

"As a manager who worked under Michael Fisher and Marty Abbott during my time at PayPal/eBay, the opportunity to directly absorb the lessons and experiences presented in this book is invaluable to me now working at Facebook."



—Yishan Wong, former CEO, Reddit, and former director of engineering, Facebook



THE ART OF SCALABILITY

Scalable Web Architecture, Processes, and Organizations
for the Modern Enterprise

S E C O N D E D I T I O N

MARTIN L. ABBOTT

MICHAEL T. FISHER

FOREWORD BY MARTY CAGAN, FOUNDER, SILICON VALLEY PRODUCT GROUP

Dette eksemplar er fremstillet af Det Kgl. Biblioteks Nota-service til Hudayfa Hassan
Ige Waberi. Eksemplaret er personligt og må ikke deles. Misbrug kan medføre
udelukkelse fra Nota-service og retsforfølgelse. Eksemplaret indeholder data, så det
kan spores tilbage til brugeren.

"Many books address how to correct failing product architectures or problematic processes, both of which are symptoms of an unspoken problem. This book not only covers those symptoms, but addresses their underlying cause—the way in which we manage, lead, organize, and staff our teams."

—Jeremy King, chief technology officer and senior vice president, global eCommerce at Walmart.com

The Comprehensive, Proven Approach to IT Scalability— Updated with New Strategies, Technologies, and Case Studies

In *The Art of Scalability, Second Edition*, leading scalability consultants Martin L. Abbott and Michael T. Fisher cover everything you need to know to smoothly scale products and services for any requirement. This extensively revised edition reflects new technologies, strategies, and lessons, as well as new case studies from the authors' pioneering consulting practice, AKF Partners.

Writing for technical and nontechnical decision-makers, Abbott and Fisher cover everything that impacts scalability, including architecture, process, people, organization, and technology. Their insights and recommendations reflect more than thirty years of experience at companies ranging from eBay to Visa, and Salesforce.com to Apple.

You'll find updated strategies for structuring organizations to maximize agility and scalability, as well as new insights into the cloud (IaaS/PaaS) transition, NoSQL, DevOps, business metrics, and more. Using this guide's tools and advice, you can systematically clear away obstacles to scalability—and achieve unprecedented IT and business performance.

COVERAGE INCLUDES

- Why scalability problems start with organizations and people, *not* technology, and what to do about it
- Actionable lessons from real successes and failures
- Staffing, structuring, and leading the agile, scalable organization
- Scaling processes for hyper-growth environments
- Architecting scalability: proprietary models for clarifying needs and making choices—including 15 key success principles
- Emerging technologies and challenges: data cost, datacenter planning, cloud evolution, and customer-aligned monitoring
- Measuring availability, capacity, load, and performance



MARTIN L. ABBOTT is founding partner at AKF Partners. Formerly the chief operations officer of Quigo, an advertising technology startup sold to AOL, he led product strategy, product management, technology development, and client services. He spent nearly six years at eBay, most recently as senior vice president of technology, chief technology officer, and member of the executive staff. **MICHAEL T. FISHER**, founding partner at AKF Partners, was previously chief technology officer of Quigo and vice president of engineering and architecture for PayPal, Inc. He spent seven years helping General Electric develop technology strategy, has earned Six Sigma Master Black Belt status, and served six years as a U.S. Army Captain and pilot.

informat.com/aw | theartofscalability.com | akfpartners.com

Cover design by Chuti Prasertsith

Cover photograph by © Panom / Shutterstock

Text printed on recycled paper

Addison-Wesley

ISBN-13: 978-0-13-403280-1

ISBN-10: 0-13-403280-2

9 780134 032801

\$44.99 U.S. | \$55.99 CANADA

Praise for *The Art of Scalability, Second Edition*

“A how-to manual for building a world-class engineering organization with step-by-step instructions on everything including leadership, architecture, operations, and processes. A driver’s manual for going from 0 to 60, scaling your business. With this book published, there’s no excuse for mistakes—in other words, RTFM.”

—Lon F. Binder, vice president, technology, Warby Parker

“I’ve worked with AKF for years on tough technical challenges. Many books address how to correct failing product architectures or problematic processes, both of which are symptoms of an unspoken problem. This book not only covers those symptoms, but also addresses their underlying cause—the way in which we manage, lead, organize, and staff our teams.”

—Jeremy King, chief technology officer and senior vice president, global ecommerce, Walmart.com

“I love this book because it teaches an important lesson most technology-focused books don’t: how to build highly scalable and successful technology organizations that build highly scalable technology solutions. There’s plenty of great technology coaching in this book, but there are also excellent examples of how to build scalable culture, principles, processes, and decision trees. This book remains one of my few constant go-to reference guides.”

—Chris Schremser, chief technology officer, ZirMed

Praise for the First Edition

“This book is much more than you may think it is. Scale is not just about designing Web sites that don’t crash when lots of users show up. It is about designing your company so that it doesn’t crash when your business needs to grow. These guys have been there on the front lines of some of the most successful Internet companies of our time, and they share the good, the bad, and the ugly about how to not just survive, but thrive.”

—Marty Cagan, founder, Silicon Valley Product Group

“A must read for anyone building a Web service for the mass market.”

