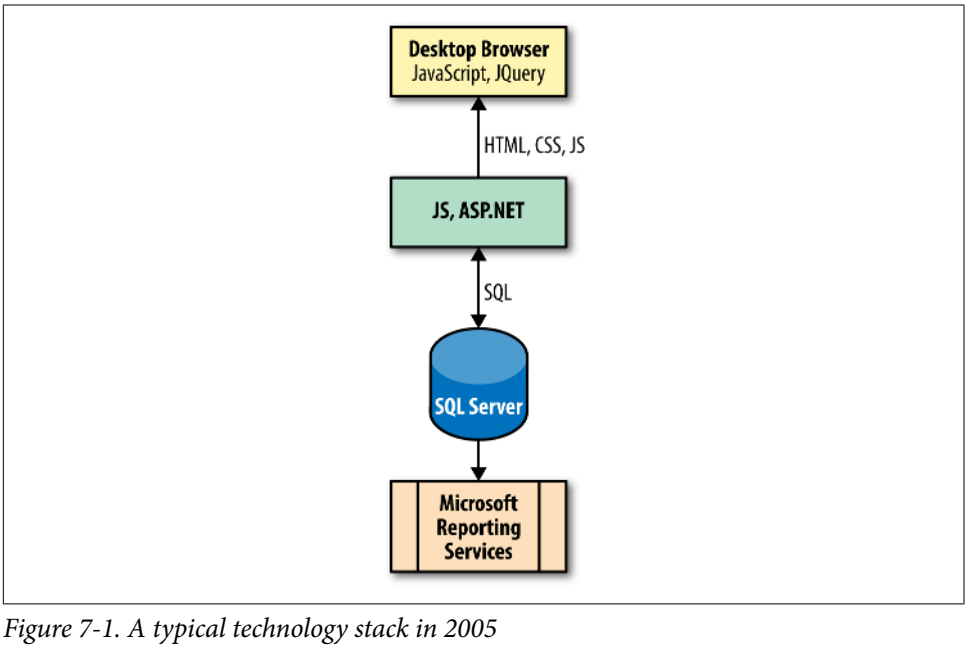



*Figure 7-1. A typical technology stack in 2005*

Figure 7-1 represents a typical software stack in 2005, where the vendor names on the boxes change depending on local conditions. Over time, as software has increasingly specialized, our technology stack has become more complex, as illustrated in Figure 7-2.

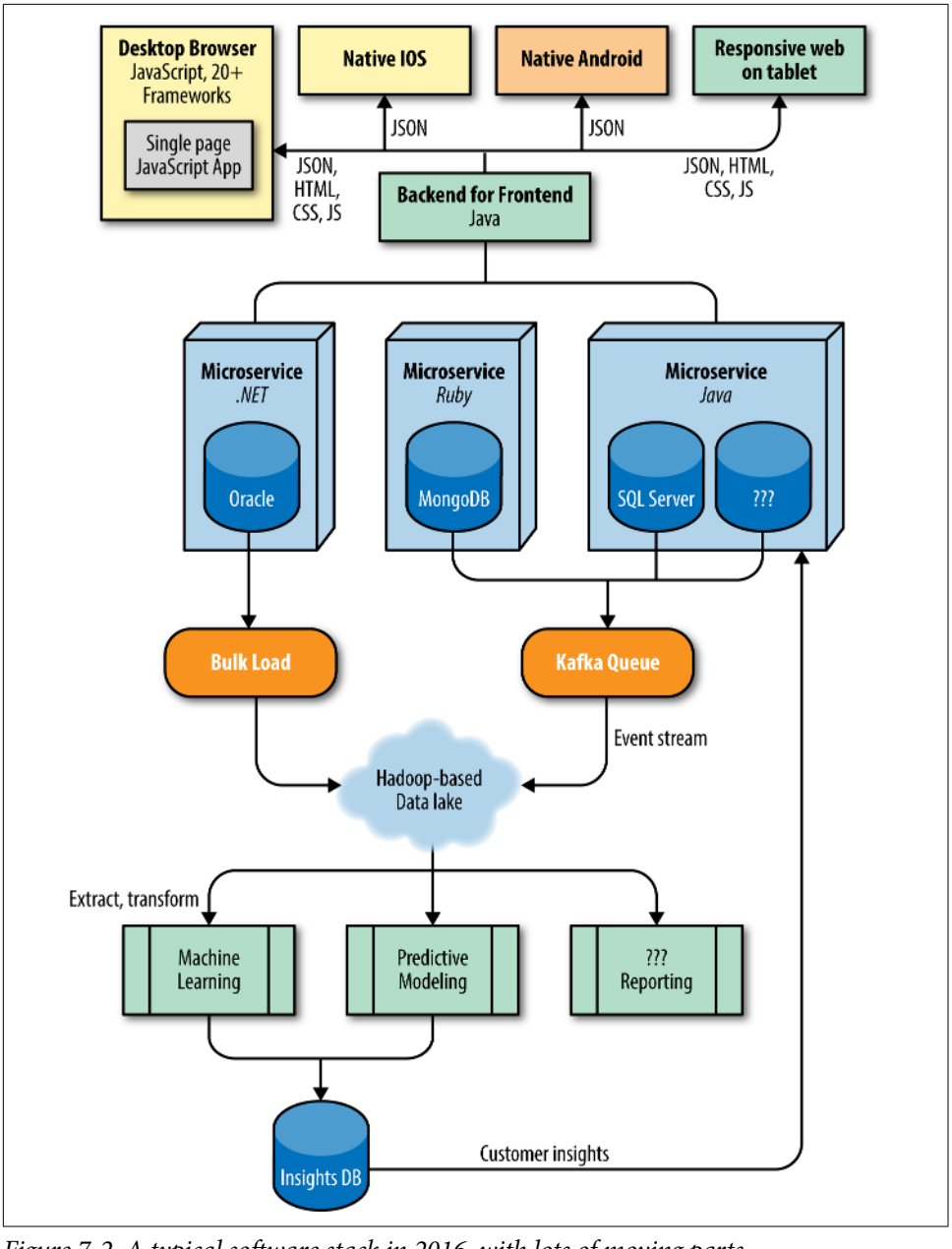



*Figure 7-2. A typical software stack in 2016, with lots of moving parts*

As seen in Figure 7-2, every part of the software ecosystem has expanded and become more complex. As the problems developers face have become more complex, so have their solutions.

*Primordial abstraction ooze*, where a breaking abstraction at a low level causes unexpected havoc, is one of the side effects of increasing complexity in the technology stack. What if one of the abstractions at the lowest level exhibits a fault—for example, some unexpected side effect from a seemingly harmless call to the database? Because so many layers exist, the fault will wind its way to the top of the stack, perhaps metastasizing along the way, manifesting in a deeply embedded error message at the UI. Debugging and forensic analysis becomes more difficult the more complex the technology stack.

> Always fully understand at least one abstraction layer below the one you normally work in.
>
> —Many software sages

While understanding the layer below is good advice, this becomes more difficult as the software becomes more specialized and therefore more complex.

Increased technology stack complexity is an example of the dynamic equilibrium problem. Not only does the ecosystem change, but the constituent parts become more complex and intertwined over time as well. Our mechanism for protecting evolutionary change—fitness functions—can protect the fragile join points of architecture. Architects define invariants at key integration points as fitness functions, which run as part of a deployment pipeline, ensuring abstractions don't start to leak in undesirable ways.

Understand the fragile places within your complex technology stack and automate protections via fitness functions.

## Antipattern: Last 10% Trap

Another kind of reusability trap exists at the other end of the abstraction spectrum, with package software, platforms, and frameworks.

Neal once was the CTO of a consulting firm that built projects for clients in a variety of 4GLs, including Microsoft Access. He eventually assisted in the decision to eliminate Access and eventually all the 4GLs from the business after observing that every Access project started as a booming success but ended in failure, and he wanted to understand why. He and a colleague observed that, in Access and other 4GLs popular at the time, 80% of what the client wanted was quick and easy to build. These environments were modeled as rapid application development tools, with drag-and-drop support for UIs and other niceties. However, the next 10% of what the client wanted was, while possible, extremely difficult—because that functionality wasn't built into the tool, framework, or language. So clever developers figured out a way to hack tools

to make things work: adding a script to execute where static things were expected, chaining methods, and other hacks. The hack only gets you from 80% to 90%. Ultimately the tool can't solve the problem completely—a phrase we coined as the *Last 10% Trap*—leaving every project a disappointment. While 4GLs made it easy to build simple things fast, they didn't scale to meet the demands of the real world. Developers returned to general purpose languages.

### The IBM San Francisco Project

In the late 1990s, IBM embarked on an ambitious plan to write the last piece of business software. A team of developers embarked on the designing of a set of reusable business components, written in that generation's Java Enterprise flavor, that would encapsulate all business functionality in broad categories: ledger, inventory, sales, etc. At one point, IBM claimed that this project constituted the largest Java project on earth. The project delivered the first few core modules, and developers started using the framework, which lead to its demise. Many features were superfluous, and many critical features were absent.

The San Francisco Project illustrates the ultimate hubris of architects and developers who try to follow their inherent instinct to categorize and taxonimize everything. Some messy real-world things defy neat solutions, including all business processes!

The San Francisco Project ultimately failed because they gradually realized a sobering fact—no matter how hard developers try, they can never distill everything to granular enough properties, part of the *infinite regress* problem: a series of propositions that continue to rely on other propositions to be true, into infinity. In software, infinite regress manifests as trying to specify anything in the ultimate level of detail—there is always another layer of granularity below any existing detail.

## Antipattern: Code Reuse Abuse

As an industry, we have benefited greatly from reusable frameworks and libraries built by others, often open source and freely available. Clearly, the ability to reuse code is good. However, like all good ideas, many companies abuse this idea and create problems for themselves. Every corporation desires code reuse because software seems so modular, like electronics components. However, despite the promise that exists for truly modular software, it has consistently evaded us.

> Software reuse is more like an organ transplant than snapping together Lego blocks.
>
> —John D. Cook

While language designers have promised developers Lego blocks for a long time, we still seem to have organs. Software reuse is difficult and doesn't come automatically. Many optimistic managers assume any code that developers write is inherently reusa-

ble, but this is not always the case. Many companies have attempted and succeeded in writing truly reusable code, but it is intentional and difficult. Developers often spend a lot of time trying to build reusable modules that turn out to have little practical reuse.

In service-oriented architectures, the common practice was to find commonalities and reuse as much as possible. For example, imagine that a company has two contexts: `Checkout` and `Shipping`. In an SOA, architects observe that both contexts include the concept of `Customer`. This in turn encouraged them to consolidate both customers into a single `Customer` service, coupling both `Checkout` and `Shipping` to the shared service. Architects worked towards a goal of ultimate *canonicality* in SOA—everything concept has a single (shared) home.

Ironically, the more effort developers put into making code reusable the harder it is to use. Making code reusable involves adding additional options and decision points to accommodate the different uses. The more developers add hooks to enable reusability the more they harm the basic *usability* of the code.

The more reusable code is, the less usable it is.

In other words, ease of code use is often inversely proportional to how reusable that code is. When developers build code to be reusable, they must add features to accomodate the myriad ways developers will will eventually use the code. All that future-proofing makes it more difficult for developers to use the code for a single purpose.

Microservices eschew code reuse, adopting the philosophy of *prefer duplication to coupling*: reuse implies coupling, and microservices architectures are extremely decoupled. However, the goal in microservices isn't to embrace duplication but rather to isolate entities within domains. Services that share a common class are no longer independent. In a microservices architecture, `Checkout` and `Shipping` would each have their own internal representation of `Customer`. If they need to collaborate on customer-related information, they send the pertinent information to each other. Architects don't try to reconcile and consolidate the disparate versions of `Customer` in their architecture. The benefits of reuse are illusory and the coupling it introduces comes with its disadvantages. Thus, while architects understand the downsides of duplication, they offset that localized damage to the architectural damage too much coupling introduces.

