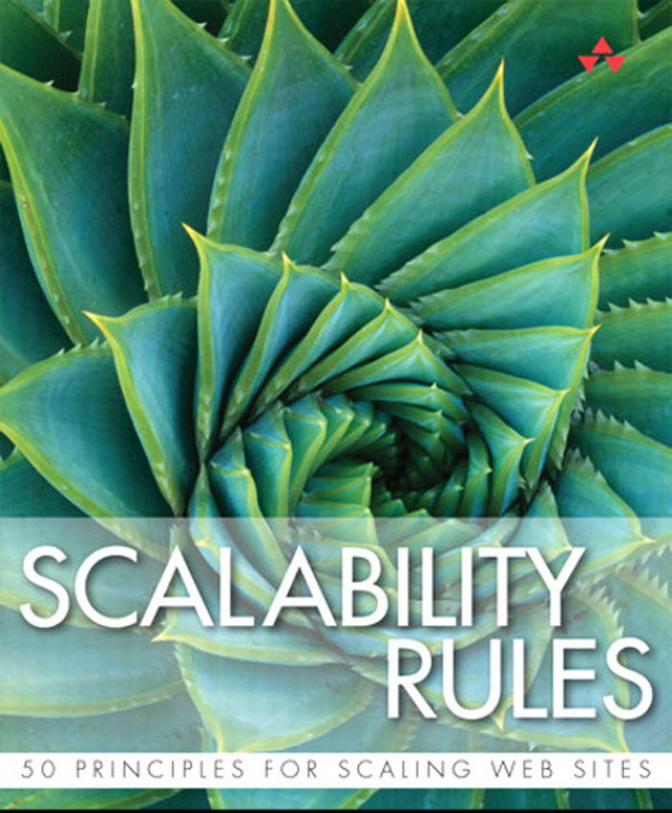

SCALABILITY RULES

50 PRINCIPLES FOR SCALING WEB SITES

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 *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**, VP, Technical Operations and Infrastructure Architecture, Bullhorn

"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**, Vice President, 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**, Chief Technology Officer, ShareThis

"For organizations looking to scale technology, people, and processes rapidly or effectively, the twin pairing of *Scalability Rules* and *The Art of Scalability* are unbeatable. The rules-driven approach in *Scalability Rules* makes this not only 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

50 Principles for Scaling Web Sites

Martin L. Abbott
Michael T. Fisher

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data:

Abbott, Martin L.
Scalability rules : 50 principles for scaling Web sites / Martin L. Abbott, Michael T. Fisher.
p. cm.
ISBN 978-0-321-75388-5 (pbk. : alk. paper) — ISBN (invalid) 01321753887 (pbk. : alk. paper) 1. Computer networks—Scalability. 2. Web sites—Security measures. I. Fisher, Michael T. II. Title.
TK5105.59.A23 2011
006.7—dc22

2011006257

ISBN-13: 978-0-321-75388-5
ISBN-10: 0-321-75388-7

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing May 2011

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Geneil Breeze

Indexer
Erika Millen

Proofreader
Linda Seifert

Technical Reviewers
Robert Guild
Geoffrey Weber
Jeremy Wright

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

❖

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 at a Glance

Preface

Thanks for your interest in *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, while 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, while others are specific and may explain a technique, such as how to modify headers to maximize the "cache-ability" of content.

Quick Start Guide

For experienced engineers, architects, and managers read through the header sections of all the rules that contain the what, when, how, and why. You can browse through each chapter reading these, or you can jump to Chapter 13, "Rule Review and Prioritization," which has a consolidated view of these 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, "Don't Duplicate Your Work"
- 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 US.

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, "Rule Review and Prioritization," 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, then 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 to 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 are lacking large scale experience use them 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 become 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 Write Another Book on Scale?

There simply aren't many good books on scalability on the market yet, and *Scalability Rules* is unique in its approach in this sparse market. It is the first book to address the topic of scalability in a rules-oriented fashion. It is the first book meant to be as much of a reference as it is an overview of the topic. It includes a chapter summarizing the 50 rules and gives a prioritization of the rules for those looking to apply the book to their existing platforms.