—Dana Stalder, general partner, Matrix Partners

“Abbott and Fisher have deep experiences with scale in both large and small enterprises. What’s unique about their approach to scalability is they start by focusing on the true foundation: people and process, without which true scalability cannot be built. Abbott and Fisher leverage their years of experience in a very accessible and practical approach to scalability that has been proven over time with their significant success.”

—Geoffrey Weber, vice president of internet operations/IT, Shutterfly

“If I wanted the best diagnoses for my health I would go to the Mayo Clinic. If I wanted the best diagnoses for my portfolio companies’ performance and scalability I would call Martin and Michael. They have recommended solutions to performance and scalability issues that have saved some of my companies from a total rewrite of the system.”

—Warren M. Weiss, general partner, Foundation Capital

“As a manager who worked under Michael Fisher and Marty Abbott during my time at PayPal/eBay, the opportunity to directly absorb the lessons and experiences presented in this book are invaluable to me now working at Facebook.”

—Yishan Wong, former CEO, Reddit, and former director of engineering, Facebook

“*The Art of Scalability* is by far the best book on scalability on the market today. The authors tackle the issues of scalability from processes, to people, to performance, to the highly technical. Whether your organization is just starting out and is defining processes as you go, or you are a mature organization, this is the ideal book

to help you deal with scalability issues before, during, or after an incident. Having built several projects, programs, and companies from small to significant scale, I can honestly say I wish I had this book one, five, and ten years ago.”

—Jeremy Wright, chief executive officer, b5media, Inc.

“Only a handful of people in the world have experienced the kind of growth-related challenges that Fisher and Abbott have seen at eBay, PayPal, and the other companies they’ve helped to build. Fewer still have successfully overcome such challenges. *The Art of Scalability* provides a great summary of lessons learned while scaling two of the largest internet companies in the history of the space, and it’s a must-read for any executive at a hyper-growth company. What’s more, it’s well-written and highly entertaining. I couldn’t put it down.”

—Kevin Fortuna, partner, AKF Consulting

“Marty and Mike’s book covers all the bases, from understanding how to build a scalable organization to the processes and technology necessary to run a highly scalable architecture. They have packed in a ton of great practical solutions from real world experiences. This book is a must-read for anyone having difficulty managing the scale of a hyper-growth company or a startup hoping to achieve hyper growth.”

—Tom Keeven, partner, AKF Consulting

“*The Art of Scalability* is remarkable in its wealth of information and clarity; the authors provide novel, practical, and demystifying approaches to identify, predict, and resolve scalability problems before they surface. Marty Abbott and Michael Fisher use their rich experience and vision, providing unique and groundbreaking tools to assist small and hyper-growth organizations as they maneuver in today’s demanding technological environments.”

—Joseph M. Potenza, attorney, Banner & Witcoff, Ltd.

The Art of Scalability

Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise

Second Edition

Martin L. Abbott
Michael T. Fisher

◆ Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Abbott, Martin L.

The art of scalability : scalable web architecture, processes, and organizations for the modern enterprise / Martin L. Abbott, Michael T. Fisher.

pages cm

Includes index.

ISBN 978-0-13-403280-1 (pbk. : alk. paper)

1. Web site development. 2. Computer networks—Scalability. 3. Business enterprises—Computer networks. I. Fisher, Michael T. II. Title.

TK5105.888.A2178 2015

658.4'06—dc23

2015009317

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-403280-1

ISBN-10: 0-13-403280-2

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing, June 2015

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
John Fuller

**Senior Project
Editor**
Mary Kesel Wilson

Copy Editor
Jill Hobbs

Indexer
Jack Lewis

Proofreader
Andrea Fox

Technical Reviewers
Roger Andelin
Chris Schremser
Geoffrey Weber

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

*“To my father, for teaching me how to succeed, and to my wife
Heather, for teaching me how to have fun.”*

—Marty Abbott

*“To my parents, for their guidance, and to my wife and son, for their
unflagging support.”*

—Michael Fisher

Contents

Foreword	xxiii
Acknowledgments	xxvii
About the Authors	xxix
Introduction.....	1
Part I: Staffing a Scalable Organization.....	7
Chapter 1: The Impact of People and Leadership on Scalability.....	9
The Case Method	9
Why People?	10
Why Organizations?	11
Why Management and Leadership?	17
Conclusion	19
Key Points.....	20
Chapter 2: Roles for the Scalable Technology Organization.....	21
The Effects of Failure	21
Defining Roles	23
Executive Responsibilities.....	25
Chief Executive Officer	25
Chief Financial Officer.....	27
Business Unit Owners, General Managers, and P&L Owners	27
Chief Technology Officer/Chief Information Officer	28
Individual Contributor Responsibilities	30
Architecture Responsibilities	30
Engineering Responsibilities.....	31
DevOps Responsibilities.....	31
Infrastructure Responsibilities	32
Quality Assurance Responsibilities.....	33
Capacity Planning Responsibilities.....	33

