SCALABILITY RULES

PRINCIPLES FOR SCALING WEB SITES

SECOND EDITION

MARTIN L. ABBOTT MICHAEL T. FISHER

"Whether you're taking on a role as a technology leader in a new company or you simply want to make great technology decisions, Scalability Rules *will be the go-to resource on your bookshelf."*
—**Chad Dickerson,** CTO, Etsy

Praise for the First Edition of *Scalability Rules*

"Once again, Abbott and Fisher provide a book that I'll be giving to our engineers. It's an essential read for anyone dealing with scaling an online business."

—**Chris Lalonde**, GM of Data Stores, Rackspace

"Abbott and Fisher again tackle the difficult problem of scalability in their unique and practical manner. Distilling the challenges of operating a fast-growing presence on the Internet into 50 easy-to-understand rules, the authors provide a modern cookbook of scalability recipes that guide the reader through the difficulties of fast growth."

—**Geoffrey Weber**, VP, Internet Operations, Shutterfly

"Abbott and Fisher have distilled years of wisdom into a set of cogent principles to avoid many nonobvious mistakes."

—**Jonathan Heiliger**, VP, Technical Operations, Facebook

"In *The Art of Scalability*, the AKF team taught us that scale is not just a technology challenge. Scale is obtained only through a combination of people, process, *and* technology. With *Scalability Rules*, Martin Abbott and Michael Fisher fill our scalability toolbox with easily implemented and time-tested rules that once applied will enable massive scale."

—**Jerome Labat**, VP, Product Development IT, Intuit

"When I joined Etsy, I partnered with Mike and Marty to hit the ground running in my new role, and it was one of the best investments of time I have made in my career. The indispensable advice from my experience working with Mike and Marty is fully captured here in this book. Whether you're taking on a role as a technology leader in a new company or you simply want to make great technology decisions, *Scalability Rules* will be the go-to resource on your bookshelf."

—**Chad Dickerson**, CTO, Etsy

"*Scalability Rules* provides an essential set of practical tools and concepts anyone can use when designing, upgrading, or inheriting a technology platform. It's very easy to focus on an immediate problem and overlook issues that will appear in the future. This book ensures strategic design principles are applied to everyday challenges."

—**Robert Guild**, Director and Senior Architect, Financial Services

"An insightful, practical guide to designing and building scalable systems. A must-read for both product building and operations teams, this book offers concise and crisp insights gained from years of practical experience of AKF principals. With the complexity of modern systems, scalability considerations should be an integral part of the architecture and implementation process. Scaling systems for hypergrowth requires an agile, iterative approach that is closely aligned with product features; this book shows you how."

—**Nanda Kishore**, CTO, ShareThis

"For organizations looking to scale technology, people, and processes rapidly or effectively, the twin pairing of *Scalability Rules* and *The Art of Scalability* is unbeatable. The rules-driven approach in *Scalability Rules* not only makes this an easy reference companion, but also allows organizations to tailor the Abbott and Fisher approach to their specific needs both immediately and in the future!"

—**Jeremy Wright**, CEO, BNOTIONS.ca, and Founder, b5media

Scalability Rules

Second Edition

This page intentionally left blank

Scalability Rules

Principles for Scaling Web Sites

Second Edition

Martin L. Abbott
Michael T. Fisher

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016944687

ISBN-13: 978-0-13-443160-4
ISBN-10: 0-13-443160-X
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
1 16

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Full-Service Production Manager
Julie B. Nahil

Project Editor
Dana Wilson

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
Barbara Lasoff

Technical Reviewers
Camille Fournier
Chris Lalonde
Mark Uhrmacher

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

❖

This book is dedicated to our friend and partner "Big" Tom Keeven. "Big" refers to the impact he's had in helping countless companies scale in his nearly 30 years in the business.

❖

This page intentionally left blank

Contents

Preface

Thanks for your interest in the second edition of *Scalability Rules*! This book is meant to serve as a primer, a refresher, and a lightweight reference manual to help engineers, architects, and managers develop and maintain scalable Internet products. It is laid out in a series of rules, each of them bundled thematically by different topics. Most of the rules are technically focused, and a smaller number of them address some critical mindset or process concern, each of which is absolutely critical to building scalable products. The rules vary in their depth and focus. Some rules are high level, such as defining a model that can be applied to nearly any scalability problem; others are specific and may explain a technique, such as how to modify headers to maximize the "cacheability" of content. In this edition we've added stories from CTOs and entrepreneurs of successful Internet product companies from startups to Fortune 500 companies. These stories help to illustrate how the rules were developed and why they are so important within high-transaction environments. No story serves to better illustrate the challenges and demands of hyper-scale on the Internet than Amazon. Rick Dalzell, Amazon's first CTO, illustrates several of the rules within this book in his story, which follows.