One of our most-commented-on blog posts is on the need for scalability to become a discipline. We and the community of technologists that tackle scalability problems believe that scalability architects are needed in today's technology organizations. In

the early days of computer systems almost everyone involved was a programmer, and then came specialization with system operators, DBAs, architects, and so on. We now have many different disciplines and specialties that make up our technology teams. One of the missing disciplines is the scalability architect.

Unlike a DBA, whose job is to get things done and not necessarily teach someone else unless they are mentoring a junior DBA, one of the primary responsibilities of the scalability architect would be to educate technology people. The scalability architects should be evangelists and teachers rather than the gatekeepers of secret knowledge. As part of that teaching we've made a step forward by putting together these 50 rules that we believe will help guide any organization in scaling its systems.

How Did You Decide What 50 Rules to Include?

The decision of which rules to include wasn't easy. This could easily be a book of 100 or even 200 rules. Our criteria for inclusion was to look at the recommendations that we make most often to our client base and find the most commonly recommended changes, additions, or modifications to their products. When we looked at these rules, we saw a fairly sharp drop-off in the rate of recommendations after the first 50 rules. That's not to say that we made these 50 recommendations equally, or that the 51st potential rule wasn't also fairly common. Rather, these 50 were just recommended more often with our clients. The rules aren't presented in order of frequency of recommendation. In Chapter, 13 we group the rules by their benefit and priority based on how we ranked each rule's risk reduction and cost of implementation or adoption.

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

The Art of Scalability (ISBN: 0137030428, published by Addison-Wesley), our first book on this topic, focused on people, process, and technology, while *Scalability Rules* is predominately a technically focused book. Don't get us wrong, we still believe that

people and process are the most important component of building scalable solutions. After all, it's the organization, including both the individual contributors and the management, which 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 in greater depth on the technical aspects of scalability.

Scalability Rules expands on the third (technical) section of our first book. The material in *Scalability Rules* is either new or discussed in a more technical fashion than in *The Art of Scalability*. Where we discussed something in that book, we expand upon it or define it in a slightly different way to help readers better understand the concept.

Acknowledgments

The rules contained within this book weren't developed by our partnership alone. They are the result of nearly 60 years of work with clients, colleagues, and partners within nearly 200 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.

We would also like to acknowledge and thank the editors who have provided guidance, feedback, and project management. Our technical editors Geoffrey Weber, Jeremy Wright, and Robert Guild shared with us their combined decades of technology experience and provided invaluable insight. Our editors from Addison-Wesley, Songlin Qiu 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 to sit in front of a computer screen and write. 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.

About the Authors

Martin L. Abbott is an executive with experience running technology and business organizations within Fortune 500 and startup companies. He is a founding partner of AKF Partners, a consulting firm focusing on meeting the technical and business hyper growth needs of today's fast-paced companies. Marty was formerly the COO of Quigo, an advertising technology startup acquired by AOL in 2007, where he was responsible for product strategy, product management, technology development, advertising, and publisher services. 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 serves on the boards of directors for OnForce, LodgeNet Interactive (NASD:LNET), and Bullhorn. He sits on a number of advisory boards for universities and public and private companies. 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 is pursuing a Doctorate of Management from Case Western Reserve University. His current research investigates the antecedents and effects of conflict within executive teams of startups.

Michael T. Fisher is a veteran software and technology executive with experience in both Fortune 500 and startup companies. "Fish" is a founding partner of AKF Partners, a consulting firm focusing on meeting the technical and business hyper growth needs of today's fast-paced companies. Michael's experience includes two years as the chief technology officer of Quigo, a startup Internet advertising company acquired by AOL in 2007. Prior to Quigo, Michael served as vice president of engineering & architecture for PayPal, Inc., an eBay company. Prior to joining PayPal, Michael spent seven years at General Electric helping to develop the company's technology strategy and processes. Michael served six years as a captain and pilot in the

US Army. He sits on a number of boards of directors and advisory boards for private and nonprofit companies. Michael has a BS in computer science from the United States Military Academy, an MSIS from Hawaii Pacific University, a Ph.D. in Information Systems from Kennedy-Western University, and an MBA from Case Western Reserve University. Michael is a certified Six Sigma Master Black Belt and is pursuing a Doctorate of Management from Case Western Reserve University. His current research investigates the drivers for the viral growth of digital services.