A Tool for Defining Responsibilities	35
Conclusion	39
Key Points.....	40
Chapter 3: Designing Organizations.....	41
Organizational Influences That Affect Scalability	41
Team Size	44
Warning Signs	47
Growing or Splitting Teams	48
Organizational Structure	51
Functional Organization	51
Matrix Organization	56
Agile Organization.....	59
Conclusion	69
Key Points.....	70
Chapter 4: Leadership 101	71
What Is Leadership?	72
Leadership: A Conceptual Model	74
Taking Stock of Who You Are	76
Leading from the Front.....	78
Checking Your Ego at the Door	79
Mission First, People Always	80
Making Timely, Sound, and Morally Correct Decisions.....	81
Empowering Teams and Scalability	82
Alignment with Shareholder Value	83
Transformational Leadership	84
Vision	84
Mission	87
Goals	89
Putting It All Together	90
The Causal Roadmap to Success.....	94
Conclusion	95
Key Points.....	96
Chapter 5: Management 101.....	99
What Is Management?	100
Project and Task Management	102
Building Teams: A Sports Analogy	105

Upgrading Teams: A Garden Analogy	107
Measurement, Metrics, and Goal Evaluation	111
The Goal Tree	114
Paving the Path for Success	115
Conclusion	116
Key Points	117
Chapter 6: Relationships, Mindset, and the Business Case	119
Understanding the Experiential Chasm	119
Why the Business Executive Might Be the Problem	120
Why the Technology Executive Might Be the Problem	121
Defeating the IT Mindset	122
The Business Case for Scale	124
Conclusion	127
Key Points	128
Part II: Building Processes for Scale	129
Chapter 7: Why Processes Are Critical to Scale	131
The Purpose of Process	132
Right Time, Right Process	135
A Process Maturity Framework	136
When to Implement Processes	137
Process Complexity	137
When Good Processes Go Bad	139
Conclusion	140
Key Points	141
Chapter 8: Managing Incidents and Problems	143
What Is an Incident?	144
What Is a Problem?	145
The Components of Incident Management	146
The Components of Problem Management	149
Resolving Conflicts Between Incident and Problem Management	150
Incident and Problem Life Cycles	150
Implementing the Daily Incident Meeting	152
Implementing the Quarterly Incident Review	153
The Postmortem Process	153

Putting It All Together	156
Conclusion	157
Key Points.....	158
Chapter 9: Managing Crises and Escalations	159
What Is a Crisis?.....	160
Why Differentiate a Crisis from Any Other Incident?.....	161
How Crises Can Change a Company	162
Order Out of Chaos	163
The Role of the Problem Manager	164
The Role of Team Managers.....	166
The Role of Engineering Leads.....	167
The Role of Individual Contributors.....	167
Communications and Control.....	168
The War Room.....	169
Escalations	170
Status Communications	171
Crisis Postmortem and Communication	172
Conclusion	173
Key Points.....	174
Chapter 10: Controlling Change in Production Environments	177
What Is a Change?	178
Change Identification	179
Change Management	180
Change Proposal.....	183
Change Approval	186
Change Scheduling.....	187
Change Implementation and Logging.....	189
Change Validation	189
Change Review.....	191
The Change Control Meeting	191
Continuous Process Improvement.....	192
Conclusion	193
Key Points.....	194
Chapter 11: Determining Headroom for Applications	197
Purpose of the Process	198
Structure of the Process	199
Ideal Usage Percentage	203

A Quick Example Using Spreadsheets.....	206
Conclusion	207
Key Points	208
Chapter 12: Establishing Architectural Principles	209
Principles and Goals	209
Principle Selection.....	212
AKF's Most Commonly Adopted Architectural Principles	214
N + 1 Design.....	214
Design for Rollback	215
Design to Be Disabled.....	215
Design to Be Monitored	215
Design for Multiple Live Sites.....	216
Use Mature Technologies	217
Asynchronous Design.....	217
Stateless Systems.....	218
Scale Out, Not Up	219
Design for at Least Two Axes of Scale	219
Buy When Non-Core	220
Use Commodity Hardware.....	220
Build Small, Release Small, Fail Fast	221
Isolate Faults.....	221
Automation over People	221
Conclusion	222
Key Points.....	223
Chapter 13: Joint Architecture Design and Architecture Review Board.....	225
Fixing Organizational Dysfunction	225
Designing for Scale Cross-Functionally	226
JAD Entry and Exit Criteria	228
From JAD to ARB	230
Conducting the Meeting.....	232
ARB Entry and Exit Criteria.....	234
Conclusion	236
Key Points.....	237
Chapter 14: Agile Architecture Design	239
Architecture in Agile Organizations	240
Ownership of Architecture	241
Limited Resources.....	242

Standards	243
ARB in the Agile Organization.....	246
Conclusion	247
Key Points.....	247
Chapter 15: Focus on Core Competencies: Build Versus Buy	249
Building Versus Buying, and Scalability	249
Focusing on Cost	250
Focusing on Strategy.....	251
“Not Built Here” Phenomenon.....	252
Merging Cost and Strategy	252
Does This Component Create Strategic Competitive Differentiation?..	253
Are We the Best Owners of This Component or Asset?	253
What Is the Competition for This Component?	254
Can We Build This Component Cost-Effectively?.....	254
The Best Buy Decision Ever	255
Anatomy of a Build-It-Yourself Failure	256
Conclusion	258
Key Points.....	258
Chapter 16: Determining Risk	259
Importance of Risk Management to Scale	259
Measuring Risk.....	261
Managing Risk	268
Conclusion	271
Key Points.....	272
Chapter 17: Performance and Stress Testing.....	273
Performing Performance Testing.....	273
Establish Success Criteria	274
Establish the Appropriate Environment	275
Define the Tests	276
Execute the Tests	277
Analyze the Data	278
Report to Engineers	279
Repeat the Tests and Analysis.....	279
Don’t Stress over Stress Testing.....	281
Identify the Objectives	281
Identify the Key Services.....	282
Determine the Load	283