Taming the Wild West of the Internet

From the perspective of innovation and industry disruption, few companies have had the success of Amazon. Since its founding in 1994, Amazon has contributed to redefining at least three industries: consumer commerce, print publishing, and server hosting. And Amazon's contributions go well beyond just disrupting industries; they've consistently been a thought leader in service-oriented architectures, development team construction, and a myriad of other engineering approaches. Amazon's size and scale along all dimensions of its business are simply mind-boggling; the company has consistently grown at a rate unimaginable for traditional brick-and-mortar businesses. Since 1998, Amazon grew from $600 million (no small business at all) in annual revenue to an astounding $107 billion (that's "billion" with a *B*) in 2015.[1] Walmart, the world's largest retailer, had annual sales of $485.7 billion in 2015.[2] But Walmart has been around since 1962, and it took 35 years to top $100 billion in sales compared to Amazon's 21 years. No book professing to codify the rules of scalability from the mouths of CTOs who have created them would be complete without one or more stories from Amazon.

Jeff Bezos incorporated Amazon (originally Cadabra) in July of 1994 and launched Amazon.com as an online bookseller in 1995. In 1997, Bezos hired Rick Dalzell, who was then the VP of information technology at Walmart. Rick spent the next ten years

at Amazon leading Amazon's development efforts. Let's join Rick as he relays the story of his Amazon career:

"When I was at Walmart, we had one of the world's largest relational databases running the company's operations. But it became clear to the Amazon team pretty quickly that the big, monolithic database approach was simply not going to work for Amazon. Even back then, we handled more transactions in a week on the Amazon system than the Walmart system had to handle in a month. And when you add to that our incredible growth, well, it was pretty clear that monoliths simply were not going to work. Jeff [Bezos] took me to lunch one day, and I told him we needed to split the monolith into services. He said, 'That's great—but we need to build a moat around this business and get to 14 million customers.' I explained that without starting to work on these splits, we wouldn't be able to make it through Christmas."

Rick continued, "Now keep in mind that this is the mid- to late nineties. There weren't a lot of companies working on distributed transaction systems. There weren't a lot of places to go to find help in figuring out how to scale transaction processing systems growing in excess of 300% year on year. There weren't any rulebooks, and there weren't any experts who had 'been there and done that.' It was a new frontier—a new Wild, Wild West. But it was clear to us that we had to distribute this thing to be successful. Contrary to what made me successful at Walmart, if we were going to scale our solution and our organization, we were going to need to split the solution and the underlying database up into a number of services." (The reader should note that an entire chapter of this book, Chapter 2, "Distribute Your Work," is dedicated to such splits.)

"We started by splitting the commerce and store engine from the back-end fulfillment systems that Amazon uses. This was really the start of our journey into the services-oriented architecture that folks have heard about at Amazon. All sorts of things came out of this, including Amazon's work on team independence and the API contracts. Ultimately, the work created a new industry [infrastructure as a service] and a new business for Amazon in Amazon Web Services—but that's another story for another time. The work wasn't easy; some components of the once-monolithic database such as customer data—what we called 'the Amazon customer database or ACB'—took several years to figure out how to segment. We started with services that were high in transaction volumes and could be quickly split in both software and data, like the front- and back-end systems that I described. Each split we made would further distribute the system and allow additional scale. Finally, we got back to solving the hairy problem of ACB and split it out around 2004.

"The team was incredibly smart, but we also had a bit of luck from time to time. It's not that we never failed, but when we would make a mistake we would quickly correct it and figure out how to fix the associated problems. The lucky piece is that none of our failures were as large and well publicized as those of some of the other companies struggling through the same learning curve. A number of key learnings in building these distributed services came out of these splits, learnings such as the need to limit session and state, stay away from distributed two-phase commit transactions, communicating asynchronously whenever possible, and so on. In fact, without a strong bias toward asynchronous communication through a publish-and-subscribe message bus, I don't

know if we could have ever split and scaled the way we did. We also learned to allow things to be eventually consistent where possible, in just about everything except payments. Real-time consistency is costly, and wherever people wouldn't really know the difference, we'd just let things get 'fuzzy' for a while and let them sync up later. And of course there were a number of 'human' or team learnings as well such as the need to keep teams small[3] and to have specific contracts between teams that use the services of other teams."

Rick's story of how he led Amazon's development efforts in scaling for a decade is incredibly useful. From his insights we can garner a number of lessons that can be applied to many companies' scaling challenges. We've used Rick's story along with those of several other notable CTOs and entrepreneurs of successful Internet product companies ranging from startups to Fortune 500 companies to illustrate how important the rules in this book are to scaling high-transaction environments.

Quick Start Guide

Experienced engineers, architects, and managers can read through the header sections of all the rules that contain the what, when, how, and why. You can browse through each chapter to read these, or you can jump to Chapter 13, "Rule Review and Prioritization," which has a consolidated view of the headers. Once you've read these, go back to the chapters that are new to you or that you find more interesting.