This page intentionally left blank

1

Reduce the Equation

We've all been there at some point in our academic or professional careers: We stare at a complex problem and begin to lose hope. Where do we begin? How can we possibly solve the problem within the allotted time? Or in the extreme case—how do we solve it within a single lifetime? There's just too much to do, the problem is too complex, and it simply can't be solved. That's it. Pack it in. Game over…

Hold on—don't lose hope! Take a few deep breaths and channel your high school or college math teacher/professor. If you have a big hairy architectural problem, do the same thing you would do with a big hairy math equation and reduce it into easily solvable parts. Break off a small piece of the problem and break it into several smaller problems until each of the problems is easily solvable!

Our view is that any big problem, if approached properly, is really just a collection of smaller problems waiting to be solved. This chapter is all about making big architectural problems smaller and doing less work while still achieving the necessary business results. In many cases this approach actually reduces (rather than increases) the amount of work necessary to solve the problem, simplify the architecture and the solution, and end up with a much more scalable solution or platform.

As is the case with many of the chapters in *Scalability Rules*, the rules 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 costly to implement and have excessive long-term costs.
>
> **Key takeaways:** Systems that are overly complex limit your ability to scale. Simple systems are more easily and cost effectively maintained and scaled.

As Wikipedia explains, overengineering falls into two broad categories.[1] The first category covers products designed and implemented to exceed the useful requirements of the product. We discuss this problem briefly for completeness, but in our estimation its impact to scale is small compared to 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 to scalability. But first, let's address the notion of exceeding requirements.

To explain the first category of overengineering, the exceeding of 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 timecard 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 increases 100-fold within the useful life of the software is tiny. The possibility that all of those people 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? Some of the easiest 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)? Our grocery store example is essentially the same activity as the select (`*`) case

above. 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? The answer to this one is really pretty simple. 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 notion of making software complex to understand for other engineers. 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. Sometimes this complex code would serve a purpose—it would run faster than code developed by the average engineer. More often than not the code complexity (in terms of ability to understand what it was doing due rather than a measure like cyclomatic complexity) would simply be an indication of one's "brilliance" or mastery of "kung fu." Medals were handed out for 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. It was a great game for those involved, but companies and shareholders were the ones paying for the tickets for a cage match no one cares about. 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/main.html.

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 mean that less senior engineers can more quickly come up to speed to support systems. Easy to understand solutions mean that problems can be found more quickly during troubleshooting, and systems can be restored to their proper working order in a faster manner. 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 difference 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 the solution. 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 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 ~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 implementing them days before the customer rush or demand.

Our firm is focused on helping clients through their scalability needs, and 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 scale improvements the day before you needed them, you would delay investments to be "just in time" and gain the benefits that Dell brought to the world with configure-to-order systems married with just in time manufacturing. In so doing you would maximize firm profits and shareholder wealth.

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. One cannot develop a solution to a problem of which they 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 is significantly less expensive than actually implementing that design in code. Given this relatively low cost we can discuss and sketch out a design for how to scale our platform well in advance of our need. Whereas we clearly would not want to put 10x, 20x, or 100x more capacity than we would need in our production environment, the cost of discussing 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 the 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 parts of the D-I-D process.

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/Medium	Medium	Medium

Implement

As time moves on, and as our perceived need for future scale draws near, we move to (I)mplementing 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 for scalability. There may be cases where the cost of scaling 100x (or greater) our current size is not different than the cost of scaling 20x, and if this is the case we might as well make those changes once rather than going in and making those changes multiple times. This might be the case if we are going to perform a modulus of our user base to distribute (or share) them 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 (5 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 it. The cost of these types of changes are 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.

Deployment

The final phase of the D-I-D process is (D)eployment. Using our modulus example above, we want to deploy our systems in a just in time fashion; there's no reason to have idle assets sitting around diluting shareholder value. Maybe we put 1.5x of 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 as to deploy 100x of your 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 of scale. Flexibility is therefore key as you may need to move capacity around as different systems within your solution expand and contract to customer demand.

Referring to Table 1.1, we can see that while each phase of the D-I-D process has varying intellectual, engineering, and asset costs, there is a clear progression of overall cost to the company. Designing and thinking about scale comes 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 above 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.

Rule 3—Simplify the Solution 3 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 discusses 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 3 how's." How do I simplify my scope, my design, and my implementation?