Code reuse can be an asset but also a potential liability. Make sure the coupling points introduced in your code don't conflict with other goals in the architecture. For example, microservices architectures typically use service templates (covered in "Case

Study: Service Templates" on page 113) to couple the parts of services together that help unify a particular architectural concern, such as monitoring or logging.

## Case Study: Reuse at PenultimateWidgets

PenultimateWidgets has highly specific requirements for data input in a specialized grid for their administration functionality. Because the application required this view in multiple places, PenultimateWidgets decided to build a reusable component, including UI, validation, and other useful default behaviors. By using this component, developers can build new, rich administration interfaces easily.

However, virtually no architecture decision comes without some tradeoff baggage. Over time, the component team has become their own silo within the organization, tying up several of PenultimateWidgets' best developers. Teams that use the component must request new features through the component team, which is swamped with bug fixes and feature requests. Worse, the underlying code hasn't kept up with modern web standards, making new functionality hard or impossible.

While the PenultimateWidgets architects achieved reuse, it eventually resulted in a bottleneck effect. One advantage of reuse is that developers can build new things quickly. Yet, unless the component team can keep up with the innovation pace of the dynamic equilibrium, technical architecture component reuse is doomed to eventually become an antipattern.

We're not suggesting teams avoid building reusable assets, but rather evaluate them continually to ensure they still deliver value. In the case of PenultimateWidgets, once architects realized that the component was a bottleneck, they broke the coupling point. Any team that wants to fork the component code to add their own new features is allowed (as long as the application development team supports the changes), and any team that wants to opt out to use a new approach is unshackled from the old code entirely.

Two pieces of advice emerge from PenultimateWidgets experience:

When coupling points impede evolution or other importance architectural characteristics, break the coupling by forking or duplication.

In PenultimateWidgets' case, they broke the coupling by allowing teams to take ownership of the shared code themselves. While adding to their burden, it released the drag on their ability to deliver new features. In other cases, perhaps some shared code can be abstracted from the larger piece, allowing more selective coupling and gradual decoupling.

Architects must continually evaluate the fitness of the "-ilities" of the architecture to ensure they still add value and haven't become antipatterns.

All too often architects make a decision that is the correct decision at the time but becomes a bad decision over time because of changing conditions like dynamic equilibrium. For example, architects design a system as a desktop application, yet the industry herds them toward a web application as users' habits change. The original decision wasn't incorrect, but the ecosystem shifted in unexpected ways.

### Pitfall: Resume-Driven Development

Architects become enamored of exciting new developments in the software development ecosystem and want to play with the newest toys. However, to choose an effective architecture, they must look closely at the problem domain and choose the most suitable architecture that delivers the most desired capabilities with the fewest damaging constraints. Unless, of course, the goal of the architecture is the *Resume-Driven Development* pitfall—utilizing every framework and library possible to tout that knowledge on a resume.

Don't build architecture for the sake of architecture—you are trying to solve a problem.

Always understand the problem domain before choosing an architecture rather than the other way around.

## Incremental Change

Many factors in software development make incremental change difficult. For many decades, software wasn't written with the goal of agility in mind but rather around goals like cost reduction, shared resources, and other external constraints. Consequently, many organizations don't have the building blocks in place to support evolutionary architectures.

As discussed in the *Continuous Delivery* book, many modern engineering practices support evolutionary architecture.

## Antipattern: Inappropriate Governance

Software architecture never exists in a vacuum; it is often a reflection of the environment in which it was designed. A decade ago, operating systems were expensive, commercial offerings. Similarly, database servers, application servers, and the entire infrastructure for hosting applications was commercial and expensive. Architects responded to these real-world pressures by designing architectures to maximize shared resources. Many architecture patterns like SOA flourished in that era. A common governance model evolved in that environment to maximize shared resources as a cost-saving measure. Many of the commercial motivations for tools like application servers grew from this tendency. However, packing multiple resources on machines is undesirable from a development standpoint because of inadvertent coupling. No matter how good the isolation between shared resources, resource contention eventually rears its head.

Over the last decade, changes have occurred to the dynamic equilibrium of the development ecosystem. Now, developers can build architectures where components have a high degree of isolation (like microservices), eliminating the accidental coupling exacerbated by shared environments. But many companies still adhere to the old governance playbook. A governance model that values shared resources and homogenized environments makes less sense because of recent improvements such as the DevOps movement.

> Every company is now a software company.
>
> —Forbes Magazine, *Nov. 30, 2011*

What Forbes means in their famous quote is that if an airline company's iPad application is terrible, it will eventually impact the company's bottom line. Software competency is required for any cutting edge company, and increasingly for any company who wishes to remain competitive. Part of that competency includes how they manage development assets like environments.

When developers can create resources like virtual machines and containers for no cost (either monetary or time), a governance model that values a single solution becomes *innappropriate governance*. A better approach appears in many microservices environments. One common characteristic of microservices architectures is the embrace of polyglot environments, where each service team can choose a suitable technology stack to implement their service rather than try to homogenize on a corporate standard. Traditional enterprise architects cringe when they hear that advice, which is polar opposite of the traditional approach. However, the goal in most microservices projects isn't to pick different technologies cavalierly, but rather to right-size the technology choice for the size of the problem.

In modern environments, it is inappropriate governance to homogenize on a single technology stack. This leads to the inadvertant overcomplication problem, where

governance decisions add useless multipliers to the effort required to implement a solution. For example, standardizing on a single vendor's relational database is a common practice in large enterprises, for obvious reasons: consistency across projects, easily fungible staff, and so on. However, a side effect of that approach is that most projects suffer from overengineering. When developers build monolith architectures, governance choices affect everyone. Thus, when choosing a database, the architect must look at the requirements of every project that will use this capability, and make a choice that will serve the most complex case. Unfortunately, many projects won't have the most complex case or anything like it. A small project may have simple persistence needs yet must take on the full complexity of an industrial strength database server for consistency.

With microservices, because none of the services are coupled via technical or data architecture, different teams can choose the right level of complexity and sophistication required to implement their service. The ultimate goal is simplification, to align service stack complex to technical requirements. This partitioning tends to work best when the team wholly owns their service, including the operational aspects.

### Forced Decoupling

One of the goals of the microservices architecture style is extreme decoupling of the technical architecture, allowing services to be replaced with no side effects. However, if developers all share the same code base or even platform, *not* coupling requires some degree of developer discipline (because the temptation to reuse existing code is strong) and safeguards to make sure coupling doesn't happen by accident. Building services in different technology stacks is one way to achieve technical architecture decoupling. Many companies try to avoid this approach because they fear it hurts the ability to move employees across projects. However, Chad Fowler, an architect at Wunderlist, took the opposite approach: he *insisted* that teams use different technology stacks to avoid inadvertent coupling. His philosophy is that accidental coupling is a bigger problem than developer portability.

Many companies are encapsulating distinct functionality into a Platform as a Service for use internally, hiding technology choices (and therefore coupling opportunities) behind well-defined interfaces.

From a practical governance standpoint in large organizations, we find the *Goldilocks Governance* model works well: pick three technology stacks for standardization—simple, intermediate, and complex—and allow individual service requirements to drive stack requirements. This gives teams the flexibility to choose a suitable technology stack while still providing the company some benefits of standards.

## Case Study: Goldilocks Governance at PenultimateWidgets

For years, architects at PenultimateWidgets tried to standardize all development on Java and Oracle. However, as they built more granular services, they realized that this stack imposed a great deal of complexity on small services. But they didn't want to fully embrace the "every project chooses their own technology stack" approach of microservices because they still wanted some portability of knowledge and skills across projects. In the end, they chose the Goldilocks Governance route with three technology stacks:

*Small*
: For very simple projects without stringent scalability or performance requirements, they chose Ruby on Rails and MySQL.

*Medium*
: For medium projects, they chose GoLang and one of Cassandra, MongoDB, or MySQL as the backend, depending on the data requirements.

*Large*
: For large projects, they stayed with Java and Oracle, as they work well with variable architecture concerns.

## Pitfall: Lack of Speed to Release

The engineering practices in continuous delivery address the factors that slow down software releases, and those practices should be considered axiomatic for evolutionary architecture to be successful. While the extreme version of Continuous Delivery, continuous deployment, isn't required for an evolutionary architecture, a strong correlation exists between the ability to release software and the ability to evolve that software design.

If companies build an engineering culture around continuous deployment, expecting that all changes will make their way to production only if they pass the gauntlet laid out by the deployment pipeline, developers become accustomed to constant change. On the other hand, if releases are a formal process that require a lot of specialized work, the chances of being able to leverage evolutionary architecture diminishes.

Continuous Delivery strives for data-driven results, employing metrics to learn how to optimize projects. Developers must be able to measure things to understand how to make them better. One of the key metrics Continuous Delivery tracks is *cycle time*, a metric related to *lead time*: the time between the initiation of an idea and that idea manifesting in working software. However, lead time includes many subjective activities, such as estimation, prioritization, and others, making it a poor engineering metric. Instead, Continuous Delivery tracks *cycle time*: the elapsed time between the initiation and completion of a unit of work, which in this case is software develop-

ment. The cycle time clock starts when a developer starts working on a new feature and expires when that feature is running in a production environment. The goal of cycle time is to measure engineering efficiency; the reduction of cycle time is one of the key goals of Continuous Delivery.

Cycle time is critical for evolutionary architecture as well. In biology, fruit flies are commonly used in experiments to illustrate genetic characteristics partially because they have a rapid life cycle—new generations appear fast enough to see tangible results. The same is true in evolutionary architecture—faster cycle time means the architecture can evolve more quickly. Thus, a project's cycle time determines how fast the architecture can evolve. In other words, evolution speed is proportional to cycle time, as expressed by

$$v \propto c$$

where v represents velocity of change and c is cycle time. Developers cannot evolve the system faster than the project's cycle time. In other words, the faster teams can release software, the faster they can evolve parts of their system.

Cycle time is therefore a critical metric in evolutionary architecture projects—faster cycle time implies a faster ability to evolve. In fact, cycle time is an excellent candidate for an atomic, process-based fitness function. For example, developers set up a project with a deployment pipeline with automation, achieving a cycle time of three hours. Over time, the cycle time gradually increases as developers add more verifications and integration points to the deployment pipeline. Because time to market is an important metric on this project, they establish a fitness function to raise an alarm if the cycle time creeps beyond four hours. Once it has hit the threshold, developers may decide to restructure how their deployment pipeline works or decide that a four hour cycle time is acceptable. Fitness functions can map to any behavior developers want to monitor on projects, including project metrics. Unifying project concerns as fitness functions allows developers to set up future decision points, also known as the *last responsible moment*, to reevaluate decisions. In the previous example, developers now must decide which is more important: three hour cycle time or the set of tests they have in place. On most projects, developers make this decision implicitly by never noticing a gradually rising cycle time and thus never prioritizing conflicting goals. With fitness functions, they can install thresholds around anticipated future decision points.

Speed of evolution is a function of cycle time; faster cycle time allows faster evolution.

Good engineering, deployment, and release practices are critical to success with an evolutionary architecture, which in turn allows new capabilities for the business via hypothesis-driven development.

# Business Concerns

Finally, we talk about inappropriate coupling driven by business concerns. Most of the time, business people aren't nefarious characters trying to make things difficult for developers, but rather have priorities that drive inappropriate decisions from an architectural standpoint, which inadvertently constrain future options. We cover a handful of business pitfalls and antipatterns.

## Pitfall: Product Customization

Salespeople want options to sell. The caricature of sales people has them selling any requested feature before determining if their product actually contains that feature. Thus, sales people want infinitely customizable software to sell. However, that capability comes at a cost along a spectrum of implementation techniques.