For less experienced readers we understand that 50 rules can seem overwhelming. We do believe that you should eventually become familiar with all the rules, but we also understand that you need to prioritize your time. With that in mind, we have picked out five chapters for managers, five chapters for software developers, and five chapters for technical operations that we recommend you read before the others to get a jump start on your scalability knowledge.

Managers:

- Chapter 1, "Reduce the Equation"
- Chapter 2, "Distribute Your Work"
- Chapter 4, "Use the Right Tools"
- Chapter 7, "Learn from Your Mistakes"
- Chapter 12, "Miscellaneous Rules"

Software developers:

- Chapter 1, "Reduce the Equation"
- Chapter 2, "Distribute Your Work"
- Chapter 5, "Get Out of Your Own Way"
- Chapter 10, "Avoid or Distribute State"
- Chapter 11, "Asynchronous Communication and Message Buses"

Technical operations:

- Chapter 2, "Distribute Your Work"
- Chapter 3, "Design to Scale Out Horizontally"
- Chapter 6, "Use Caching Aggressively"
- Chapter 8, "Database Rules"
- Chapter 9, "Design for Fault Tolerance and Graceful Failure"

As you have time later, we recommend reading all the rules to familiarize yourself with the rules and concepts that we present no matter what your role. The book is short and can probably be read in a coast-to-coast flight in the United States.

After the first read, the book can be used as a reference. If you are looking to fix or re-architect an existing product, Chapter 13 offers an approach to applying the rules to your existing platform based on cost and the expected benefit (presented as a reduction of risk). If you already have your own prioritization mechanism, we do not recommend changing it for ours unless you like our approach better. If you don't have an existing method of prioritization, our method should help you think through which rules you should apply first.

If you are just starting to develop a new product, the rules can help inform and guide you as to best practices for scaling. In this case, the approach of prioritization represented in Chapter 13 can best be used as a guide to what's most important to consider in your design. You should look at the rules that are most likely to allow you to scale for your immediate and long-term needs and implement those.

For all organizations, the rules can serve to help you create a set of architectural principles to drive future development. Select the 5, 10, or 15 rules that will help your product scale best and use them as an augmentation of your existing design reviews. Engineers and architects can ask questions relevant to each of the scalability rules that you select and ensure that any new significant design meets your scalability standards. While these rules are as specific and fixed as possible, there is room for modification based on your system's particular criteria. If you or your team has extensive scalability experience, go ahead and tweak these rules as necessary to fit your particular scenario. If you and your team lack large-scale experience, use the rules exactly as is and see how far they allow you to scale.

Finally, this book is meant to serve as a reference and handbook. Chapter 13 is set up as a quick reference and summary of the rules. Whether you are experiencing problems or simply looking to develop a more scalable solution, Chapter 13 can be a quick reference guide to help pinpoint the rules that will help you out of your predicament fastest or help you define the best path forward in the event of new development. Besides using this as a desktop reference, also consider integrating this into your organization by one of many tactics such as taking one or two rules each week and discussing them at your technology all-hands meeting.

Why a Second Edition?

The first edition of *Scalability Rules* was the first book to address the topic of scalability in a rules-oriented fashion. Customers loved its brevity, ease of use, and convenience. But time and time again readers and clients of our firm, AKF Partners, asked us to tell the stories behind the rules. Because we pride ourselves in putting the needs of our clients first, we edited this book to include stories upon which the rules are based.

In addition to telling the stories of multiple CTOs and successful entrepreneurs, editing the book for a second edition allowed us to update the content to remain consistent with the best practices in our industry. The second edition also gave us the opportunity to subject our material to another round of technical peer reviews and production editing. All of this results in a second edition that's easier to read, easier to understand, and easier to apply.

How Does *Scalability Rules* Differ from *The Art of Scalability*?

The Art of Scalability, Second Edition (ISBN 0134032802, published by Addison-Wesley), our first book on the topic of scalability, focused on people, process, and technology, whereas *Scalability Rules* is predominantly a technically focused book. Don't get us wrong; we still believe that people and process are the most important components of building scalable solutions. After all, it's the organization, including both the individual contributors and the management, that succeeds or fails in producing scalable solutions. The technology isn't at fault for failing to scale—it's the people who are at fault for building it, selecting it, or integrating it. But we believe that *The Art of Scalability* adequately addresses the people and process concerns around scalability, and we wanted to go into greater depth on the technical aspects of scalability.

Scalability Rules expands on the third (technical) section of *The Art of Scalability*. The material in *Scalability Rules* is either new or discussed in a more technical fashion than in *The Art of Scalability*. As some reviewers on Amazon point out, *Scalability Rules* works well as both a standalone book and as a companion to *The Art of Scalability*.

Notes

1. "Net Sales Revenue of Amazon from 2004 to 2015,"

   www.statista.com/statistics/266282/annual-net-revenue-of-amazoncom/.