How Do I Simplify the 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) and 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 5x as much work, and your product would be significantly less complex! With 4/5ths fewer features, your system will no doubt have fewer dependencies between functions and as a result will be able to scale both more effectively and cost effectively. Moreover, the 80% of the time that is freed up can be used to both 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 are huge proponents of this approach, discussing the need and opportunity to prune work in both their book *Rework*[2] and in their blog post titled "You Can Always Do Less."[3] 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."[4] 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 as we are doing less.

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 only asking a database 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 that we first look to see if we already have the information being requested within a local shared resource like local memory. Complexity elimination is about doing less work, and design simplification is about doing that work faster and easier.

Imagine a case where we are looking to read some source data, perform a computation on intermediate tokens from this source data, and then bundle up these tokens. In many cases, each of these verbs might be broken into a series of services. In fact, this approach looks similar to that employed by the popular "map-reduce" algorithm. This approach isn't overly complex, so it doesn't violate Rule 1. But if we know that files to be read are small and we don't need to combine tokens across files, it might make sense to take the simple path of making this a simple monolithic application rather than decomposing it into services. Going back to our timecard example, if the goal is simply to compute hours for a single individual it makes sense to have multiple cloned monolithic applications reading a queue of timecards and performing the computations. Put simply, the step of design simplification asks us how to get the job done in an easy to understand, cost-effective, and scalable way.

How Do I Simplify My Implementation?

Finally, we get to the question of implementation. Consistent with Rule 2—the D-I-D Process for Scale, we define an implementation as the actual coding of a solution. This is where we get into questions such as whether it makes more sense to solve a problem with recursion or iteration. Should we define an array of a certain size, or be prepared to allocate memory dynamically as we need it? Do I make the solution, open-source the solution, or buy it? The answers to all these solutions have a consistent theme: "How can we leverage the experiences of others and existing solutions to simplify our implementation?"

Given that we can't be the best at building everything, we should first look to find widely adopted open source or third-party solutions to meet our needs. If those don't exist, we should look to see if someone within our own organization has developed a scalable solution to solve the problem. In the absence of a proprietary solution, we should again look externally to see if someone has described a scalable approach to solve the problem that we can legally copy or mimic. Only in the absence of finding one of these three things should we embark on attempting to solve the solution ourselves. The simplest implementation is almost always one that has already been implemented and proven scalable.

Rule 4—Reduce DNS Lookups

> **Rule 4: What, When, How, and Why**
>
> **What:** Reduce the number of DNS lookups from a user perspective.
>
> **When to use:** On all Web pages where performance matters.
>
> **How to use:** Minimize the number of DNS lookups required to download pages, but balance this with the browser's limitation for simultaneous connections.
>
> **Why:** DNS lookups take a great deal of time, and large numbers of them can amount to a large portion of your user experience.
>
> **Key takeaways:** Reduction of objects, tasks, computation, and so on is a great way of speeding up page load time, but division of labor must be considered as well.

As we've seen so far in this chapter, reducing is the name of the game for performance improvements and increased scalability. A lot of rules are focused on the architecture of the Software as a Service (SaaS) solution, but for this rule let's consider your customer's browser. If you use any of the browser level debugging tools such as Mozilla Firefox's plug-in Firebug,[5] you'll see some interesting results when you load a page from your application. One of the things you will most likely notice is that similarly sized objects on your page take different amounts of time to download. As you look closer you'll see some of these objects have an additional step at the beginning of their download. This additional step is the DNS lookup.

The Domain Name System (DNS) is one of the most important parts of the infrastructure of the Internet or any other network that utilizes the Internet Protocol Suite (TCP/IP). It allows the translation from domain name (www.akfpartners.com) to an IP address (184.72.236.173) and is often analogized to a phone book. DNS is maintained by a distributed database system, the nodes of which are the name servers. The top of the hierarchy consists of the root name servers. Each domain has at least one authoritative DNS server that publishes information about that domain.

This process of translating domains into IP addresses is made quicker by caching on many levels, including the browser, computer operating system, Internet service provider, and so on. However, in our world where pages can have hundreds or thousands of objects, many from different domains, small milliseconds of time can add up to something noticeable to the customer.