*Unique build for each customer*
: In this scenario, salespeople promise unique versions of features on a tight time scale, forcing developers to use techniques like version control branches and tagging to track versions.

*Permanent feature toggles*
: We introduced feature toggles in Chapter 3, which are sometimes used strategically to create permanent customizations. Developers can use feature toggles to build either different versions for different clients or to create a "freemium" version of a product—a free version that allows users to unlock premium features for a cost.

*Product-driven customization*
: Some products go so far as to add customization via the UI. Features in this case are permanent parts of the application and require the same care as all other product features.

With both feature toggles and customization, the testing burden increases significantly because the product contains many permutations of possible pathways. Along with testing scenarios, the number of fitness functions developers need to develop likely increases as well, to protect possible permutations.

Customization also impedes evolvability, but this shouldn't discourage companies from building customizable software, but rather to realistically assess the associated costs.

## Antipattern: Reporting

Most applications have different uses depending on the business function. For example, some users need order entry, while others require reports for analysis. Organizations struggle to provide all the possible perspectives (e.g., order entry versus monthly reporting) required by businesses, especially if everything must come from the same monolithic architecture and/or database structure. Architects struggled in the service-oriented architecture era trying to support every business concern via the same set of "reusable" services. They found that the more generic the service, the more developers needed to customize it to be of use.

Reporting is a good example of inadvertent coupling in monolithic architectures. Architects and DBAs want to use the same database schema for both system of record and reporting, but encounter problems because a design to support both is optimized for neither. A common pitfall developers and report designers conspire to create in layered architecture illustrates the tension between concerns. Architects build layered architecture to cut down on incidental coupling, creating layers of isolation and separation of concerns. However, reporting doesn't need separate layers to support its function, just data. Additionally, routing requests through layers adds latency. Thus, many organizations with good layered architectures allow report designers to couple reports directly to database schemas, destroying the ability to make changes to the schema without wrecking reports. This is a good example of conflicting business goals subverting the work of architects and making evolutionary change extremely difficult. While no one set out to make the system hard to evolve, it was the cumulative effect of decisions.

Many microservices architectures solve the reporting problem by separating behavior, where the isolation of services benefits separation but not consolidation. Architects commonly build these architectures using event streaming or message queues to populate domain "system of record" databases, each embedded within the architectural quantum of the service, using eventual consistency rather than transactional behavior. A set of reporting services also listens to the event stream, populating a denormalized reporting database optimized for reporting. Using eventual consistency frees architects from coordination—a form of coupling from an architectural standpoint—allowing different abstractions for different uses of the application.

For example, in PenultimateWidgets' microservices architecture, they have domains separated into bounded contexts, each owning the "system of record" data for that domain. Developers at PenultimateWidgets use eventual consistency and message queues to populate and communicate, and have a set of reporting services, separate from the domain services, as shown in Figure 7-3.

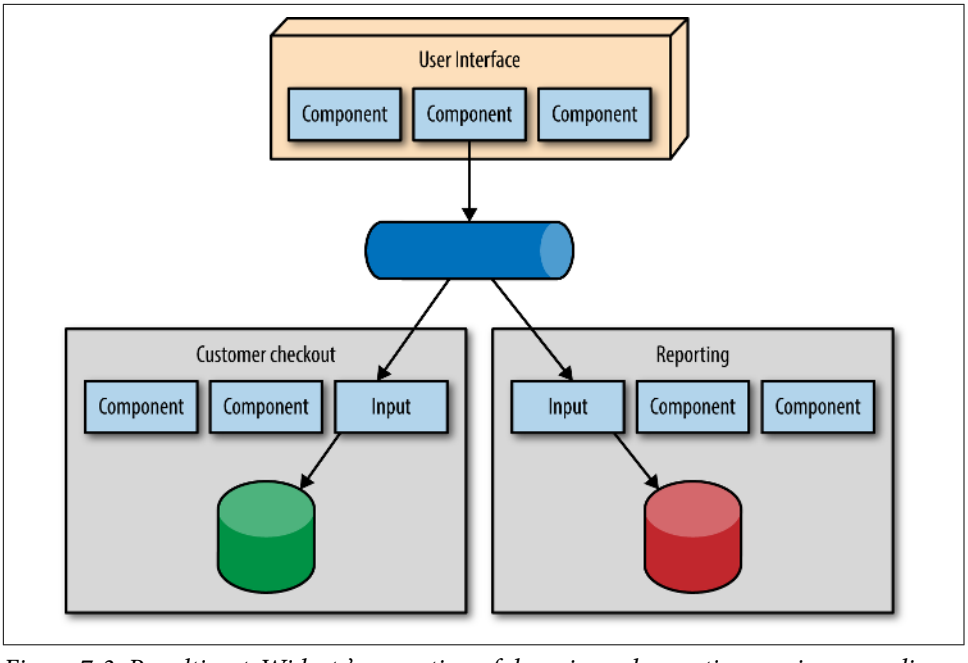



*Figure 7-3. PenultimateWidgets' separation of domain and reporting services, coordinated via message queues*

As seen in Figure 7-3, when the UI reports a Create, Read, Update, Define (CRUD) operation, both the domain microservice and reporting service listen to the notification and take appropriate action. Thus, the set of reporting services handles reporting concerns without affecting the domain services. Removing the inappropriate coupling introduced by conflating domains and reporting allows each team to focus on more specific yet simpler tasks.

## Pitfall: Planning Horizons

Budgeting and planning processes often drive the need for assumptions and early decisions as the basis for those assumptions. However, the larger the planning horizon without an opportunity to revisit the plan means many decisions (or assumptions) are made with the least amount of information. In the early planning phases, developers spend significant effort on activities like research, often in the form of reading, to validate their assumptions. Based on their studies, what is "best practice" or "best in class" at that time form part of the basic fundamental assumptions before developers write any code or release software to end users. More and more effort put into the assumptions, even if they turn out to be false in six months, leads to a strong attachment to them. The Sunk Cost Fallacy describes decisions affected by emotional investment. Put simply, the more someone invests time or effort into something, the

harder it becomes to abandon it. In software, this is seen in the form of the *irrational artifact attachment*—the more time and effort you invest in planning or a document, the more likely you will protect what's contained in the plan or document even in the face of evidence that it is inaccurate or outdated.

Don't become irrationally attached to handcrafted artifacts.

Beware of long planning cycles that force architects into irreversible decisions and find ways to keep options open. Breaking large programs of work into smaller, early deliverables tests the feasibility of both the architectural choices and the development infrastructure. Architects should avoid following technologies that require a significant upfront investment before software is actually built (e.g., large licenses and support contracts) before they have validated through end-user feedback that the technology actually fits the problem they are trying to solve.

CHAPTER 8

# Putting Evolutionary Architecture into Practice

Finally, we look at the steps required to implement the ideas around evolutionary architecture. This includes both technical and business concerns, including organization and team impacts. We also suggest where to start and how to sell these ideas to your business.

## Organizational Factors

The impact of software architecture has a surprisingly wide breadth on a variety of factors not normally associated with software, including team impacts, budgeting, and a host of others.

Teams structured around domains rather than technical capabilities have several advantages when it comes to evolutionary architecture and exhibit some common characteristics.

### Cross-Functional Teams

Domain-centric teams tend to be *cross-functional*, meaning every project role is covered by someone on the project. The goal of a domain-centric team is to eliminate operational friction. In other words, the team has all the roles needed to design, implement, and deploy their service, including traditionally separate roles like operations. But these roles must change to accommodate this new structure, which includes the following roles:

*Business Analysts*
: Must coordinate the goals of this service with other services, including other service teams.

*Architecture*
: Design architecture to eliminate inappropriate coupling that complicates incremental change. Notice this doesn't require an exotic architecture like microservices. A well-designed modular monolithic application may display the same ability to accommodate incremental change (although architects must design the application explicitly to support this level of change).

*Testing*
: Testers must become accustomed to the challenges of integration testing across domains, such as building integration environments, creating and maintaining contracts, and so on.

*Operations*
: Slicing up services and deploying them separately (often alongside existing services and deployed continuously) is a daunting challenge for many organizations with traditional IT structures. Naive old school architects believe that component and operational modularity are the same thing, but this is often not the case in the real world. Automating DevOps tasks like machine provisioning and deployment are critical to success.

*Data*
: Database administrators must deal with new granularity, transaction, and system of record issues.

One goal of cross-functional teams is to eliminate coordination friction. On traditional siloed teams, developers often must wait on a DBA to make changes or wait for someone in operations to provide resources. Making all the roles local eliminates the incidental friction of coordination across silos.

While it would be luxurious to have every role filled by qualified engineers on every project, most companies aren't that lucky. Key skill areas are always constrained by external forces like market demand. So, many companies aspire to create cross-functional teams but cannot because of resources. In those cases, constrained resources may be shared across projects. For example, rather than have one operations engineer per service, perhaps they rotate across several different teams.

By modeling architecture and teams around the domain, the common unit of change is now handled within the same team, reducing artificial friction. A domain-centric architecture may still use layered architecture for its other benefits, such as separation of concerns. For example, the implementation of a particular microservice might depend on a framework that implements the layered architecture, allowing that team to easily swap out a technical layer. Microservices encapsulate the technical architecture inside the domain, inverting the traditional relationship.

### Finding New Resources via Automating DevOps

Neal once consulted for a company that offered a hosted service. They had a dozen development teams, all with well-defined modules. However, they had an operations group who managed all maintenance, provisioning, monitoring, and other common tasks. The manager commonly received complaints from developers who wanted faster turnaround on needed resources like database and web servers. To alleviate some of the pressure, he started assigning an operations person one day a week to each project. During that day, the developers were happy as can be—no waiting around for resources! Alas, the manager didn't have enough resources to do that regularly.

Or so he thought. We discerned that much of the manual work performed by operations was accidental complexity: misconfigured machines, a hodgepodge of manufacturers and brands, and many other repairable offenses. Once everything was well cataloged, we helped them automate the provisioning of new machines using Puppet. After this work, the operations team had enough members to permanently embed an operations engineer on each project and still have enough people to manage the automated infrastructure.

They didn't hire new engineers, nor did they significantly change their job roles. Instead, they applied modern engineering practices to automate things that humans shouldn't deal with on a regular basis, freeing them to be better partners in development efforts.

## Organized Around Business Capabilities

Organizing teams around domains implicitly means organizing them around business capabilities. Many organizations expect their technical architecture to represent its own complex abstraction, loosely related to business behavior because architect's traditional emphasis has been around purely technical architecture, that is typically segregated by functionality. For example, a layered architecture is designed to make swapping technical architecture layers easier, not make working on a domain entity like `Customer` easier. Most of this emphasis was driven by external factors. For example, many architectural styles of the past decade focused heavily on maximizing shared resources because of expense.

Architects have gradually detangled themselves from commercial restrictions via the embrace of open source in all corners of most organizations. Shared resource architecture has inherent problems around inadvertent interference between parts. Now that developers have the option of creating custom-made environments and functionality, it is easier for them to shift emphasis away from technical architectures and

focus more on domain-centric ones to better match the common unit of change in most software projects.

Organize teams around business capabilities, not job functions.

## Product over Project

One mechanism many companies use to shift their team emphasis is to model their work around *products* rather than *projects*. Software projects have a common workflow in most organizations. A problem is identified, a development team is formed, and they work on the problem until "completion," at which time they turn the software over to operations for care, feeding, and maintenance for the rest of its life. Then the project team moves on to the next problem.

This causes a slew of common problems. First, because the team has moved on to other concerns, bug fixes and other maintenance work is often difficult to manage. Second, because the developers are isolated from the operational aspects of their code, they care less about things like quality. In general, the more layers of indirection between a developer and their running code, the less connection they have to that code. This sometimes leads to an "us versus them" mentality between operational silos, which isn't surprising, as many organizations have incentivized workers to exist in conflict.