Establish the Appropriate Environment	283
Identify the Monitors	284
Create the Load	284
Execute the Tests	284
Analyze the Data	285
Performance and Stress Testing for Scalability	287
Conclusion	288
Key Points.....	289
Chapter 18: Barrier Conditions and Rollback	291
Barrier Conditions	291
Barrier Conditions and Agile Development	293
Barrier Conditions and Waterfall Development	295
Barrier Conditions and Hybrid Models	296
Rollback Capabilities	297
Rollback Window	297
Rollback Technology Considerations	298
Cost Considerations of Rollback	299
Markdown Functionality: Design to Be Disabled	300
Conclusion	301
Key Points.....	302
Chapter 19: Fast or Right?	303
Tradeoffs in Business	303
Relation to Scalability.....	306
How to Think About the Decision	307
Conclusion	311
Key Points.....	313
Part III: Architecting Scalable Solutions	315
Chapter 20: Designing for Any Technology	317
An Implementation Is Not an Architecture.....	317
Technology-Agnostic Design.....	318
TAD and Cost	319
TAD and Risk	320
TAD and Scalability.....	321
TAD and Availability	323
The TAD Approach	323

Conclusion	325
Key Points.....	325
Chapter 21: Creating Fault-Isolative Architectural Structures	327
Fault-Isolative Architecture Terms	327
Benefits of Fault Isolation	329
Fault Isolation and Availability: Limiting Impact	329
Fault Isolation and Availability: Incident Detection and Resolution ..	334
Fault Isolation and Scalability	334
Fault Isolation and Time to Market	334
Fault Isolation and Cost	335
How to Approach Fault Isolation	336
Principle 1: Nothing Is Shared	337
Principle 2: Nothing Crosses a Swim Lane Boundary.....	338
Principle 3: Transactions Occur Along Swim Lanes	338
When to Implement Fault Isolation.....	339
Approach 1: Swim Lane the Money-Maker	339
Approach 2: Swim Lane the Biggest Sources of Incidents.....	339
Approach 3: Swim Lane Along Natural Barriers	340
How to Test Fault-Isolative Designs	341
Conclusion	341
Key Points.....	342
Chapter 22: Introduction to the AKF Scale Cube.....	343
The AKF Scale Cube.....	343
The <i>x</i> -Axis of the Cube.....	344
The <i>y</i> -Axis of the Cube.....	346
The <i>z</i> -Axis of the Cube.....	349
Putting It All Together	350
When and Where to Use the Cube	352
Conclusion	353
Key Points.....	354
Chapter 23: Splitting Applications for Scale.....	357
The AKF Scale Cube for Applications	357
The <i>x</i> -Axis of the AKF Application Scale Cube	359
The <i>y</i> -Axis of the AKF Application Scale Cube	361
The <i>z</i> -Axis of the AKF Application Scale Cube	363
Putting It All Together	365

Practical Use of the Application Cube.....	367
Observations.....	370
Conclusion	371
Key Points.....	372
Chapter 24: Splitting Databases for Scale.....	375
Applying the AKF Scale Cube to Databases	375
The <i>x</i> -Axis of the AKF Database Scale Cube	376
The <i>y</i> -Axis of the AKF Database Scale Cube	381
The <i>z</i> -Axis of the AKF Database Scale Cube	383
Putting It All Together	385
Practical Use of the Database Cube.....	388
Ecommerce Implementation	388
Search Implementation.....	389
Business-to-Business SaaS Solution.....	391
Observations.....	392
Timeline Considerations.....	393
Conclusion	393
Key Points.....	394
Chapter 25: Caching for Performance and Scale.....	395
Caching Defined	395
Object Caches.....	399
Application Caches.....	402
Proxy Caches	402
Reverse Proxy Caches.....	403
Caching Software	406
Content Delivery Networks	407
Conclusion	408
Key Points.....	409
Chapter 26: Asynchronous Design for Scale.....	411
Syncing Up on Synchronization	411
Synchronous Versus Asynchronous Calls	412
Scaling Synchronously or Asynchronously	414
Example Asynchronous Systems.....	416
Defining State.....	418
Conclusion	422
Key Points	423

Part IV: Solving Other Issues and Challenges	425
Chapter 27: Too Much Data	427
The Cost of Data	427
The Value of Data and the Cost-Value Dilemma.	430
Making Data Profitable	431
Option Value	431
Strategic Competitive Differentiation	432
Cost-Justify the Solution (Tiered Storage Solutions)	432
Transform the Data	433
Handling Large Amounts of Data	434
Big Data	438
A NoSQL Primer	440
Conclusion	444
Key Points.	444
Chapter 28: Grid Computing	447
History of Grid Computing	447
Pros and Cons of Grids.	449
Pros of Grids.	449
Cons of Grids	452
Different Uses for Grid Computing.	454
Production Grid	454
Build Grid.	455
Data Warehouse Grid.	456
Back-Office Grid	456
Conclusion	457
Key Points.	458
Chapter 29: Soaring in the Clouds	459
History and Definitions	460
Public Versus Private Clouds	462
Characteristics and Architecture of Clouds	463
Pay by Usage.	463
Scale on Demand	464
Multiple Tenants.	465
Virtualization.	466
Differences Between Clouds and Grids.	467