2. Walmart, Corporate and Financial Facts,

   http://corporate.walmart.com/_news_/news-archive/investors/2015/02/19/walmart-announces-q4-underlying-eps-of-161-and-additional-strategic-investments-in-people-e-commerce-walmart-us-comp-sales-increased-15-percent.

3. Authors' note: Famously known as Amazon's Two-Pizza Rule—no team can be larger than that which two pizzas can feed.

Register your copy of *Scalability Rules, Second Edition*, at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134431604) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Acknowledgments

The rules contained within this book weren't developed by our partnership alone. They are the result of nearly 70 years of work with clients, colleagues, and partners within nearly 400 companies, divisions, and organizations. Each of them contributed, in varying degrees, to some or all of the rules within this book. As such, we would like to acknowledge the contributions of our friends, partners, clients, coworkers, and bosses for whom or with which we've worked over the past several (combined) decades. The CTO stories from Rick Dalzell, Chris Lalonde, James Barrese, Lon Binder, Brad Peterson, Grant Klopper, Jeremy King, Tom Keeven, Tayloe Stansbury, Chris Schremser, and Chuck Geiger included herein are invaluable in helping to illustrate the need for these 50 rules. We thank each of you for your time, thoughtfulness, and consideration in telling us your stories.

We would also like to acknowledge and thank the editors who have provided guidance, feedback, and project management. The technical editors from both the first and second editions—Geoffrey Weber, Chris Lalonde, Camille Fournier, Jeremy Wright, Mark Urmacher, and Robert Guild—shared with us their combined decades of technology experience and provided invaluable insight. Our editors from Addison-Wesley, Songlin Qiu, Laura Lewin, Olivia Basegio, and Trina MacDonald, provided supportive stylistic and rhetorical guidance throughout every step of this project. Thank you all for helping with this project.

Last but certainly not least, we'd like to thank our families and friends who put up with our absence from social events as we sat in front of a computer screen and wrote. No undertaking of this magnitude is done single-handedly, and without our families' and friends' understanding and support this would have been a much more arduous journey.

This page intentionally left blank

About the Authors

Martin L. Abbott is a founding partner of AKF Partners, a growth consulting firm focusing on meeting the needs of today's fast-paced and hyper-growth companies. Marty was formerly the COO of Quigo, an advertising technology startup acquired by AOL in 2007. Prior to Quigo, Marty spent nearly six years at eBay, most recently as SVP of technology and CTO and member of the CEO's executive staff. Prior to eBay, Marty held domestic and international engineering, management, and executive positions at Gateway and Motorola. Marty has served on a number of boards of directors for public and private companies. He spent a number of years as both an active-duty and reserve officer in the US Army. Marty has a BS in computer science from the United States Military Academy, an MS in computer engineering from the University of Florida, is a graduate of the Harvard Business School Executive Education Program, and has a Doctorate of Management from Case Western Reserve University.

Michael T. Fisher is a founding partner of AKF Partners, a growth consulting firm focusing on meeting the needs of today's fast-paced and hyper-growth companies. Prior to cofounding AKF Partners, Michael held many industry roles including CTO of Quigo, acquired by AOL in 2007, and VP of engineering and architecture for PayPal. He served as a pilot in the US Army. Michael received a PhD and MBA from Case Western Reserve University's Weatherhead School of Management, an MS in information systems from Hawaii Pacific University, and a BS in computer science from the United States Military Academy (West Point). Michael is an adjunct professor in the Design and Innovation Department at Case Western Reserve University's Weatherhead School of Management.

This page intentionally left blank

1

Reduce the Equation

By nearly any measure, Jeremy King has had a successful and exciting career. In the mid-1990s, before the heyday of the Internet, Jeremy was involved in Bay Networks' award-winning SAP implementation. From there, Jeremy joined the dot-com boom as the VP of engineering at Petopia.com. Jeremy often jokes that he received his "real-world MBA" from the "University of Hard Knocks" during the dot-com bubble at Petopia. From Petopia, Jeremy joined eBay as the director of architecture for V3, eBay's then-next-generation commerce platform. If Petopia offered a lesson in economics and financing, eBay (where Jeremy later became a VP) offered an unprecedented education in scaling systems. Jeremy spent three years as the EVP of technology at LiveOps and is now the CTO at WalmartLabs.

eBay taught Jeremy many lessons, including the need for simplicity in architecture. For context, when Jeremy joined in 2001, eBay stood with rarefied companies like Amazon and Google at the extreme edge of online transaction processing (OLTP) and scale. For the full year of 2001, eBay recorded $2.735 billion[1] in gross merchandise sales as compared to Walmart's worldwide sales of $191.3 billion (online sales were not reported)[2] and Amazon's total sales of $3.12 billion.[3] Underneath this heady success, however, lay a dark past for eBay.