By thinking of software as a *product*, it shifts the company's perspective in three ways. First, products live forever, unlike the lifespan of projects. Cross-functional teams (frequently based on the Inverse Conway Maneuver) stay associated with their product. Second, each product has an owner who advocates for its use within the ecosystem and manages things like requirements. Third, because the team is cross-functional, each role needed by the product is represented: business analyst, developers, QA, DBA, operations, and any other required roles.

The real goal of shifting from a *project* to a *product* mentality concerns long-term company buy-in. Product teams take ownership responsibility for the long-term quality of their product. Thus, developers take ownership of quality metrics and pay more attention to defects. This perspective also helps provide a long-term vision to the team.

### Amazon's "Two Pizza" Teams

Amazon became famous for their product team approach, which they called *two-pizza teams*. Their philosophy is that no team shall be larger than can be fed with two large pizzas. The motivation behind this partitioning is more about communication than team size—the larger the team, the more people each team member must communicate with. Each team is cross-functional, and they also embrace the philosophy of "you build it, you run it," meaning each team has complete ownership of their service, including operationalizing it.

Having small, cross-functional teams also takes andvantage of human nature. Amazon's "two-pizza team" mimics small group primate behavior. Most sports teams have around 10 players, and anthropologists believe that preverbal hunting parties were also around this size. Building highly responsible teams leverages innate social behavior, making team members more responsible. For example, suppose a developer in a traditional project structure wrote some code two years ago that blew up in the middle of the night, forcing someone in operations to respond to a pager in the night and fix it. The next morning, our careless developer may not even realize they accidentally caused a panic in the middle of the night. On a cross-functional team, if the developer wrote code that blew up in the night and someone from his team had to respond to it, the next morning, our hapless developer has to look across the table at the sad, tired eyes of their team member they inadvertently affected. It should make our errant developer want to be a better teammate.

Creating cross-functional teams prevents finger pointing across silos and engenders a feeling of ownership in the team, encouraging team members to do their best work.

## Dealing with External Change

We advocate building components that are highly decoupled in terms of technical architecture, team structure, and so on to allow maximum opportunities for evolution, in the real world, components must interact with one another to share information that collaboratively solves domain problems. So how can we build components that can freely evolve yet make sure we can maintain the integrity of our integration points?

For any dimension in our architecture that requires protection from the side effects of evolution, we create fitness functions. A common practice in microservices architectures is the use of consumer-driven contracts, which are atomic integration architecture fitness functions. Consider the illustration shown in Figure 8-1.

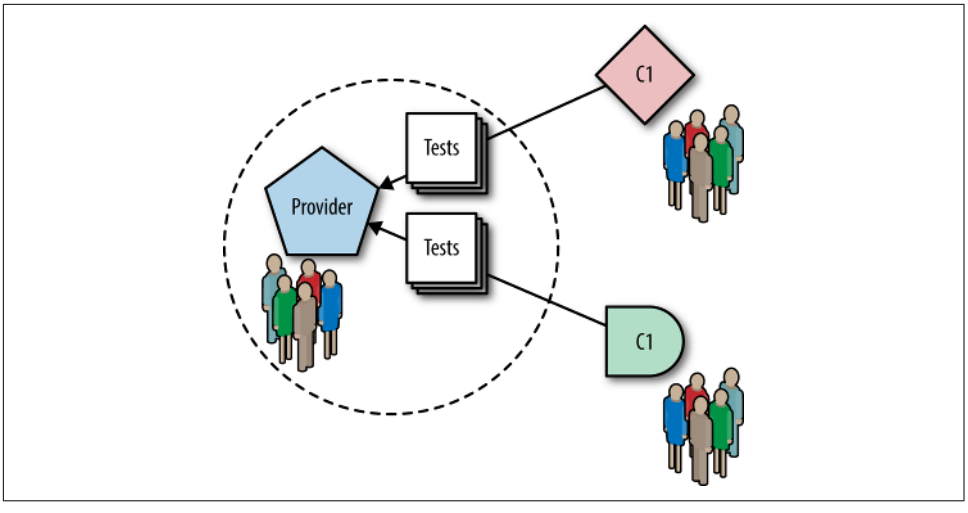



*Figure 8-1. Consumer-driven contracts use tests to establish contracts between a provider and consumer(s)*

In Figure 8-1, the *provider* team is supplying information (typically data in a lightweight format) to each of the consumers, *C1* and *C2*. In consumer-driven contracts, the consumers of information put together a suite of tests that encapsulate what they need from the provider and hand off those tests to the provider, who promises to keep those tests passing at all times. Because the tests cover the information needed by the consumer, the provider can evolve in any way that doesn't break these fitness functions. In the scenario shown in Figure 8-1, the *provider* runs tests on behalf of all three consumers in addition to their own suite of tests. Using fitness functions like this is informally known as an *engineering safety net*. Maintaining integration protocol consistency shouldn't be done manually when it is easy to build fitness functions to handle this chore.

One implicit assumption included in the incremental change aspect of evolutionary architecture is a certain level of engineering maturity amongst the development teams. For example, if a team is using consumer-driven contracts but they also have broken builds for days at time, they can't be sure their integration points are still valid. Using engineering practice to police practices via fitness functions relieves lots of manual pain from developers but requires a certain level of maturity to be successful.

## Connections Between Team Members

Many companies have found anecdotally that large development teams don't work well, and J. Richard Hackman, a famous expert on team dynamics, offers an explanation as to why. It's not the number of people but the number of connections they must

maintain. He uses the formula shown in Equation 8-1 to determine how many connections exist between people, where *n* is the number of people.

*Equation 8-1. Number of connections between people*

$$\frac{n(n-1)}{2}$$

In Equation 8-1, as the number of people grows, the number of connections grows rapidly, as shown in Figure 8-2.

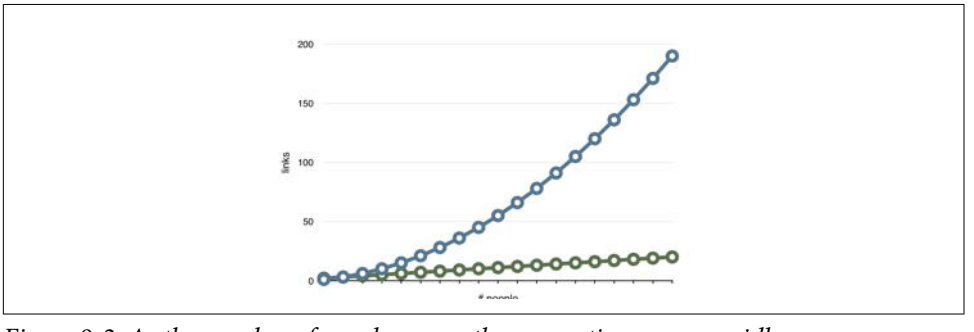



*Figure 8-2. As the number of people grows, the connections grow rapidly.*

In Figure 8-2, when the number of people on a team reaches 20, they must manage 190 links; when it reaches 50 team members, the number of links is a daunting 1225. Thus, the motivation to create small teams revolves around the desire to cut down on communication links. And these small teams should be cross-functional to eliminate artificial friction imposed by coordinating across silos.

Each team shouldn't have to know what other teams are doing, unless integration points exist between the teams. Even then, fitness functions should be used to ensure integrity of integration points.

Strive for a low number of connections between development teams.

## Team Coupling Characteristics

The way firms organize and govern their own structures significantly influences the way that software is built and architected. In this section, we explore the different organizational and team aspects that make building evolutionary architectures easier

or harder. Most architects don't think about how team structure affects the coupling characteristics of the architecture, but it has a huge impact.

## Culture

> **Culture**, (n.): The ideas, customs, and social behavior of a particular people or society.
>
> —Oxford Dictionary

Architects should care about how engineers build their system and watch out for the behaviors their organization rewards. The activities and decision-making processes architects use to choose tools and create designs can have a big impact on how well software endures evolution. Well-functioning architects take on leadership roles, creating the technical culture and designing approaches for how developers build systems. They teach and encourage in individual engineers the skills necessary to build evolutionary architecture.

An architect can seek to understand a team's engineering culture by asking questions like:

- Does everyone on the team know what fitness functions are and consider the impact of new tool or product choices on the ability to evolve new fitness functions?
- Are teams measuring how well their system meets their defined fitness functions?
- Do engineers understand cohesion and coupling?
- Are there conversations about what domain and technical concepts belong together?
- Do teams choose solutions not based on what technology they want to learn, but based on its ability to make changes?
- How are teams responding to business changes? Do they struggle to incorporate small changes, or are they spending too much time on small business change?

Adjusting the behavior of the team often involves adjusting the process around the team, as people respond to what is asked of them to do.

> Tell me how you measure me, and I will tell you how I will behave.
>
> —Dr. Eliyahu M. Goldratt (The Haystack Syndrome)

If a team is unaccustomed to change, an architect can introduce practices that start making that a priority. For example, when a team considers a new library or framework, the architect can ask the team to explicitly evaluate, through a short experiment, how much extra coupling the new library or framework will add. Will engineers be able to easily write and test code outside of the given library or frame-

work, or will the new library and framework require additional runtime setup that may slow down the development loop?

In addition to the selection of new libraries or frameworks, code reviews are a natural place to consider how well newly changed code supports future changes. If there is another place in the system that will suddenly use another external integration point, and that integration point will change, how many places would need to be updated? Of course, developers must watch out for overengineering, prematurely adding additional complexity or abstractions for change. The Refactoring book contains relevant advice:

**Three strikes and you refactor**

The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.

Many teams are driven and rewarded most often for delivering new functionality, with code quality and the evolvable aspect considered only if teams make it a priority. An architect that cares about evolutionary architecture needs to watch out for team actions that prioritize design decisions that help with evolvability or to finds ways to encourage it.

## Culture of Experimentation

Successful evolution demands experimentation, but some companies fail to experiment because they are too busy delivering to plans. Successful experimentation is about running small activities on a regular basis to try out new ideas (both from a technical and product perspective) and to integrate successful experiments into existing systems.

> The real measure of success is the number of experiments that can be crowded into 24 hours.
>
> —Thomas Alva Edison

Organizations can encourage experimentation in a variety of ways:

*Bringing ideas from outside*
: Many companies send their employees to conferences and encourage them to find new technologies, tools, and approaches that might solve a problem better. Other companies bring in external advice or consultants as sources of new ideas.

*Encouraging explicit improvement*
: Toyota is most famous for their culture of kaizen, or continuous improvement. Everyone is expected to continually seek constant improvements, particularly those closest to the problems and empowered to solve them.

*Spike and stabilize*
: A spike solution is an extreme programming practice where teams generate a throw-away solution to quickly learn a tough technical problem, explore an unfamiliar domain, or increase confidence in estimates. Using spike solutions increases learning speed at the cost of software quality; no one would want to put a spike solution straight into production because it would lack the necessary thought and time to make it operational. It was created for learning, not as the well engineered solution.

*Creating innovation time*
: Google is well known for their 20% time, where employees can work on any project for 20% of their time. Other companies organize Hackathons and allow teams to find new products or improvements to existing products. Atlassian holds regular 24-hour sessions called ShipIt days.

*Following set-based development*
: Set-based development focuses on exploring multiple approaches. At first glance, multiple options appear costly because of extra work, but in exploring several options simultaneously, teams end up with a better understanding of the problem at hand and discover real constraints with tooling or approach. The key to effective set-based development is to prototype several approaches in a short time-period (i.e., less than a few days) to build more concrete data and experience. A more robust solution often appears after taking into account several competing solutions.