Pros and Cons of Cloud Computing	468
Pros of Cloud Computing	468
Cons of Cloud Computing	471
Where Clouds Fit in Different Companies	476
Environments	476
Skill Sets	478
Decision Process	478
Conclusion	481
Key Points	482
Chapter 30: Making Applications Cloud Ready	485
The Scale Cube in a Cloud	485
x-Axis	485
y- and z-Axes	486
Overcoming Challenges	487
Fault Isolation in a Cloud	487
Variability in Input/Output	489
Intuit Case Study	491
Conclusion	493
Key Points	494
Chapter 31: Monitoring Applications	495
“Why Didn’t We Catch That Earlier?”	495
A Framework for Monitoring	496
User Experience and Business Metrics	499
Systems Monitoring	501
Application Monitoring	502
Measuring Monitoring: What Is and Isn’t Valuable?	503
Monitoring and Processes	504
Conclusion	506
Key Points	507
Chapter 32: Planning Data Centers	509
Data Center Costs and Constraints	509
Location, Location, Location	511
Data Centers and Incremental Growth	514
When Do I Consider IaaS?	516
Three Magic Rules of Three	519
The First Rule of Three: Three Magic Drivers of Data Center Costs	520

The Second Rule of Three: Three Is the Magic Number for Servers	520
The Third Rule of Three: Three Is the Magic Number for Data Centers	521
Multiple Active Data Center Considerations.....	525
Conclusion	527
Key Points.....	528
Chapter 33: Putting It All Together	531
What to Do Now?.....	532
Further Resources on Scalability,.....	535
Blogs.....	535
Books	535
Part V: Appendices	537
Appendix A: Calculating Availability.....	539
Hardware Uptime.....	540
Customer Complaints.....	541
Portion of Site Down.....	542
Third-Party Monitoring Service	543
Business Graph	544
Appendix B: Capacity Planning Calculations.....	547
Appendix C: Load and Performance Calculations	555
Index	563

Foreword

Perhaps your company began as a brick-and-mortar retailer, or an airline, or a financial services company.

A retailer creates (or buys) technology to coordinate and manage inventory, distribution, billing, and point of sale systems. An airline creates technology to manage the logistics involved in flights, crews, reservations, payment, and fleet maintenance. A financial services company creates technology to manage its customers' assets and investments.

But over the past several years, almost all of these companies, as well as their counterparts in nearly every other industry, have realized that to remain competitive, they need to take their use of technology to an entirely different level—they now need to engage directly with their customers.

Every industry is being reshaped by technology. If they hope to maintain their place as competitive, viable enterprises, companies have no choice but to embrace technology, often in ways that go well beyond their comfort zone.

For example, most retailers now find they need to sell their goods directly to consumers online. Most airlines are trying very hard to entice their customers to purchase their air travel online directly through the airline's site. And nearly all financial services companies work to enable their customers to manage assets and trade directly via their real-time financial sites.

Unfortunately, many of these companies are trying to manage this new customer-facing and customer-enabling technology in the same way they manage their internal technology. The result is that many of these companies have very broken technology and provide terrible customer experiences. Even worse, they don't have the organization, people, or processes in place to improve them.

What companies worldwide are discovering is that there is a very profound difference between utilizing technology to help *run* your company, and leveraging technology to provide your actual products and services directly *for* your customers. It also explains why "technology transformation" initiatives are popping up at so many companies.

This book is all about this necessary transformation. Such a transformation represents a shift in organization, people, process, and especially culture, and scalability is at the center of this transformation.

- Scaling from hundreds of your employees using your technology, to millions of your customers depending on your technology

- Scaling from a small IT cost-center team serving their colleagues in finance and marketing, to a substantial profit-center technology team serving your customers
- More generally, scaling your people, processes, and technology to meet the demands of a modern technology-powered business

But why is technology for your customers so different and so much more difficult to manage than technology for your employees? Several reasons:

- You pay your employees to work at your company and use the technology you tell them they need to use. In contrast, every customer makes his or her own purchase decision—and if she doesn't want it, she won't use it. Your customers must *choose* to use your technology.
- With your own employees, you can get away with requiring training courses, reading manuals, or holding their hands if necessary. In contrast, if your customers can't figure out how to use your technology, they are just a click away from your competitor.
- For internal technology, we measure scale and simultaneous usage in the hundreds of users. For our customers, that scope increases to hundreds of thousands or very often millions of users.
- With internal technology, if a problem arises with the technology, the users are your employees and they are forced to deal with it. For your customers, an issue such as an outage immediately disrupts revenue streams, usually gets the attention of the CEO, and sometimes even draws the notice of the press.
- The harsh truth is that most customer technology simply has a dramatically higher bar set in terms of the definition, design, implementation, testing, deployment, and support than is necessary with most internal technology.