In June of 1999 eBay experienced a near death sentence because of an outage lasting almost 24 hours.[4] The eBay site continued to undergo outages of varying durations for months after the outage of June 1999, and although each was caused by different trigger events, all of them could be traced back to the inability of the site to scale to nearly unprecedented user growth. These outages changed the culture of the company forever. More specifically, they focused the company on attempting to set the standard for high availability and reliability of its service offering.

In 1999, eBay sold most of its merchandise using an auction format. Auctions are unique entities because, as compared to typical online commerce transactions, auction items tend to be short-lived and have an inordinately high volume of bids (write transactions) and views (read transactions) near the end of their expected duration. Most items for sale in traditional platforms have a relatively flat number of transactions through their life with the typical seasonal peaks, whereas eBay had millions of items for sale, and all of the user activity would be directed at a fraction of those items at any given time. This represented a unique problem as database load, for instance, would be primarily on a small number of items, and the database (then a monolith) that supported eBay would struggle with physical and logical contention on these items. This in turn could manifest itself as site slowness and even complete outages.

Jeremy's first eBay job was to lead a team to redefine eBay's software architecture with the goal of keeping incidents like the June 1999 outage from happening again. This was a task made even more complex by the combination of eBay's meteoric growth and the difficulty of the auction format. The project, internally named V3, was a reimplementation of the eBay commerce engine in Java (the prior implementation was C++) and a re-architecture of the site to allow for multiple "sharded" databases along the X, Y, and Z axes described in Chapter 2, "Distribute Your Work."

The team approached every component with a maniacal focus, attempting to ensure that everything could be nearly infinitely scaled and that any failure could be resolved quickly with minimal impact. "My primary lesson learned," Jeremy indicated, "was that we treated everything as if it had the same complexity and business criticality as the actual bidding process on an auction. Absolutely everything from the display of images to eBay's reputation [often called Feedback] system on the site was treated similarly with similar fault tolerance.

"It's important to remember," continued Jeremy, "that this was 2001 and that there were very few companies experiencing our growth at our size online. As such, we really couldn't go anywhere—whether to a vendor or an open-source solution—to solve our problems. We had to invent everything ourselves, something I'd really prefer not to do."

At first glance, it is difficult to tease out the lesson in Jeremy's comments. So what if everything was built in the same bulletproof fashion as an auction? "Well," said Jeremy with a laugh, "not everything is as complex as an auction. Let's take the reputation engine, for instance. It doesn't have the same competition over short periods of time as auctions. As such, it really doesn't need to have the same principles applied to it. The system simply scales more cost-effectively and potentially is as highly available, if you take an approach that recognizes you don't have the same level of transactional competition on subsets of data over short periods of time. More importantly, Feedback has one write transaction per item sold, whereas an auction may have hundreds of attempted write transactions in a second on a single item. This isn't the same as decrementing inventory; it's a complex comparison of bid price to all other bids and then the calculation of a 'second price.' But we treated that and everything else as if we were solving the same problem as auctions—specifically, to be able to scale under incredible demand, for small subsets of data, much of it happening in the last few seconds of the life of an auction."

That being clear, we wondered what the impact of making some pieces of V3 more complex than others might be. "That's easy," said Jeremy. "While V3 overall was a success, I think in hindsight we could have potentially completed it faster or cheaper or both. Moreover, because some aspects of it were overly complex or alternatively not as simple as the problem demanded, the maintenance costs of those aspects are higher. Contrast this approach with an architecture principle I've since learned and applied at both Walmart and LiveOps: match the effort and approach to the complexity of the problem. Not every solution has the same complexity—take the simplest approach to achieve the desired outcome. While having a standard platform or language can seem

desirable from a scaling, maintainability, or reuse aspect, taking the simple approach on a new open-source project, language, or platform can vastly change the cost, time to market, or innovation you can deliver for your customers."

Jeremy's story is about not making things harder than they need to be, or putting it another way, keeping things as simple as possible. Our view is that a complex problem is really just a collection of smaller, simpler problems waiting to be solved. This chapter is about making big problems small, and in so doing achieving big results with less work.

As is the case with many of the chapters in *Scalability Rules*, the rules here vary in size and complexity. Some are overarching rules easily applied to several aspects of our design. Some rules are very granular and prescriptive in their implementation to specific systems.

Rule 1—Don't Overengineer the Solution

> **Rule 1: What, When, How, and Why**
>
> **What:** Guard against complex solutions during design.
>
> **When to use:** Can be used for any project and should be used for all large or complex systems or projects.
>
> **How to use:** Resist the urge to overengineer solutions by testing ease of understanding with fellow engineers.
>
> **Why:** Complex solutions are excessively costly to implement and are expensive to maintain long term.
>
> **Key takeaways:** Systems that are overly complex limit your ability to scale. Simple systems are more easily and cost-effectively maintained and scaled.