*Connecting engineers with end-users*
: Experimentation is only successful when teams understand the impact of their work. In many firms with an experimentation mindset, teams and product people see first-hand the impact of decisions on end-customers and are encouraged to experiment to explore this impact. A/B testing is one such practice companies use with this experimentation mindset. Another practice companies implement is sending teams and engineers to observe how users interact with their software to achieve a certain task. This practice, taken from the pages of the usability community, builds empathy with end-users and engineers often return with a better understanding of user needs, and with new ideas to better fulfill them.

## CFO and Budgeting

Many traditional functions of enterprise architecture, such as budgeting, must reflect changing priorities in an evolutionary architecture. In the past, budgeting was based on the ability to predict long-term trends in a software development ecosystem. However, as we've suggested throughout this book, the fundamental nature of dynamic equilibrium destroys predictability.

In fact, an interesting relationship exists between architectural quanta and the cost of architecture. As the number of quanta rises, the cost per quantum goes down, until architects reach a sweet spot, as illustrated in Figure 8-3.

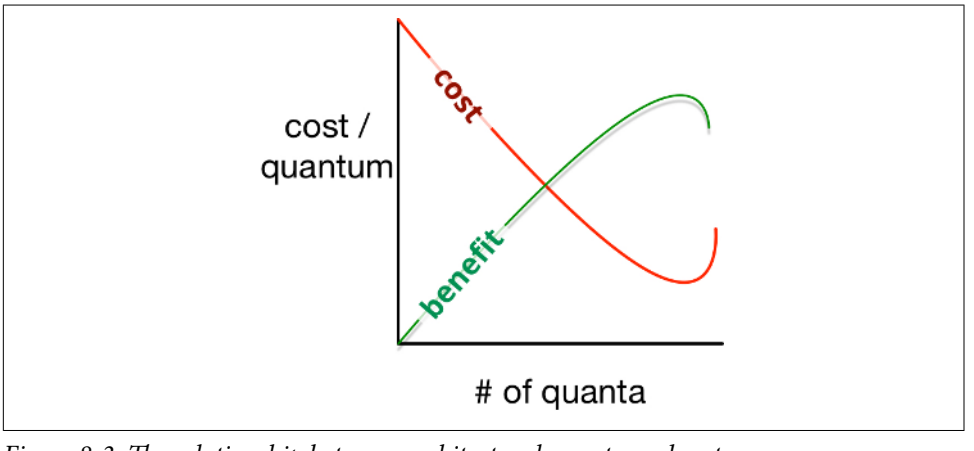



*Figure 8-3. The relationship between architectural quanta and cost*

In Figure 8-3, as the number of architectural quanta rises, the cost of each diminishes because of several factors. First, because the architecture consists of smaller parts, the separation of concerns should be more discrete and defined. Second, rising numbers of physical quanta require automation of their operational aspects because, beyond a certain point, it is no longer practical for people to handle chores manually.

However, it is possible to make quanta so small that the shear numbers become more costly. For example, in a microservices architecture, it is possible to build services at the granularity of a single field on a form. At that level, the coordination cost between each small part starts dominating other factors in the architecture. Thus, at the extremes of the graph, the sheer number of quanta drives benefit per quantum down.

In an evolutionary architecture, architects strive to find the sweet spot between the proper quantum size and the corresponding costs. Every company is different. For example, a company in an aggressive market may need to move faster and therefore desire a smaller quantum size. Remember, the speed at which new generations appear is proportional to cycle time, and smaller quanta tend to have shorter cycle times.

Another company may find it pragmatic to build a service-based architecture (covered in Chapter 4) with larger "portion of the application" quantum sizes because it more closely models common change.

As we face an ecosystem that defies planning, many factors determine the best match between architecture and cost. This reflects our observation that the role of architect has expanded: Architectural choices have more impact than ever.

Rather than adhere to decades-old "best practice" guides about enterprise architecture, modern architects must understand the benefits of evolvable systems along with the inherent uncertainty that goes with them.

## Building Enterprise Fitness Functions

In an evolutionary architecture, the role of the enterprise architect revolves around *guidance* and *enterprise-wide fitness functions*. Microservices architectures reflect this changing model. Because each service is operationally decoupled from the others, sharing resources isn't a consideration. Instead, architects provide guidance around the purposeful coupling points in the architecture (such as service templates) and platform choices. Enterprise architecture typically owns this shared infrastructure function and constrains platform choices to those supported consistently enterprise wide.

### Case Study: Legality of Open Source Libraries

At one point, the PenultimateWidgets lawyers started questioning the legal use of the open source libraries at the company. They pored over the licenses of each of the frameworks and libraries and determined that PenultimateWidgets wasn't using anything that causes problems. But then one of the lawyers asked, "How will we know if there is a change in the licensing terms?" There was no such service.

However, once the legal team certified the current libraries, developers located the license text within the library and created a *temporal fitness function* that always checks for changes in that string. Thus, every time the library license changes (for any reason), the fitness function triggers that something has changed. Of course, the fitness function won't be sophisticated enough to determine if the change is appropriate—someone will be stuck with that chore—but architects can build fitness functions that trigger directed attention rather than automating a solution.

The other new role that evolutionary architecture creates has enterprise architects defining enterprise-wide fitness functions. Enterprise architects are typically respsonsible for enterprise-wide nonfunctional requirements, such as scalability and security. Many organizations lack the ability to automatically assess how well projects perform individually and in aggregate for these characteristics. Once projects adopt fitness

functions to protect parts of their architecture, enterprise architects can utilize the same mechanism to verify that enterprise-wide characteristics remain intact.

If each project uses a deployment pipeline to apply fitness functions as part of their build, enterprise architects can insert some of their own fitness functions as well. This allows each project to verify cross-cutting concerns, such as scalability, security, and other enterprise-wide concerns, on a continual basis, discovering flaws as early as possible. Just as projects in microservices share service templates to unify parts of technical architecture, enterprise architects can use deployment pipelines to drive consistent testing across projects.

### Case Study: PenultimateWidgets as a Platform

Business at PenultimateWidgets is going so well they have decided to sell part of their platform to other sellers of things like widgets. Part of the appeal of the PenultimateWidgets platform is its proven scalability, resiliency, performance, and other assets. However, their architects don't want to sell the platform only to start hearing stories of failures because users extend it in damaging ways.

To help preserve the important characteristics of the platform, the PenultimateWidgets architects provide a deployment pipeline along with the platform with built-in fitness functions around important dimensions. To remain certified, users of the platform must preserve the existing fitness functions and (hopefully) add their own as they extend the platform.

## Where Do You Start?

Many architects with existing architectures that resemble Big Balls of Mud struggle with where to start adding evolvability. While appropriate coupling and using modularity are some of the first steps you should take, sometimes there are other priorities. For example, if your data schema is hopelessly coupled, determining how DBAs can achieve modularity might be the first step. Here are some common strategies and reasons to adopt the practices around building evolutionary architectures.

### Low-Hanging Fruit

If an organization needs an early win to prove the approach, architects may choose the easiest problem that highlights the evolutionary architecture approach. Generally, this will be part of the system that is already decoupled to a large degree and hopefully not on the critical path to any dependencies. Increasing modularity and decreasing coupling allows teams to demonstrate other aspects of evolutionary architecture, namely fitness functions and incremental change. Building better isolation allows more focused testing and the creation of fitness functions. Better isolation of deploya-

ble units makes building deployment pipelines easier and provides a platform for building more robust testing.

Metrics are a common adjunct to the deployment pipeline in incremental change environments. If teams use this effort as a proof-of-concept, developers should gather appropriate metrics for both before and after scenarios. Gathering concrete data is the best way to for developers to vet the approach; remember the adage that *demonstration defeats discussion*.

This "easiest first" approach minimizes risk at the possible expense of value, unless a team is lucky enough to have *easy* and *high value* align. This is a good strategy for companies that are skeptical and want to dip their toes in the metaphorical water of evolutionary architecture.

## Highest-Value

An alternative approach to "easiest first" is "highest value first"—find the most critical part of the system and build evolutionary behavior around it first. Companies may take this approach for several reasons. First, if architects are convinced that they want to pursue an evolutionary architecture, choosing the highest value portion first indicates commitment. Second, for companies still evaluating these ideas, their architects may be curious as to how applicable these techniques are within their ecosystem. Thus, by choosing the highest value part first, they demonstrate the long-term value proposition of evolutionary architecture. Third, if architects have doubts that these ideas can work for their application, vetting the concepts via the most valuable part of the system provides actionable data as to whether they want to proceed.

## Testing

Many companies lament the lack of testing their systems have. If developers find themselves in a code base with anemic or no testing, they may decide to add some critical tests before undertaking the more ambitious move to evolutionary architecture.

It is generally frowned upon for developers to undertake a project that only adds tests to a code base. Management looks upon this activity with suspicion, especially if new feature implementation is delayed. Rather, architects should combine increasing modularity with high-level functional tests. Wrapping functionality with unit tests provides better scaffolding for engineering practices such as test-driven development (TDD) but takes time to retrofit into a code base. Instead, developers should add coarse-grained functional tests around some behavior before restructuring the code, allowing you to verify that the overall system behavior hasn't changed because of the restructuring.

Testing is a critical component to the incremental change aspect of evolutionary architecture, and fitness functions leverage tests aggressively. Thus, at least some level of testing enables these techniques, and a strong correlation exists between comprehensiveness of testing and ease of implementing an evolutionary architecture.

## Infrastructure

New capabilities come slow to some companies, and the operations group is a common victim of lack of innovation. For companies that have a dysfunctional infrastructure, getting those problems solved may be a precursor to building an evolutionary architecture. Infrastructure issues come in many forms. For example, some companies outsource all their operational responsibilities to another company and thus don't control that critical piece of their ecosystem; the difficultly of DevOps rises orders of magnitude when saddled with the overhead of cross-company coordination.

Another common infrastructure dysfunction is an impenetrable firewall between development and operations, where developers have no insight into how code eventually runs. This structure is common in companies rife with politics across divisions, where each silo acts autonomously.

Lastly, architects and developers in some organizations have ignored good practices and consequently built massive amounts of technical debt that manifests within infrastructure. Some companies don't even have a good idea of what runs where and other basic knowledge of the interactions between architecture and infrastructure.

### Infrastructure Can Impact Architecture

Neal once did consulting work for a company that ran a hosted service for users. The company a large number of servers (approximately 2500 at the time), and had built silos *within* the operations group: One team installed hardware, another installed operating systems, and a third team installed applications. Needless to say, when a developer wanted a resource, they cast a ticket into the black hole of operations, where more tickets were generated and bounced around for weeks until resources appeared. To exacerbate the problem, the company's CIO had left the year before, and the CFO was handling his department. Of course, the CFO was concerned primarily with cost savings, not modernizing what he viewed as merely overhead.

While investigating operation weaknesses, one of the developers mentioned that each server only accommodated about five users, which was shocking considering the simplicity of the application. Sheepishly, developers explained that they had abused HTTP session state to legendary degrees, essentially treating it as a huge in-memory database. Thus, they could only host a few users per server. The problem was that their operations group could not produce a realistic production-like environment for debugging purposes, and they absolutely forbade developers from debugging (or even

extensive monitoring) for production, mostly because of political forces. Without the ability to interact with a realistic version of the application, developers couldn't untangle the mess they had gradually created.

Performing some back of the envelope calculations, we ascertained that the company could likely run on an order of magnitude fewer servers, more like 250. Yet, the company was too busy buying new servers, installing operating systems, and so on. The grand irony, of course, is that their cost-saving measures actually cost the company a huge sum.

Ultimately, the besieged developers created their own guerilla DevOps group and started managing servers themselves, bypassing the traditional operations organization entirely. A fight loomed in the future between the two groups, but in the short term, the developers started making progress in restructuring their application.

Ultimately, the advice parallels the annoying-but-accurate consultant's answer of *It Depends!* Only architects, developers, DBAs, DevOps, testing, security, and the other host of contributors can ultimately determine the best roadmap toward evolutionary architecture.