For most companies, establishing a true customer technology competency is *the single most important thing for them to be doing to ensure their survival*, yet remarkably some of them don't even realize they have a problem. They assume that "technology is technology" and the same people who managed their enterprise resource planning implementation shouldn't have too much trouble getting something going online.

If your company is in need of this transformation, then this book is essential reading. It provides a proven blueprint for the necessary change.

Marty and Michael have been there and done that with most of the technology industry's leading companies. I have known and worked with both of these guys for many years. They are not management consultants who could barely launch a

brochure site. They are hands-on leaders who have spent decades in the trenches with their teams creating technology-powered businesses serving hundreds of millions of users and customers. They are the best in the world at what they do, and this new edition is a goldmine of information for any technology organization working to raise its game.

—*Marty Cagan*
Founder, Silicon Valley Product Group

Det Kgl. Bibliotek
Den Kongelige Bibliotek

Det Kgl. Bibliotek er et statslig etablissement under embedsbestyrket ledelse af en direktør, der er ansat ved et statslig etablissement under embedsbestyrket ledelse af en minister. Det Kgl. Bibliotek har til opgave at samle og opbevare det nationale kulturarvs dokumentation og at udlevere den til forskning og undervisning.

Det Kgl. Bibliotek

Det Kgl. Bibliotek har til opgave at samle og opbevare det nationale kulturarvs dokumentation og at udlevere den til forskning og undervisning.

Acknowledgments

The authors would like to recognize, first and foremost, the experience and advice of our partner and cofounder Tom Keeven. The process and technology portions of this book were built over time with the help of Tom and his many years of experience. Tom started the business that became AKF Partners. We often joke that Tom has forgotten more about architecting highly available and scalable sites than most of us will ever learn.

We would also like to thank several AKF team members—Geoff Kershner, Dave Berardi, Mike Paylor, Kirk Sanford, Steve Mason, and Alex Hooper—who contributed their combined decades of experience and knowledge not only to this second edition, but also to AKF Partners’ consulting practice. Without their help putting the concepts from the first edition into practice and helping to mature them over time, this second edition would not be possible.

Additionally, the authors owe a great debt of gratitude to this edition’s technical reviewers—Geoffrey Weber, Chris Schremser, and Roger Andelin. All three of these individuals are experienced technology executives who have decades of hands-on experience designing, developing, implementing, and supporting large-scale systems in industries ranging from ecommerce to health care. They willingly agreed to accept our poorly written drafts and help turn them into easily consumable prose for the benefit of our readers.

This edition would not be possible without the support provided by the team at Addison-Wesley, including executive editor Laura Lewin, development editor Songlin Qiu, and editorial assistant Olivia Basegio. Laura quickly became the champion for a second edition after discussing the significant changes with regard to scaling systems and organizations that have occurred over the five years since the first edition was published. Songlin has been an invaluable partner in ensuring both the first and second editions of *The Art of Scalability* were consistent, clear, and correct. Olivia has saved us multiple times when technical challenges threatened to delay or derail us.

We further would like to recognize our colleagues and teams at Quigo, eBay, and PayPal. These are the companies at which we really started to build and test many of the approaches mentioned in the technology and process sections of this book. The list of names within these teams is quite large, but the individuals know who they are.

Finally, we’d like to acknowledge the U.S. Army and United States Military Academy. Together they created a leadership lab unlike any other we can imagine.

Multiple reviewers have reviewed this book as we have attempted to provide the best possible work for the reader. However, in a work this large, errors will inevitably occur. All errors in the text are completely the authors’ fault.

About the Authors

Martin L. Abbott is a founding partner at the growth and scalability advisory firm AKF Partners. He was formerly chief operations officer at Quigo, an advertising technology startup sold to AOL, where he was responsible for product strategy, product management, technology development, and client services. Marty spent nearly six years at eBay, most recently as senior vice president of technology, chief technology officer, and member of the executive staff. Prior to his time at eBay, Marty held domestic and international engineering, management, and executive positions at Gateway and Motorola. He has served on the boards of directors of several private and public companies. Marty has a B.S. in computer science from the United States Military Academy, has an M.S. in computer engineering from the University of Florida, is a graduate of the Harvard Business School Executive Education Program, and has a Doctor of Management from Case Western Reserve University.

Michael T. Fisher is a founding partner at the growth and scalability advisory firm AKF Partners. Prior to cofounding AKF Partners, Michael was the chief technology officer at Quigo, a startup Internet advertising company that was acquired by AOL in 2007. Before his time at Quigo, Michael served as vice president, engineering and architecture, for PayPal, Inc., an eBay company. Prior to joining PayPal, he spent seven years at General Electric helping to develop the company's technology strategy and was a Six Sigma Master Black Belt. Michael served six years as a Captain and pilot in the U.S. Army. He received a Ph.D. and an MBA from Case Western Reserve University's Weatherhead School of Management, an M.S. in information systems from Hawaii-Pacific University, and a B.S. 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.

Introduction

Thanks for picking up the second edition of *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. This book has been recognized by academics and professionals as one of the best resources available to learn the art of scaling systems and organizations. This second edition includes new content, revisions, and updates. As consultants and advisors to hundreds of hyper-growth companies, we have been fortunate enough to be on the forefront of many industry changes, including new technologies and new approaches to implementing products. While we hope our clients see value in our knowledge and experience, we are not ignorant of the fact that a large part of the value we bring to bear on a subject comes from our interactions with so many other technology companies. In this edition, we share even more of these lessons learned from our consulting practice.