As is explained in its Wikipedia entry, overengineering falls into two broad categories.[5] The first category covers products designed and implemented to exceed their useful requirements. We discuss this problem briefly for completeness, but in our estimation its impact on scale is small compared to that of the second problem. The second category of overengineering covers products that are made to be overly complex. As we earlier implied, we are most concerned about the impact of this second category on scalability. But first, let's address the notion of exceeding requirements.

To explain the first category of overengineering, exceeding useful requirements, we must first make sense of the term *useful*, which here means simply "capable of being used." For example, designing an HVAC unit for a family house that is capable of heating that house to 300 degrees Fahrenheit in outside temperatures of 0 Kelvin simply has no use for us anywhere. The effort necessary to design and manufacture such a solution is wasted as compared to a solution that might heat the house to a comfortable living temperature in environments where outside temperatures might get close to –20 degrees Fahrenheit. This type of overengineering might have cost overrun elements, including a higher cost to develop (engineer) the solution and a higher cost to implement the

solution in hardware and software. It may further impact the company by delaying the product launch if the overengineered system took longer to develop than the useful system. Each of these costs has stakeholder impact as higher costs result in lower margins, and longer development times result in delayed revenue or benefits. *Scope creep*, or the addition of scope between initial product definition and initial product launch, is one manifestation of overengineering.

An example closer to our domain of experience might be developing an employee time card system capable of handling a number of employees for a single company that equals or exceeds 100 times the population of Planet Earth. The probability that the Earth's population will increase 100-fold within the useful life of the software is tiny. The possibility that all of those people would work for a single company is even smaller. We certainly want to build our systems to scale to customer demands, but we don't want to waste time implementing and deploying those capabilities too far ahead of our need (see Rule 2).

The second category of overengineering deals with making something overly complex and making something in a complex way. Put more simply, the second category consists of either making something work harder to get a job done than is necessary, making a user work harder to get a job done than is necessary, or making an engineer work harder to understand something than is necessary. Let's dive into each of these three areas of overly complex systems.

What does it mean to make something work harder than is necessary? Jeremy King's example of building all the features constituting eBay's site to the demanding requirements of the auction bidding process is a perfect example of making something (e.g., the Feedback system) work harder than is necessary. Some other examples come from the real world. Imagine that you ask your significant other to go to the grocery store. When he agrees, you tell him to pick up one of everything at the store, and then to pause and call you when he gets to the checkout line. Once he calls, you will tell him the handful of items that you would like from the many baskets of items he has collected, and he can throw everything else on the floor. "Don't be ridiculous!" you might say. But have you ever performed a `select (*) from schema_name.table_name` SQL statement within your code only to cherry-pick your results from the returned set (see Rule 35 in Chapter 8, "Database Rules")? Our grocery store example is essentially the same activity as the `select (*)` case. How many lines of conditionals have you added to your code to handle edge cases and in what order are they evaluated? Do you handle the most likely case first? How often do you ask your database to return a result set you just returned, and how often do you re-create an HTML page you just displayed? This particular problem (doing work repetitively when you can just go back and get your last correct answer) is so rampant and easily overlooked that we've dedicated an entire chapter (Chapter 6, "Use Caching Aggressively") to this topic! You get the point.

What do we mean by making a user work harder than is necessary? In many cases, less is more. Many times in the pursuit of trying to make a system flexible, we strive to cram as many odd features as possible into it. Variety is not always the spice of life.

Many times users just want to get from point A to point B as quickly as possible without distractions. If 99% of your market doesn't care about being able to save their blog as a .pdf file, don't build in a prompt asking them if they'd like to save it as a .pdf. If your users are interested in converting .wav files to MP3 files, they are already sold on a loss of fidelity, so don't distract them with the ability to convert to lossless compression FLAC files.

Finally, we come to the far-too-common problem of making software too complex for other engineers to easily and quickly understand. Back in the day it was all the rage, and in fact there were competitions, to create complex code that would be difficult for others to understand. Medals were handed out to the person who could develop code that would bring senior developers to tears of acquiescence within code reviews. Complexity became the intellectual cage within which geeky code slingers would battle for organizational dominance. For those interested in continuing in the geek fest, but in a "safe room" away from the potential stakeholder value destruction of doing it "for real," we suggest you partake in the International Obfuscated C Code Contest at www0.us.ioccc.org/index.html. For everyone else, recognize that your job is to develop simple, easy-to-understand solutions that are easy to maintain and create shareholder value.

We should all strive to write code that everyone can understand. The real measure of a great engineer is how quickly that engineer can simplify a complex problem (see Rule 3) and develop an easily understood and maintainable solution. Easy-to-follow solutions allow less-senior engineers to more quickly come up to speed to support systems. Easy-to-understand solutions mean that problems can be found earlier during troubleshooting, and systems can be restored to their proper working order faster. Easy-to-follow solutions increase the scalability of your organization and your solution.