## Case Study: Enterprise Architecture at PenultimateWidgets

PenultimateWidgets is considering revamping a major part of their legacy platform, and a team of enterprise architects generated a spreadsheet listing all the properties the new platform should exhibit: security, performance metrics, scalability, deployability, and a host of other properties. Each category contained 5 to 20 cells, each with some specific criteria. For example, one of the uptime metrics insisted that each service offer five nines (99.999) of availability. In total, they identified 62 discrete items.

But they realized some problems with this approach. First, would they verify each of these 62 properties on projects? They could create a policy, but who would verify that policy on an ongoing basis? Verifying all these things manually, even on an ad hoc basis, would be a considerable challenge.

Second, would it make sense to impose strict availability guidelines across every part of the system? Is it critical that the administrator's management screens offer five nines? Creating blanket policies often leads to egregious overengineering.

To solve these problems, the enterprise architects defined their criteria as fitness functions and created a deployment pipeline template each project starts with. Within the deployment pipeline, the architects designed fitness functions to automatically check critical features such as security, leaving individual teams to add specific fitness functions (like availability) for their service.

# Future State?

What is the future state of evolutionary architecture? As teams become more familiar with the ideas and practices, they will subsume them into business as usual and start using these ideas to build new capabilities, such as data-driven development.

Much work must be done around the more difficult kinds of fitness functions, but progress is already occurring as organizations solve problems and open source many of their solutions. In the early days of agility, people lamented that some problems were just too hard to automate, but intrepid developers kept chipping away and now entire data centers have succumbed to automation. For instance, Netflix has made tremendous innovations in conceptualizing and building tools like the Simian Army, supporting holistic continuous fitness functions (but not yet calling them that).

There are a couple of promising areas.

## Fitness Functions Using AI

Gradually, large open source artificial intelligence frameworks are becoming available for regular projects. As developers learn to utilize these tools to support software development, we envision fitness functions based on AI that look for anomalous behavior. Credit card companies already apply heuristics such as flagging near-simultaneous transactions in different parts of the world; architects can start to build investigatory tools to look for odd behaviors in architecture.

## Generative Testing

A practice common in many functional programming communities gaining wider acceptance is the idea of *generative testing*. Traditional unit tests include assertions of correct outcomes within each test case. However, with generative testing, developers run a large number of tests and capture the outcomes then use statistical analysis on the results to look for anomalies. For example, consider the mundane case of boundary checking ranges of numbers. Traditional unit tests check the known places where numbers break (negatives, rolling over numerical sizes, and so on) but are immune to unanticipated edge cases. Generative tests check every possible value and report on edge cases that break.

# Why (or Why Not)?

No silver bullets exist, including in architecture. We don't recommend that every project take on the extra cost and effort of evolvability unless it benefits them.

## Why Should a Company Decide to Build an Evolutionary Architecture?

Many businesses find that the cycle of change has accelerated over the past few years, as reflected in the aforementioned *Forbes* observation that every company must be competent at software development and delivery.

### Predictable versus evolvable

Many companies value long-term planning for resources and other strategic matters; companies obviously value *predictability*. However, because of the dynamic equilibrium of the software development ecosystem, predictability has expired. Enterprise architects may still make plans, but they may be invalidated at any moment.

Even companies in staid, established industries shouldn't ignore the perils of systems that cannot evolve. The taxi industry was a multicentury, international institution when it was rocked by ride-sharing companies that understood and reacted to the implications of the shifting ecosystem. The phenomenon known as The Innovators Dilemma predicts that companies in well-established markets are likely to fail as more agile startups address the changing ecosystem better.

Building evolvable architecture takes extra time and effort, but the reward comes when the company can react to substantive shifts in the marketplace without major rework. Predictability will never return to the nostalgic days of mainframes and dedicated operations centers. The highly volatile nature of the development world increasingly pushes all organizations toward incremental change.

### Scale

For a while, the best practice in architecture was to build transactional systems backed by relational databases, using many of the features of the database to handle coordination. The problem with that approach is scaling—it becomes hard to scale the backend database. Lots of byzantine technologies spawned to mitigate this problem, but they were only bandaids to the fundamental problem of scale: coupling. Any coupling point in an architecture eventually prevents scale, and relying on coordination at the database eventually hits a wall.

Amazon faced this exact problem. The original site was designed with a monolithic frontend tied to a monolithic backend modeled around databases. When traffic increased, they had to scale up the databases. At some point, they reached the limits of database scale, and the impact on their site was decreasing performance—every page loaded more slowly.

Amazon realized that coupling everything to one *thing* (whether a relational database, enterprise service bus, and so on) ultimately limited scalability. By redesigning their architecture in a more microservices style that eliminated inappropriate coupling, they allowed their overall ecosystem to scale.

A side benefit of that level of decoupling is enhanced evolvability. As we have illustrated throughout the book, inappropriate coupling represents the biggest challenge to evolution. Building a scalable system also tends to correspond to an evolvable one.

#### Advanced business capabilities

Many companies look with envy at Facebook, Netflix, and other cutting-edge technology companies because they have sophisticated features. Incremental change allows well-known practices such as hypotheses and data-driven development. Many companies yearn to incorporate their users into their feedback loop via multivariate testing. A key building block for many advanced DevOps practices is an architecture that can evolve. For example, developers find it difficult to perform A/B testing if a high degree of coupling exists between components, making isolation of concerns more daunting. Generally, an evolutionary architecture allows a company better technical responsiveness to inevitable but unpredictable changes.

#### Cycle time as a business metric

In "Deployment Pipelines" on page 31, we made the distinction between *Continuous Delivery*, where at least one stage in the deployment pipeline performs a manual *pull*, and *Continuous Deployment*, where every stage automatically promotes to the next upon success. Building continuous deployment takes a fair amount of engineering sophistication—why would a company go quite that far?

Because cycle time has become a business differentiator in some markets. Some large conservative organizations view software as overhead and thus try to minimize cost. Innovative companies see software as a competitive advantage. For example, if AcmeWidgets has created an architecture where the cycle time is three hours, and PenultimateWidgets still has a six-week cycle time, AcmeWidgets has an advantage they can exploit.

Many companies have made cycle time a first-class business metric, mostly because they live in a highly competitive market. All markets eventually become competitive in this way. For example, in the early 1990s, some big companies were more aggressive in moving toward automating manual workflows via software and gained a huge advantage as all companies eventually realized that necessity.

#### Isolating architectural characteristics at the quantum level

Thinking of traditional nonfunctional requirements as fitness functions and building a well-encapsulated architectural quantum allows architects to support different characteristics per quantum, one of the benefits of a microservices architecture. Because the technical architecture of each quantum is decoupled from other quanta, architects can choose different architectures for different use cases. For example, developers on one small service may choose a microkernel architecture because they want to sup-

port a small core that allows incremental addition. Another team of developers may choose an event-driven architecture for their service because of scalability concerns. If both services were part of a monolith, architects would have to make tradeoffs to attempt to satisfy both requirements. By isolating technical architecture at a small quantum level, architects are free to focus on the primary characteristics of a singular quantum, not analyzing the tradeoffs for competing priorities.

## Case Study: Selective Scale at PenultimateWidgets

PenultimateWidgets has some services that require little in the way of scale and are therefore written in simple technology stacks. However, a couple of services stand out. The `Import` service must import inventory figures from brick-and-mortar stores every night for the accounting system. Thus, the architectural characteristics and fitness functions the developers built into `Import` include *scalability* and *resiliency*, which greatly complicate the technical architecture of that service. Another service, `MarketingFeed`, is typically called by each store at opening to get daily sales and marketing updates. Operationally, `MarketingFeed` needs *elasticity* to be able to handle the burst of requests as stores open across time zones.

A common problem in highly coupled architectures is inadvertent overengineering. In a more coupled architecture, developers would have to build scalability, resiliency, and elasticity into every service, complicating the ones that don't need those capabilities. Architects are accustomed to choosing architectures against a spectrum of tradeoffs. Building architectures with clearly defined quantum boundaries allows exact specification of the required architectural characteristics.

### Adaptation versus evolution

Many organizations fall into the trap of gradually increasing technical debt and reluctance to make needed restructuring modifications, which in turns makes systems and integration points increasingly brittle. Companies try to pave over this brittleness with connection tools like service buses, which alleviates some of the technical headaches but doesn't address deeper logical cohesion of business processes. Using a service bus is an example of *adapting* an existing system to use in another setting. But as we've highlighted previously, a side effect of adaptation is increased technical debt. When developers adapt something, they preserve the original behavior and layer new behavior alongside it. The more adaptation cycles a component endures, the more parallel behavior there is, increasing complexity, hopefully strategically.

The use of feature toggles offers a good example of the benefits of adaptation. Often, developers use toggles when trying several alternate alternatives via hypotheses-driven development, testing their users to see what resonates best. In this case, the technical debt imposed by toggles is purposeful and desirable. Of course, the engi-

neering best practices around these types of toggles is to remove them as soon as the decision is resolved.

Alternatively, *evolving* implies fundamental change. Building an evolvable architecture entails changing the architecture in situ, protected from breakages via fitness functions. The end result is a system that continues to evolve in useful ways without an increasing legacy of outdated solutions lurking within.

## Why Would a Company Choose Not to Build an Evolutionary Architecture?

We don't believe that evolutionary architecture is the cure for all ailments! Companies have several legitimate reasons to pass on these ideas.

### Can't evolve a ball of mud

One of the key "-ilities" architects neglect is *feasibility*—should the team undertake this project? If an architecture is a hopelessly coupled Big Ball of Mud, making it possible to evolve it cleanly will take an enormous amount of work—likely more than rewriting it from scratch. Companies loath throwing anything away that has perceived value, but often rework is more costly than rewrite.

How can companies tell if they're in this situation? The first step to converting an existing architecture into an evolvable one is *modularity*. Thus, a developer's first task requires finding whatever modularity exists in the current system and restructuring the architecture around those discoveries. Once the architecture becomes less entangled, it becomes easier for architects to see underlying structures and make reasonable determinations about the effort needed for restructuring.

### Other architectural characteristics dominate

*Evolvability* is only one of many characteristics architects must weigh when choosing a particular architecture style. No architecture can fully support conflicting core goals. For example, building high performance and high scale into the same architecture is difficult. In some cases, other factors may outweigh evolutionary change.

Most of the time, architects choose an architecture for a broad set of requirements. For example, perhaps an architecture needs to support high availability, security, and scale. This leads towards general architecture patterns, such as monolith, microservices, or event-driven. However, a family of architectures known as *domain-specific architectures* that attempt to maximize a single characteristic.

An excellent example of a domain-specific architecture is LMAX, a custom trading solution. Their primary goal was fast transaction throughput, and they experimented with a variety of techniques with no success. Ultimately, by analyzing at the lowest level, they discovered the key to scalability was making their logic small enough to fit

in the CPU's cache, and preallocating all memory to prevent garbage collection. Their architecture achieved a stunning 6 million transactions per second on a single Java thread!

Having built their architecture for such a specific purpose, evolving it to accomodate other concerns would present difficulties (unless developers are extraordinarily lucky and architectural concerns overlap). Thus, most domain-specific architectures aren't concerned with evolution because their specific purpose overrides other concerns.

#### Sacrificial architecture

Martin Fowler defined a sacrificial architecture as one designed to throw away. Many companies need to build simple versions initially to investigate a market or prove viability. Once proven, they can build the *real* architecture to support the characteristics that have manifested.

Many companies do this strategically. Often, companies build this type of architecture when creating a minimum viable product to test a market, anticipating building a more robust architecture if the market approves. Building a sacrificial architecture implies that architects aren't going to try to evolve it but rather replace it at the appropriate time with something more permanent. Cloud offerings make this an attractive option for companies experimenting with the viability of a new market or offering.