In this second edition, we have added several key topics that we believe are critical to address in a book on scalability. One of the most important new topics focuses on a new organizational structure that we refer to as the Agile Organization. Other notable topics include the changing rationale for moving from data centers to clouds (IaaS/PaaS), why NoSQL solutions aren't in and of themselves a panacea for scaling, and the importance of business metrics to the health of the overall system.

In the first edition of *The Art of Scalability*, we used a fictional company called AllScale to demonstrate many of the concepts. This fictional company was an aggregation of many of our clients and the challenges they faced in the real world. While AllScale provided value in highlighting the key points in the first edition, we believe that real stories make more of an impact with readers. As such, we've replaced AllScale with real-world stories of successes and failures in the current edition.

The information contained in this book has been carefully designed to be appropriate for any employee, manager, or executive of an organization or company that provides technology solutions. For the nontechnical executive or product manager, this book can help you prevent scalability disasters by arming you with the tools needed to ask the right questions and focus on the right areas. For technologists and engineers, this book provides models and approaches that, once employed, will help you scale your products, processes, and organizations.

Our experience with scalability goes beyond academic study and research. Although we are both formally trained as engineers, we don't believe academic

programs teach scalability very well. Rather, we have learned about scalability by suffering through the challenges of scaling systems for a combined 30-plus years. We have been engineers, managers, executives, and advisors for startups as well as *Fortune 500* companies. The list of companies that our firm or we as individuals have worked with includes such familiar names as General Electric, Motorola, Gateway, eBay, Intuit, Salesforce, Apple, Dell, Walmart, Visa, ServiceNow, DreamWorks Animation, LinkedIn, Carbonite, Shutterfly, and PayPal. The list also includes hundreds of less famous startups that need to be able to scale as they grow. Having learned the scalability lessons through thousands of hours spent diagnosing problems and thousands more hours spent designing preventions for those problems, we want to share our combined knowledge. This motivation was the driving force behind our decisions to start our consulting practice, AKF Partners, in 2007, and to write the first edition of this book, and it remains our preeminent goal in this second edition.

Scalability: So Much More Than Just Technology

Pilots are taught, and statistics show, that many aircraft incidents are the result of multiple failures that snowball into total system failure and catastrophe. In aviation, these multiple failures, which are called an error chain, often start with human rather than mechanical failure. In fact, Boeing identified that 55% of all aircraft incidents involving Boeing aircraft between 1995 and 2005 had human factors-related causes.¹

Our experience with scalability-related issues follows a similar trend. The chief technology officer (CTO) or executive responsible for scale of a technology platform may see scalability as purely a technical endeavor. This perception is the first, and very human, failure in the error chain. Because the CTO is overly technology focused, she fails to define the processes necessary to identify scalability bottlenecks—failure number two. Because no one is identifying bottlenecks or chokepoints in the architecture, the user count or transaction volume exceeds a certain threshold and the entire product fails—failure number three. The team assembles to solve the problem, but because it has never invested in processes to troubleshoot incidents and their related problems, the team misdiagnoses the failure as “the database needs to be tuned”—failure number four. The vicious cycle goes on for days, with people focusing on different pieces of the technology stack and blaming everything from

1. Boeing. (May 2006). “Statistical Summary of Commercial Jet Airplane Accidents Worldwide Operations.”

SCALABILITY: SO MUCH MORE THAN JUST TECHNOLOGY

3

firewalls, to applications, to the persistence tiers to which the apps speak. Team interactions devolve into shouting matches and finger-pointing sessions, while services remain slow and unresponsive. Customers walk away, team morale flat-lines, and shareholders are left holding the bag.

The key point here is that crises resulting from an inability to scale to end-user demands are almost never technology problems alone. In our experience as former executives and advisors to our clients, scalability issues start with organizations and people, and only then spread to process and technology. People, being human, make ill-informed or poor choices regarding technical implementations, which in turn sometimes manifest themselves as a failure of a technology platform to scale. People also ignore the development of processes that might help them learn from past mistakes and sometimes put overly burdensome processes in place, which in turn might force the organization to make poor decisions or make decisions too late to be effective. A lack of attention to the people and processes that create and support technical decision making can lead to a vicious cycle of bad technical decisions, as depicted in the left side of Figure I.1. This book is the first of its kind focused on creating a virtuous cycle of people and process scalability to support better, faster, and more scalable technology decisions, as depicted in the right side of Figure I.1.