Before we go any deeper into our discussion of reducing the DNS lookups we need to understand at a high level how most browsers download pages. This isn't meant to be an in-depth study of browsers, but understanding the basics will help you optimize your application's performance and scalability. Browsers take advantage of the fact that almost all Web pages are comprised of many different objects (images, JavaScript files, css files, and so on) by having the ability to download multiple objects through simultaneous connections. Browsers limit the maximum number of simultaneous persistent connections per server or

proxy. According to the HTTP/1.1 RFC[6] this maximum should be set to 2; however, many browsers now ignore this RFC and have maximums of 6 or more. We'll talk about how to optimize your page download time based on this functionality in the next rule. For now let's focus on our Web page broken up into many objects and able to be downloaded through multiple connections.

Every distinct domain that serves one or more objects for a Web page requires a DNS lookup either from cache or out to a DNS name server. For example, let's assume we have a simple Web page that has four objects: 1) the HTML page itself that contains text and directives for other objects, 2) a CSS file for the layout, 3) a JavaScript file for a menu item, and 4) a JPG image. The HTML comes from our domain (akfpartners.com), but the CSS and JPG are served from a subdomain (static.akfpartners.com), and the JavaScript we've linked to from Google (ajax.googleapis.com). In this scenario our browser first receives the request to go to page www.akfpartners.com, which requires a DNS lookup of the akfpartners.com domain. Once the HTML is downloaded the browser parses it and finds that it needs to download the CSS and JPG both from static.akfpartners.com, which requires another DNS lookup. Finally, the parsing reveals the need for an external JavaScript file from yet another domain. Depending on the freshness of DNS cache in our browser, operating system, and so on, this lookup can take essentially no time up to hundreds of milliseconds. Figure 1.1 shows a graphical representation of this.

As a general rule, the fewer DNS lookups on your pages the better your page download performance will be. There is a downside to combining all your objects into a single domain, and we've hinted at the reason in the previous discussion about maximum simultaneous connects. We explore this topic in more detail in the next rule.

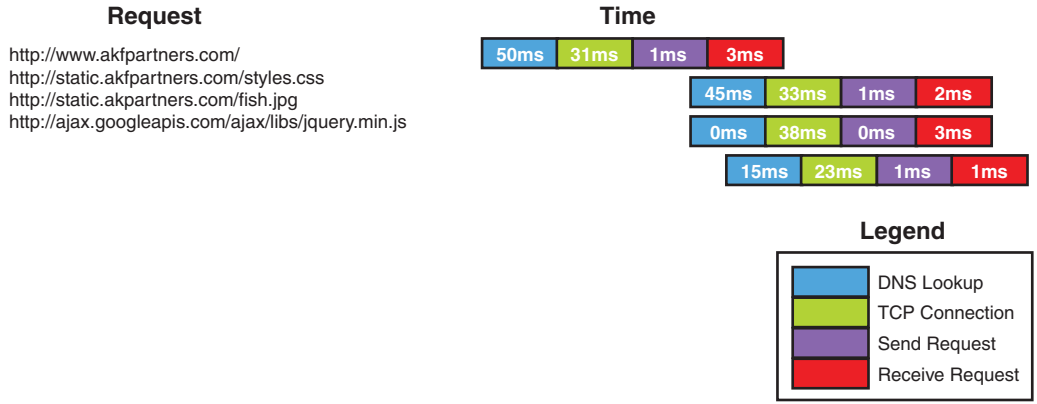

Figure 1.1 Object download time

Rule 5—Reduce Objects Where Possible

> **Rule 5: What, When, How, and Why**
>
> **What:** Reduce the number of objects on a page where possible.
>
> **When to use:** On all web pages where performance matters.
>
> **How to use:**
>
> - Reduce or combine objects but balance this with maximizing simultaneous connections.
> - Test changes to ensure performance improvements.
>
> **Why:** The number of objects impacts page download times.
>
> **Key takeaways:** The balance between objects and methods that serve them is a science that requires constant measurement and adjustment; it's a balance between customer usability, usefulness, and performance.