#### Planning on closing the business soon

Evolutionary architecture helps businesses adapt to changing ecosystem forces. If a company doesn't plan to be in business in a year, there's no reason to build evolvability into their architecture.

Some companies are in this position; they just don't realize it yet.

## Convincing Others

Architects and developers struggle to make nontechnical managers and coworkers understand the benefits of something like evolutionary architecture. This is especially true of parts of the organization most disrupted by some of the necessary changes. For example, developers who lecture the operations group about doing their job incorrectly will generally find resistance.

We introduced the best solution to this problem in Chapter 6. Rather than try to *convince* reticent parts of the organization, *demonstrate* how these ideas improve their practices.

## Case Study: Consulting Judo

A colleague was working with a big retailer trying to convince the enterprise architects and operations group to embrace more modern DevOps practices, such as automated machine provisioning, better monitoring, and so on. Yet her pleas fell on deaf ears because of two common refrains: "We don't have time" and "Our setup is so complex, those things will never work here."

She applied an excellent technique called *consulting judo*. Judo as a martial art has numerous techniques that use the opponent's weight against them. *Consulting judo* entails finding a particular pain point and fixing it as an exemplar. The pain point at the retailer was QA environments: There were never enough of them. Consequently, teams would attempt to share environments, but that caused major headaches. Having found her case study, she received approval to creating QA environments using modern DevOps tools and techniques.

When she was complete, she demonstrated the falseness of both previous assumptions. Now, any team that needs a QA environment can provision one trivially. Her effort in turn convinced operations to invest more fully into modern techniques because of demonstrable value. Demonstration defeats discussion.

# The Business Case

Business people are often wary of ambitious IT projects, which sound like expensive replumbing exercises. However, many businesses find that many desirable capabilities have their basis in more evolutionary architectures.

## "The Future Is Already Here…"

> The future is already here—it's just not very evenly distributed.
>
> —William Gibson

Many companies view software as overhead, like the heating system. When software architects talk to executives at those companies about innovation in software, they imagine plumbers upselling them on pretty but expensive overhead. However, that antiquated view of the strategic importance of software is discredited. Consequently, decision makers who control software purchases tend to became institutionally conservative, valuing cost savings over innovation. Enterprise architects make this mistake for understandable reasons—they look at other companies within their ecosystem to see how they approach these decisions. But that approach is dangerous because a disruptive company that has modern software architecture may move into the existing company's realm and suddenly dominate because they have better information technology.

### Moving Fast Without Breaking Things

Most large enterprises complain about the pace of change within the organization. One side effect of building an evolutionary architecture manifests as better engineering efficiency. All the practices we call *incremental change* improve automation and efficiency. Defining top-level enterprise architecture concerns as fitness functions both unifies a disparate set of concerns under one umbrella and forces developers to think in terms of objective outcomes.

Building an evolutionary architecture implies that teams can make incremental changes at the architectural level with confidence. In Chapter 2, we described a GitHub case study where a foundational component of an architecture with no regressions (while uncovering other undiscovered bugs). Business people fear breaking change. If developers build an architecture that allows incremental change with better confidence than older architectures, both business and engineering win.

### Less Risk

With improved engineering practices comes decreased risk. Evolutionary architecture forces modern practices on teams in the guise of incremental change, a beneficial side effect. Once developers have confidence that their practices will allow them to make changes in the architecture without breaking things, companies can increase their release cadence.

### New Capabilities

The best way to sell the ideas of evolutionary architecture to the business revolves around the new business capabilities it delivers, such as hypothesis-driven development. Business people glaze over when architects wax poetic about technical improvements, so it is better to couch the impact in their terms.

## Building Evolutionary Architectures

Our ideas about building evolutionary architectures build upon and rely on many existing things: testing, metrics, deployment pipelines, and a host of other supporting infrastructure and innovation. We're creating a new perspective to unify previously diversified concepts using fitness functions. For us, anything that verifies the architecture is a fitness function, and treating all those mechanisms uniformly makes automation and verification easier.

We want architects to start thinking of architectural characteristics as *evaluable* things rather than ad hoc aspirations, allowing them to build more resilient architectures.

Making some systems more evolvable won't be easy, but we don't really have a choice: The software development ecosystem is going to continue to churn out new ideas

from unexpected places. Organizations who can react and thrive in that environment will have a serious advantage.

# Index


## A

A/B testing, 150
abstraction distraction antipattern, 111
abstractions, leaky, 125-127
accidental (unintentional) coupling, 74, 133, 137
Ackoff, Russel, 17
adaptation, evolution vs., 14, 160
Amazon
  scaling problems, 158
  two-pizza teams, 145
Amazon Cloud, 36
Ambler, Scott, 83
anticorruption layers, 77, 111
antipatterns, 123-139
  abstraction distraction, 111
  Big Ball of Mud, 52, 161
  coupling (see inappropriate coupling)
  inappropriate governance, 132
  incremental change, 131-133
  reporting, 137
  Vendor King, 77, 123
application programming interfaces (APIs), 4
application services, 66
appropriate coupling
  in Big Ball of Mud, 53
  in broker EDAs, 63
  in COTS software, 99
  in ESB-driven SOA, 68
  in layered monolithic architecture, 56
  in mediator EDAs, 64
  in microkernels, 60
  in microservices, 72
  in modular monoliths, 57
  in monolithic architectures, 54
  in serverless architectures, 78
  in service-based architectures, 76
  service templates for, 113
architectural concerns, 8
architectural coupling, 47-68
  appropriate (see appropriate coupling)
  architectural quanta and granularity, 48-51
  controlling quanta size, 78
  modularity, 47
  PenultimateWidgets case study, 79-81
  serverless architectures, 76-78
architectural quanta, 48-51
  controlling size of, 78
  cost of architecture and, 151
  defined, 48
  for ESB-driven SOA, 67
  in microservices architecture, 48-51
  isolating architectural characteristics at level of, 159
architectural styles
  Big Ball of Mud, 52
  event-driven architectures, 60-65
  evolvability of, 51-60
  monoliths, 53-58
  service-oriented architectures, 65-68
artificial intelligence (AI), 157
atomic fitness functions, 19, 62
automation
  of DevOps, 143
  of fitness functions, 20, 96

## B

BaaS (Backend as a Service), 76

back port, 21
Big Ball of Mud antipattern, 52, 161
bit rot, 6
blue/green deployments, 109
bounded context
  as quantum boundary in microservices architecture, 48-51
  defined, 48
break upon upgrade test, 21
broker EDA, 60-63
Brooks, Fred
  on sacrificial architecture, 114
  on second system syndrome, 115
Brown, Simon, 9, 58
budgeting, 151
building evolutionary architecture, 141-165
  building enterprise fitness functions, 152
  business case for, 163
  CFO and budgeting, 151
  connections between team members, 146
  convincing others of benefits, 162
  cross-functional teams, 141-143
  culture of experimentation, 149
  easiest-first starting point, 153
  external change, 145
  fitness functions using AI, 157
  future state of, 157
  generative testing, 157
  highest-value-first starting point, 154
  infrastructure as starting point, 155
  organizational factors, 141-147
  organizing teams around business capabilities, 143
  PenultimateWidgets as platform, 153
  PenultimateWidgets case study, 156
  product over project, 144
  reasons for building, 158-161
  reasons not to build, 161
  reasons to build, 158-161
  starting points, 153-156
  team coupling characteristics, 147-150
  team culture, 148
  testing as starting point, 154
business capabilities
  as basis for team organization, 143
  as reason for building evolutionary architecture, 159, 164
business case for evolutionary architecture, 163
business concerns
  inappropriate data coupling, 136-139
  microservices and, 70
  planning horizons pitfall, 138
  product customization pitfall, 136
  reporting antipattern, 137
business metric, cycle time as, 159

## C

change
  as constant in software development, 5
  long-term planning and, 3
  pace of, 164
Chaos Monkey, 36
cloud environments, sacrificial architecture and, 115
code reuse
  abuse of, 128-130
  code usability vs., 129
  microservices and, 129
  PenultimateWidgets case study, 130
coding standards, fitness functions and, 18
component cycles, 30, 79-81
components, defined, 47
Conformity Monkey, 36
considered harmful, 116
consulting judo, 163
continual architecture, 6
continual fitness functions, 19
Continuous Delivery
  Continuous Deployment vs., 159
  cycle time and, 134
  deployment pipelines and, 31-35
  microservices and, 71
  origins of, 25
  snapshots vs., 118
  with microservices, 73
Continuous Deployment, 33, 159
continuous integration (CI), 31
contracts, 58
Conway's Law, 11
Conway, Melvin, 11
Cook, John D., 128
coordination friction, 142
COTS (Commercial Off The Shelf) software, 99
coupling, 30
  (see also appropriate coupling; architectural coupling; inappropriate coupling)
  analyzing with JDepend, 30
  Big Ball of Mud, 52

duplication vs., 129
cross-functional teams, 141-143
culture
of experimentation, 149
team, 148
customization pitfall, 136
cycle time
and engineering efficiency, 96
as reason for building evolutionary architecture, 159
Continuous Delivery and, 134
fitness functions and, 135
cyclomatic complexity, 18

## D

data (see evolutionary data)
data coupling, inappropriate (see inappropriate coupling)
data-driven development, 42
database migration tools, 84
databases
evolutionary design, 83-89
evolving schemas, 83-85
Shared Database Integration, 85-89
DBAs
defined, 83
vendors and tool choices, 90
decoupling, forced, 133
dependences, external, 115-117
deployment pipelines
at Penultimate Widgets, 33, 41
combining fitness functions categories, 35-37
continuous integration vs., 31
incremental change and, 31-35
with fitness functions, 32
DevOps
automating, 143
with microservices, 73
Dijkstra, Edsger, 116
dimensions
identifying when building evolvable architectures, 95
of evolutionary architecture, 8-11
partitioning techniques for, 9
disruptive change, 4
Docker, 4
domain dimension, microservices and, 68-74
domain-centric teams, 141-143
domain-driven design (DDD)
and bounded context, 48
ubiquitous language in, 50
domain-specific architectures, 161
domain-specific fitness functions, 21
duplication, coupling vs., 129
dynamic equilibrium, 4
dynamic fitness functions, 20

## E

easiest-first approach, 153
eBay, 114, 162
Eclipse Java IDE, 59
ecosystem, software development, 3
Edison, Thomas Alva, 149
emergent fitness functions, 21
endpoints, versioning, 119
engineering safety net, 146
enterprise architecture, 156
Enterprise Resource Planning (ERP) software, 123
enterprise services, 66
enterprise-wide fitness functions, 152
Evans, Eric, 48, 50
event-driven architectures (EDA), 60-65
broker, 60-63
mediator, 63-65
evolution, adaptation vs., 14, 160
evolutionary architecture (generally)
adaptable architecture vs., 14
balancing long-term planning with constant change, 3
basics, 3
Conway's Law and, 11
defined, 6
dimensions of, 8-11
evolvability of architectural styles, 51-60
future state of, 157
guided change and, 7
in practice, 14, 141-165
(see also building evolutionary architecture)
incremental change and, 6
pitfalls and antipatterns, 123-139
(see also inappropriate coupling)
preventing degradation of, 6
reasons for name, 13
reasons not to build, 161
reasons to build, 158-161

evolutionary data, 83-94
  database design for, 83-89
  inappropriate data coupling, 89-93
  PenultimateWidgets case study, 93
evolvability
  as meta-characteristic, 6
  conflicting core goals and, 161
  predictability vs., 110, 158
evolvable architectures
  anticorruption layers, 111
  avoiding snowflake servers, 109
  building, 95-107
  Continuous Delivery vs. snapshots, 118
  COTS implications, 99
  coupling and cohesion for, 97
  defining fitness functions for each dimension, 96
  engineering practices, 98
  fitness functions, 98
  guidelines for building, 107-119
  identifying dimensions affected by evolution, 95
  in new projects, 97
  making decisions reversible, 109
  mechanics of building, 95-96
  migrating architectures, 100-107
  mitigating external change, 115-117
  PenultimateWidgets case study, 119-122
  prefer evolvable over predictable, 110
  refactoring vs. restructuring, 99
  removing needless variability, 107-109
  retrofitting existing architectures, 97-100
  sacrificial architectures, 114
  service template case study, 113
  updating libraries vs. frameworks, 117
  using deployment pipelines to automate fitness functions, 96
  versioning, 119
expand/contract pattern, 86
experimentation, culture of, 149
external change, 145
external dependences, 115-117