A great test to determine whether something is too complex is to have the engineer in charge of solving a given complex problem present his or her solution to several engineering cohorts within the company. The cohorts should represent different engineering experience levels as well as varying tenures within the company (we make a distinction here because you might have experienced engineers with very little company experience). To pass this test, each of the engineering cohorts should easily understand the solution, and each cohort should be able to describe the solution, unassisted, to others not otherwise knowledgeable about it. If any cohort does not understand the solution, the team should debate whether the system is overly complex.

Overengineering is one of the many enemies of scale. Developing a solution beyond that which is useful simply wastes money and time. It may further waste processing resources, increase the cost of scale, and limit the overall scalability of the system (how far that system can be scaled). Building solutions that are overly complex has a similar effect. Systems that work too hard increase your cost and limit your ultimate size. Systems that make users work too hard limit how quickly you are likely to increase the number of users and therefore how quickly you will grow your business. Systems that are too complex to understand kill organizational productivity and the ease with which you can add engineers or add functionality to your system.

Rule 2—Design Scale into the Solution (D-I-D Process)

Rule 2: What, When, How, and Why

What: An approach to provide JIT (just-in-time) scalability.

When to use: On all projects; this approach is the most cost-effective (resources and time) to ensure scalability.

How to use:

- Design for 20x capacity.
- Implement for 3x capacity.
- Deploy for roughly 1.5x capacity.

Why: D-I-D provides a cost-effective, JIT method of scaling your product.

Key takeaways: Teams can save a lot of money and time by thinking of how to scale solutions early, implementing (coding) them a month or so before they are needed, and deploying them days before the customer rush or demand.

Our firm is focused on helping clients address their scalability needs. As you might imagine, customers often ask us, "When should we invest in scalability?" The somewhat flippant answer is that you should invest (and deploy) the day before the solution is needed. If you could deploy scalability improvements the day before you needed them, your investments would be "just in time" and this approach would help maximize firm profits and shareholder wealth. This is similar to what Dell brought to the world with configure-to-order systems combined with just-in-time manufacturing.

But let's face it—timing such an investment and deployment "just in time" is simply impossible, and even if possible it would incur a great deal of risk if you did not nail the date exactly. The next best thing to investing and deploying "the day before" is AKF Partners' *Design-Implement-Deploy* or *D-I-D* approach to thinking about scalability. These phases match the cognitive phases with which we are all familiar: starting to think about and designing a solution to a problem, building or coding a solution to that problem, and actually installing or deploying the solution to the problem. This approach does not argue for nor does it need a waterfall model. We argue that agile methodologies abide by such a process by the very definition of the need for human involvement. You cannot develop a solution to a problem of which you are not aware, and a solution cannot be manufactured or released if it is not developed. Regardless of the development methodology (agile, waterfall, hybrid, or whatever), everything we develop should be based on a set of architectural principles and standards that define and guide what we do.

Design

We start with the notion that discussing and designing something are both significantly less expensive than actually implementing that design in code. Given this relatively

Table 1.1 **D-I-D Process for Scale**

	Design	Implement	Deploy
Scale objective	20x to infinite	3x to 20x	1.5x to 3x
Intellectual cost	High	Medium	Low to medium
Engineering cost	Low	High	Medium
Asset cost	Low	Low to medium	High to very high
Total cost	Low to medium	Medium	Medium

low cost, we can discuss and sketch out a design for how to scale our platform well in advance of our need. For example, we clearly would not want to deploy 10x, 20x, or 100x more capacity than we would need in our production environment. However, the cost of discussing and deciding how to scale something to those dimensions is comparatively small. The focus then in the (D)esign phase of the D-I-D scale model is on scaling to between 20x and infinity. Our intellectual costs are high as we employ our "big thinkers" to think through the "big problems." Engineering and asset costs, however, are low as we aren't writing code or deploying costly systems. Scalability summits, a process in which groups of leaders and engineers gather to discuss scale-limiting aspects of a product, are a good way to identify the areas necessary to scale within the design phase of the D-I-D process. Table 1.1 lists the phases of the D-I-D process.

Implement

As time moves on, and as our perceived need for future scale draws near, we move to (I)mplement our designs within our software. We reduce our scope in terms of scale needs to something that's more realistic, such as 3x to 20x our current size. We use "size" here to identify that element of the system that is perceived to be the greatest bottleneck of scale and therefore in the greatest need of modification to achieve our business results. There may be cases where the cost of scaling 100x (or greater) our current size is not different from the cost of scaling 20x. If this is the case, we might as well make those changes once rather than going in and making changes multiple times. This might be the case if we are going to perform a modulus of our user base to distribute (or share) users across multiple (N) systems and databases. We might code a variable Cust_MOD that we can configure over time between 1 (today) and 1,000 (five years from now). The engineering (or implementation) cost of such a change really doesn't vary with the size of N, so we might as well make Cust_MOD capable of being as large as possible. The cost of these types of changes is high in terms of engineering time, medium in terms of intellectual time (we already discussed the designs earlier in our lifecycle), and low in terms of assets as we don't need to deploy 100x our systems today if we intend to deploy a modulus of 1 or 2 in our first phase.