Web pages consist of many different objects (HTML, CSS, images, JavaScript, and so on), which allows our browsers to download them somewhat independently and often in parallel. One of the easiest ways to improve Web page performance and thus increase your scalability (fewer objects to serve per page means your servers can serve more pages) is to reduce the number of objects on a page. The biggest offenders on most pages are graphical objects such as pictures and images. As an example let's take a look at Google's search page (www.google.com), which by their own admission is minimalist in nature.[7] At the time of writing Google had five objects on the search page: the HTML, two images, and two JavaScript files. In my very unscientific experiment the search page loaded in about 300 milliseconds. Compare this to a client that we were working with in the online magazine industry, whose home page had more than 200 objects, 145 of which were images and took on average more than 11 seconds to load. What this client didn't realize was that slow page performance was causing them to lose valuable readers. Google published a white paper in 2009 claiming that tests showed an increase in search latency of 400 milliseconds reduced their daily searches by almost 0.6%.[8]

Reducing the number of objects on the page is a great way to improve performance and scalability, but before you rush off to remove all your images there are a few other things to consider. First is obviously the important information that you are trying to convey to your customers. With no images your page will look like the 1992 W3 Project page, which claimed to be the first Web page.[9] Since you need images and JavaScript and CSS files, your second consideration might be to combine all similar objects into a single file. This is not a bad idea, and in fact there are techniques such as CSS image sprites for this exact purpose. An image sprite is a combination of small images into one larger image that can be manipulated with CSS to display any single individual image. The benefit of this is that the number of images requested is significantly reduced. Back to our discussion on the Google search page, one of the two images on the search page is a sprite that consists of about two dozen smaller images that can be individually displayed or not.[10]

So far we've covered that reducing the number of objects on a page will improve performance and scalability, but this must be balanced with the need for modern looking pages thus requiring images, CSS, and JavaScript. Next we covered how these can be combined into a single object to reduce the number of distinct requests that must be made by the browser to render the page. Yet another balance to be made is that combining everything into a single object doesn't make use of the maximum number of simultaneous persistent connections per server that we discussed previously in Rule 3. As a recap this is the browser's capability to download multiple objects simultaneously from a single domain. If everything is in one object, having the capability to download two or more simultaneous objects doesn't help. Now we need to think about breaking these objects back up into a number of smaller ones that can be downloaded simultaneously. One final variable to add to the equation is that part above about simultaneous persistent connections "per server, which will bring us full circle to our DNS discussion noted in Rule 4.

The simultaneous connection feature of a browser is a limit ascribed to each domain that is serving the objects. If all objects on your page come from a single domain (www.akfpartners.com), then whatever the browser has set as the maximum

number of connections is the most objects that can be downloaded simultaneously. As mentioned previously, this maximum is suggested to be set at 2, but many browsers by default have increased this to 6 or more. Therefore, you want your content (images, CSS, JavaScript, and so on) divided into enough objects to take advantage of this feature in most browsers. One technique to really take advantage of this browser feature is to serve different objects from different subdomains (for example, static1.akfpartners.com, static2.akfpartners.com, and so on). The browser considers each of these different domains and allows for each to have the maximum connects concurrently. The client that we talked about earlier who was in the online magazine industry and had an 11-second page load time used this technique across seven subdomains and was able to reduce the average load time to less than 5 seconds.

Unfortunately there is not an absolute answer about ideal size of objects or how many subdomains you should consider. The key to improving performance and scalability is testing your pages. There is a balance between necessary content and functionality, object size, rendering time, total download time, domains, and so on. If you have 100 images on a page, each 50KB, combining them into a single sprite is probably not a great idea because the page will not be able to display any images until the entire 4.9MB object downloads. The same concept goes for JavaScript. If you combine all your .js files into one, your page cannot use any of the JavaScript functions until the entire file is downloaded. The way to know for sure which is the best alternative is to test your pages on a variety of browsers with a variety of ISP connection speeds.

In summary, the fewer the number of objects on a page the better for performance, but this must be balanced with many other factors. Included in these factors are the amount of content that must be displayed, how many objects can be combined, how to maximize the use of simultaneous connections by adding domains, the total page weight and whether penalization can help, and so on. While this rule touches on many Web site performance improvement techniques the real focus is how to improve performance and thus increase the scalability of your site through the reduction of objects on the page. Many other

techniques for optimizing performance should be considered, including loading CSS at the top of the page and JavaScript files at the bottom, minifying files, and making use of caches, lazy loading, and so on.

Rule 6—Use Homogenous Networks

Rule 6: What, When, How, and Why

What: Don't mix the vendor networking gear.

When to use: When designing or expanding your network.