## F

FaaS (Function as a Service), 77
Facebook, 43
fan in operation, 34
fan out operation, 34
Farley, Dave, 25
feature toggles, 34, 110, 136, 160
fitness functions, 15-24
  adding to PenultimateWidgets' invoicing service, 40-42
  AI for, 157
  and retrofitting existing architectures, 98
  atomic vs. holistic, 19
  automated vs. manual, 20
  basics, 17
  brief definition, 15
  categories of, 18-21
  combining categories of, 35-37
  cycle time and, 135
  defined, 7
  deployment pipelines with, 32
  domain-specific, 21
  engineering safety net, 146
  enterprise-wide, 152
  guided change with (see guided change with fitness functions)
  importance of early identification, 22-23
  in COTS software, 100
  intentional over emergent, 21
  key, 22
  not relevant, 23
  ownership and maintenance of, 30
  relevant, 23
  review of, 23
  static vs. dynamic, 20
  systemwide, 16
  temporal, 21
  triggered vs. continual, 19
fluid dependencies, 118
forced decoupling, 133
Fowler, Chad, 108, 133
Fowler, Martin, 114, 162
frameworks, libraries vs., 117
functional cohesion, 48
functionality, porting of, 44
future state of evolutionary architecture, 157
  fitness functions using AI, 157
  generative testing, 157

## G

generative testing, 157
Gibson, William, 163
GitHub, architectural restructuring at, 37-39
Goldilocks Governance
  at PenultimateWidgets, 134-136

defined, 133
Google, 20% time at, 150
governance
Goldilocks, 134-136
inappropriate, 132
greenfield projects, 97
guarded dependencies, 119
guided change with fitness functions, 7
in Big Ball of Mud, 53
in broker EDAs, 62
in COTS software, 100
in ESB-driven SOA, 67
in layered monolithic architecture, 56
in mediator EDAs, 64
in microkernels, 60
in microservices, 72
in modular monoliths, 57
in monolithic architectures, 54
in serverless architectures, 77
in service-based architectures, 76

## H

Hackman, J. Richard, 146
Hickey, Rich, 112
highest-value-first approach, 154
holistic fitness functions, 19
Humble, Jez, 25
hypothesis-driven development, 42-44

## I

IBM (San Francisco Project), 128
immutable infrastructure, 108
inadvertent (accidental) coupling, 74, 133, 137
inappropriate coupling, 89-93, 123
(see also appropriate coupling)
age/quality of data, 92
Big Ball of Mud as example of, 53
business concerns, 136-139
code reuse abuse, 128-130
incremental change antipatterns, 131-133
Last 10% trap, 127
leaky abstractions, 125-127
PenultimateWidgets case study, 130, 134-136
planning horizons pitfall, 138
product customization pitfall, 136
reporting antipattern, 137
Resume-Driven Development, 131
technical architecture, 123-130
two-phase commit transactions, 90-92
vendor king antipattern, 123
inappropriate governance, 132
incremental change
antipatterns, 131-133
basics, 6
building blocks for agility at architecture level, 28-39
combining fitness functions categories, 35-37
deployment pipelines and, 31-35
engineering of, 25-45
GitHub case study, 37-39
hypothesis- and data-driven development, 42-44
identifying conflicting evolution goals, 39
in Big Ball of Mud, 52
in broker EDAs, 62
in COTS software, 99
in ESB-driven SOA, 67
in layered monolithic architecture, 55
in mediator EDAs, 64
in microkernels, 60
in microservices, 72
in modular monoliths, 57
in monolithic architectures, 54
in serverless architectures, 77
in service-based architectures, 75
inappropriate governance antipattern, 132
PenultimateWidgets case studies, 25-28, 40-42, 44
testability of architecture, 29-31
indirection, 106
infrastructure dysfunction, 155
infrastructure services, 66
integration coupling, 72
intentional fitness functions, 21
internal resolution, 119
Inverse Conway Maneuver
for identifying protected dimensions, 96
PenultimateWidgets and, 13
irrational artifact attachment, 139
isolation of layers, 55

## J

Java, 4, 30, 80
JavaScript, 116
JDepend, 30, 80
Johnson, Ralph, 1

just in time anticorruption layer, 112

## K

kaizen (continuous improvement), 150
Kenney, Kevlin, 106
key fitness functions, 22

## L

Last 10% trap, 124, 127
last responsible moment principle, 111, 135
layered architecture, 6, 54
lead time, 134
leaky abstractions, 125-127
legacy data, 92
Let's Stop Working and Call It A Success Concession principle, 124
libraries
  as component, 47
  frameworks vs., 117
Linux, 4
LMAX, 161
long-term planning, 3-6

## M

manual fitness functions, 20
Meadows, Donella H., 8
mediator EDA, 63-65
message bus, 66
microkernel, 58-60
microservices architecture, 68-74
  bounded context as quantum boundary, 48-51
  duplication over coupling in, 129
  forced decoupling, 133
  governance and, 133
  principles of, 70-72
  service-based architecture vs., 74
  share nothing architecture, 26
  transactional systems and, 91
migration
  from one architectural style to another, 100-107
  steps in, 101-104
modularity, architectural coupling and, 47
monitoring-driven development (MDD), 20
monolithic architectures, 53-58
  layered architecture, 54
  microkernel, 58-60
  migrating to service-based architecture, 101-104
  quanta in, 48
  reporting antipattern and inadvertent coupling, 137

## N

naive partitioning, 106
nested feedback loop, 44
Netflix, 36
Newman, Sam, 107

## O

open-source libraries, legal issues with, 152

## P

package dependency cycles, 79-81
pair programming, 108
PenultimateWidgets (fictional case study)
  adding fitness functions to invoicing service, 40-42
  deployment pipelines at, 33, 41
  evolving routing, 93
  functionality porting decisions, 44
  Goldilocks Governance, 134-136
  guarding against component cycles, 79-81
  Inverse Conway Maneuver, 13
  legality of open-source libraries, 152
  operational aspects of incremental change at, 25-28
  reusable components, 130
  selective scaling, 160
  selling platform, 153
  separation of domain services and reporting services, 137
  star rating service upgrade, 7, 26-28, 119-122
performance requirements, fitness functions and, 17
pitfalls, 123-139
  planning horizons, 138
  product customization, 136
plug-ins, 58-60
porting of functionality, 44
predictability, evolvability vs., 110, 158
primordial abstraction ooze, 127
product customization pitfall, 136
product, project vs., 144

production environment, exploratory testing in, 35
proof of concept, 114
provider team, 146
pull model, 116
pull updates, 117
push updates, 117

## Q

QA (quality assurance), 35

## R

refactoring, restructuring vs., 99
registry, 59
relevant fitness functions, 23
reporting services antipattern, 137
Resume-Driven Development, 131
retrofitting existing architectures, 97-100
  COTS implications, 99
  coupling and cohesion for, 97
  engineering practices, 98
  fitness functions, 98
  refactoring vs. restructuring, 99
reusability trap, 127
reusable frameworks
  abuse of, 128-130
  PenultimateWidgets case study, 130
reuse of code (see code reuse)
reversible decisions, 109
Richards, Mark, 110
risk, managing with incremental change, 164
routing
  as evolutionary mechanism, 121
  PenultimateWidgets case study, 93
Rumsfeld, Donald, 110

## S

sacrificial architectures, 114, 162
Sadalage, Pramod, 83
San Francisco Project, 128
scaling
  as reason for building evolutionary architecture, 158
  at PenultimateWidgets, 160
schemas
  age/quality of data and, 92
  with evolutionary database design, 83-85
Scientist (GitHub framework), 37-39
second system syndrome, 115
selective scaling, 160
separation of concerns, 55
serverless architectures, 76-78
  BaaS, 76
  FaaS, 77
servers, snowflake, 109
service discovery, 106
service endpoints, versioning, 119
service templates
  case study, 113
  with microservices, 72
service, as component, 47
service-based architectures, 74-76
  migrating monolithic architectures to, 101-104
  transactional systems and, 91
service-oriented architectures (SOA), 65-68
  code reuse in, 129
  microservices, 68-74
  service-based architectures, 74-76
set-based development, 150
share nothing architecture, 26, 71, 73
Shared Database Integration, 85-89
  with existing data and integration points, 88
  with legacy data but no integration points, 88
  with no integration points and no legacy data, 87
Simian Army, 36
snapshots, 118
snowflake computers, 108
snowflake servers, 109
software architecture, 1
  Conway's Law and, 11-14
  defined, 5
  dimensions of, 8-11
  Ralph Johnson's definition, 1
software development ecosystem, 3
speculative updating, 118
spike solutions, 150
Spolsky, Joel, 125
static fitness functions, 20
Sunk Cost Fallacy, 138
systemwide fitness functions, 16

## T

Taylor, Jeffrey, 43
teams

connections between members, 146
coupling characteristics, 147-150
cross-functional, 141-143
culture of experimentation, 149
dealing with external change, 145
engineering culture, 148
ideal size, 146
organizational factors, 141-147
organizing around business capabilities, 143
product over project, 144
risks of not identifying fitness functions, 22
structured by functional skills, 11
structured by service boundaries, 12
(see also Inverse Conway Maneuver)
two-pizza, 145
technical architecture
code reuse abuse, 128-130
Last 10% trap, 127
leaky abstractions, 125-127
Resume-Driven Development, 131
vendor king antipattern, 123
technical debt, 111, 115, 160
temporal fitness functions, 21, 152
testability, 29-31
testing
as starting point, 154
generative, 157
Toyota, 150
tradeoffs, 16
transactions, 90-92
triggered fitness functions, 19
Twitter, 114
two-pizza teams, 145


## U


ubiquitous language, 50
undo, 85
unintentional (accidental) coupling, 74, 133, 137
unknown unknowns, 96, 110


## V


Vendor King antipattern, 77, 123
vendors, DBA tool choices and, 90
version numbering, 119


## W


Wheeler, Dave, 106

## 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 in the intersection of agile engineering techniques and software architecture. Neal has authored magazine articles, seven books (and counting), and dozens of video presentations and has spoken at hundreds of developers 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 in 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 web site 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 coming to 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's of Science in Computer Science from Rice University, and her Ph.D. in Computer Science from Rice University. She is also the co-author of *Domain-Specific Languages*, *The ThoughtWorks Anthology*, and *Building Evolutionary Architectures*.

**Patrick Kua** is a Principal Technical Consultant at ThoughtWorks, having worked in the technology industry for over 15 years. He is well known for bringing a balanced

blend between technology, people, and process to improve the effectiveness of software delivery. You can also find him speaking at many conferences on the topics of technical leadership, architecture, and building strong engineering cultures.

He is author of *The Retrospective Handbook: A Guide for Agile Teams and Talking with Tech Leads: From Novices to Practitioners* and established a regular training program to support developers transitioning into the role of a Tech Lead and/or Architect.

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

## 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-accomodate 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. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Jean Vincent Félix Lamouroux's *Exposition Methodique des genres de L'Ordre des Polypiers*. The cover fonts are URW Typewriter 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.