Deploy

The final phase of the D-I-D process is (D)eploy. Using our modulus example, we want to deploy our systems in a just-in-time fashion; there's no reason to have idle assets diluting shareholder value. Maybe we put 1.5x our peak capacity in production if we are a moderately high-growth company and 5x our peak capacity in production if we are a hyper-growth company. We often guide our clients to leverage the cloud for burst capacity so that we don't have 33% of our assets waiting around for a sudden increase in user activity. Asset costs are high in the deployment phase, and other costs range from low to medium. Total costs tend to be highest for this category because to deploy 100x necessary capacity relative to demand would kill many companies. Remember that scale is an elastic concept; it can both expand and contract, and our solutions should recognize both aspects. Therefore, flexibility is key, because you may need to move capacity around as different systems within your solution expand and contract in response to customer demand.

Designing and thinking about scale come relatively cheaply and thus should happen frequently. Ideally these activities result in some sort of written documentation so that others can build upon it quickly should the need arise. Engineering (or developing) the architected or designed solutions can happen later and cost a bit more overall, but there is no need to actually implement them in production. We can roll the code and make small modifications as in our modulus example without needing to purchase 100x the number of systems we have today. Finally, the process lends itself nicely to purchasing equipment just ahead of our need, which might be a six-week lead time from a major equipment provider or having one of our systems administrators run down to the local server store in extreme emergencies. Obviously, in the case of infrastructure as a service (IaaS, aka cloud) environments, we do not need to purchase capacity in advance of need and can easily "spin up" compute assets for the deploy phase on a near-as-needed and near-real-time basis.

Rule 3—Simplify the Solution Three Times Over

> **Rule 3: What, When, How, and Why**
>
> **What:** Used when designing complex systems, this rule simplifies the scope, design, and implementation.
>
> **When to use:** When designing complex systems or products where resources (engineering or computational) are limited.
>
> **How to use:**
> - Simplify scope using the Pareto Principle.
> - Simplify design by thinking about cost effectiveness and scalability.
> - Simplify implementation by leveraging the experience of others.
>
> **Why:** Focusing just on "not being complex" doesn't address the issues created in requirements or story and epoch development or the actual implementation.
>
> **Key takeaways:** Simplification needs to happen during every aspect of product development.

Whereas Rule 1 dealt with avoiding surpassing the "usable" requirements and eliminating complexity, this rule addresses taking another pass at simplifying everything from your perception of your needs through your actual design and implementation. Rule 1 is about fighting against the urge to make something overly complex, and Rule 3 is about attempting to further simplify the solution by the methods described herein. Sometimes we tell our clients to think of this rule as "asking the three hows": How do I simplify my scope, my design, and my implementation?

How Do I Simplify My Scope?

The answer to this question of simplification is to apply the Pareto Principle (also known as the 80-20 rule) frequently. What 80% of your benefit is achieved from 20% of the work? In our case, a direct application is to ask, "What 80% of your revenue will be achieved by 20% of your features?" Doing significantly less (20% of the work) while achieving significant benefits (80% of the value) frees up your team to perform other tasks. If you cut unnecessary features from your product, you can do five times as much work, and your product will be significantly less complex! With four-fifths fewer features, your system will no doubt have fewer dependencies between functions and as a result will be able to scale both more efficiently and more cost-effectively. Moreover, the 80% of your time that is freed up can be used to launch new product offerings as well as invest in thinking ahead to the future scalability needs of your product.

We're not alone in our thinking on how to reduce unnecessary features while keeping a majority of the benefit. The folks at 37signals, now rebranded as Basecamp, are huge proponents of this approach, discussing the need and opportunity to prune work in both their book *Rework*[6] and in their blog post titled "You Can Always Do Less."[7] Indeed, the concept of the "minimum viable product" popularized by Eric Reis and evangelized by Marty Cagan is predicated on the notion of maximizing the "amount of validated learning about customers with the least effort."[8] This "agile" focused approach allows us to release simple, easily scalable products quickly. In so doing we get greater product throughput in our organizations (organizational scalability) and can spend additional time focusing on building the minimal product in a more scalable fashion. By simplifying our scope, we have more computational power because we are doing less. If you don't believe us, go back and read Jeremy King's story and his lessons learned. Had the eBay team reduced the scope of features like Feedback, the V3 project would have been delivered sooner, at lower cost, and for relatively the same value to the end consumer.

How Do I Simplify My Design?

With this new, smaller scope, the job of simplifying our implementation just became easier. Simplifying design is closely related to the complexity aspect of overengineering. Complexity elimination is about cutting off unnecessary trips in a job, and simplification is about finding a shorter path. In Rule 1, we gave the example of asking a database only for that which you need; `select(*) from schema_name.table_name` became `select (column) from schema_name.table_name`. The approach of design simplification suggests