How to use:

- Do not mix different vendors' networking gear (switches and routers).
- Buy best of breed for other networking gear (firewalls, load balancers, and so on).

Why: Intermittent interoperability and availability issues simply aren't worth the potential cost savings.

Key takeaways: Heterogeneous networking gear tends to cause availability and scalability problems. Choose a single provider.

We are technology agnostic, meaning that we believe almost any technology can be made to scale when architected and deployed correctly. This agnosticism ranges from programming language preference to database vendors to hardware. The one caveat to this is with network gear such as routers and switches. Almost all the vendors claim that they implement standard protocols (for example, Internet Control Message Protocol RFC792,[11] Routing Information Protocol RFC1058,[12] Border Gateway Protocol RFC4271[13]) that allow for devices from different vendors to communicate, but many also implement proprietary protocols such as Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP). What we've found in our own practice, as well as with many of our customers, is that each vendor's interpretation of how to implement a standard is often different. As an analogy, if you've ever developed the user interface for a Web page and tested it in a couple different browsers such as Internet

Explorer, Firefox, and Chrome, you've seen firsthand how different implementations of standards can be. Now, imagine that going on inside your network. Mixing Vendor A's network devices with Vendor B's network devices is asking for trouble.

This is not to say that we prefer one vendor over another—we don't. As long as they are a "reference-able" standard utilized by customers larger than you, in terms of network traffic volume, we don't have a preference. This rule does not apply to networking gear such as hubs, load balancers, and firewalls. The network devices that we care about in terms of homogeneity are the ones that must communicate to route communication. For all the other network devices that may or may not be included in your network such as intrusion detection systems (IDS), firewalls, load balancers, and distributed denial of service (DDOS) protection appliances, we recommend best of breed choices. For these devices choose the vendor that best serves your needs in terms of features, reliability, cost, and service.

Summary

This chapter was about making things simpler. Guarding against complexity (aka overengineering—Rule 1) and simplifying every step of your product from your initial requirements or stories through the final implementation (Rule 3) gives us products that are easy to understand from an engineering perspective and therefore easy to scale. By thinking about scale early (Rule 2) even if we don't implement it, we can have solutions ready on demand for our business. Rules 4 and 5 teach us to reduce the work we force browsers to do by reducing the number of objects and DNS lookups we must make to download those objects. Rule 6 teaches us to keep our networks simple and homogenous to decrease the chances of scale and availability problems associated with mixed networking gear.

Endnotes

1. Wikipedia, “Overengineering,” http://en.wikipedia.org/wiki/Overengineering.
2. Jason Fried and David Heinemeier Hansson, *Rework* (New York: Crown Business, 2010).
3. 37Signals, “You Can Always Do Less,” Signal vs. Noise blog, January 14, 2010, http://37signals.com/svn/posts/2106-you-can-always-do-less.
4. Wikipedia, “Minimum Viable Product,” http://en.wikipedia.org/wiki/Minimum_viable_product.
5. To get or install Firebug, go to http://getfirebug.com/.
6. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, Network Working Group Request for Comments 2616, “Hypertext Transfer Protocol—HTTP/1.1,” June 1999, www.ietf.org/rfc/rfc2616.txt.
7. The Official Google Blog, “A Spring Metamorphosis—Google’s New Look,” May 5, 2010, http://googleblog.blogspot.com/2010/05/spring-metamorphosis-googles-new-look.html.
8. Jake Brutlag, “Speed Matters for Google Web Search,” Google, Inc., June 2009, http://code.google.com/speed/files/delayexp.pdf.
9. World Wide Web, www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html.
10. Google.com, www.google.com/images/srpr/nav_logo14.png.
11. J. Postel, Network Working Group Request for Comments 792, “Internet Control Message Protocol,” September 1981, http://tools.ietf.org/html/rfc792.
12. C. Hedrick, Network Working Group Request for Comments 1058, “Routing Information Protocol,” June 1988, http://tools.ietf.org/html/rfc1058.
13. Y. Rekhter, T. Li, and S. Hares, eds., Network Working Group Request for Comments 4271, “A Border Gateway Protocol 4 (BGP-4), January 2006, http://tools.ietf.org/html/rfc4271.

This page intentionally left blank

Index

Symbols/Numbers

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U-V

W

X-Y-Z