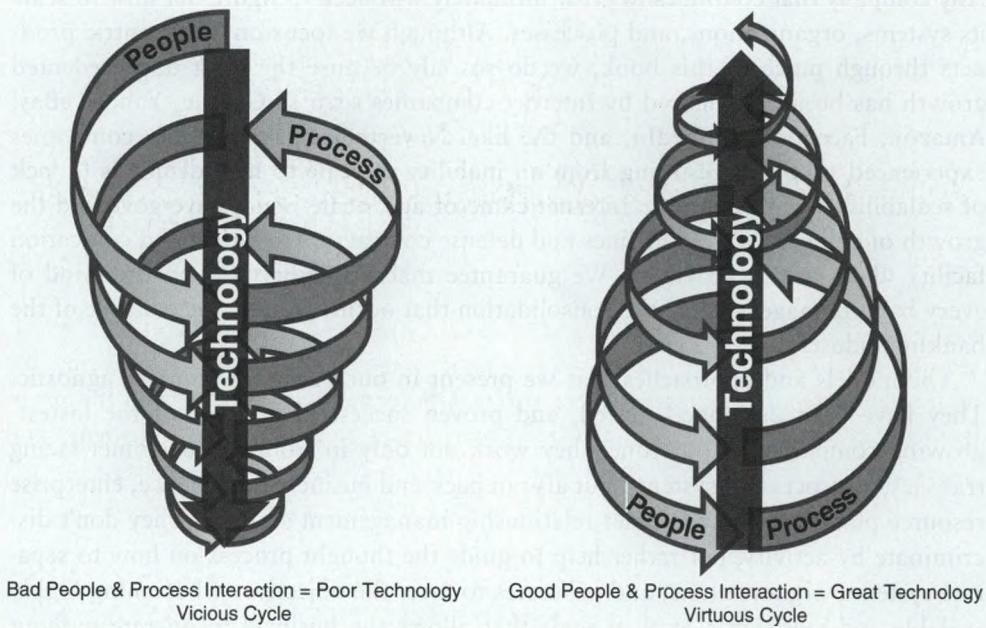


Figure I.1 *Vicious and Virtuous Technology Cycles Utility*

Art Versus Science

Our choice of the word *art* in the title of this book is a deliberate one. *Art* conjures up images of a fluid nature, whereas *science* seems much more structured and static. It is this image that we heavily rely on, as our experience has taught us that there is no single approach or way to guarantee an appropriate level of scale within a platform, organization, or process. A successful approach to scaling must be crafted around the ecosystem created by the intersection of the current technology platform, the characteristics of the organization, and the maturity and appropriateness of the existing processes. This book focuses on providing skills and teaching approaches that, if employed properly, will help solve nearly any scalability or availability problem.

This is not to say that we don't advocate the application of the scientific method in nearly any approach, because we absolutely do. *Art* here is a nod to the notion that you simply cannot take a "one size fits all" approach to any potential system and expect to meet with success.

Who Needs Scalability?

Any company that continues to grow ultimately will need to figure out how to scale its systems, organizations, and processes. Although we focus on Web-centric products through much of this book, we do so only because the most unprecedented growth has been experienced by Internet companies such as Google, Yahoo, eBay, Amazon, Facebook, LinkedIn, and the like. Nevertheless, many other companies experienced problems resulting from an inability to scale to new demands (a lack of scalability) long before the Internet came of age. Scale issues have governed the growth of companies from airlines and defense contractors to banks and colocation facility (data center) providers. We guarantee that scalability was on the mind of every bank manager during the consolidation that occurred after the collapse of the banking industry.

The models and approaches that we present in our book are industry agnostic. They have been developed, tested, and proven successful in some of the fastest-growing companies of our time; they work not only in front-end customer-facing transaction-processing systems, but also in back-end business intelligence, enterprise resource planning, and customer relationship management systems. They don't discriminate by activity, but rather help to guide the thought process on how to separate systems, organizations, and processes to meet the objective of becoming highly scalable and reaching a level of scale that allows the business to operate without concerns about its ability to meet customer or end-user demands.

Book Organization and Structure

We've divided the book into five parts. Part I, "Staffing a Scalable Organization," focuses on organization, management, and leadership. Far too often, managers and leaders are promoted based on their talents within their area of expertise. Engineering leaders and managers, for example, are very often promoted based on their technical acumen and aren't given the time or resources needed to develop their business, management, and leadership acumen. Although they might perform well in the architectural and technical aspects of scale, their expertise in organizational scale needs is often shallow or nonexistent. Our intent is to provide these managers and leaders with a foundation from which they can grow and prosper as managers and leaders.

Part II, "Building Processes for Scale," focuses on the processes that help hyper-growth companies scale their technical platforms. We cover topics ranging from technical issue resolution to crisis management. We also discuss processes meant for governing architectural decisions and principles to help companies scale their platforms.

Part III, "Architecting Scalable Solutions," focuses on the technical and architectural aspects of scale. We introduce proprietary models developed within AKF Partners, our consulting and advisory practice. These models are intended to help organizations think through their scalability needs and alternatives.

Part IV, "Solving Other Issues and Challenges," discusses emerging technologies such as grid computing and cloud computing. We also address some unique problems within hyper-growth companies such as the immense growth and cost of data as well as issues to consider when planning data centers and evolving monitoring strategies to be closer to customers.

Part V, "Appendices," explains how to calculate some of the most common scalability numbers. Its coverage includes the calculation of availability, capacity planning, and load and performance.

The lessons in this book have not been designed in the laboratory, nor are they based on unapplied theory. Rather, these lessons have been designed and implemented by engineers, technology leaders, and organizations through years of struggling to keep their dreams, businesses, and systems afloat. The authors have had the great fortune to be a small part of many of these teams in many different roles—sometimes as active participants, at other times as observers. We have seen how putting these lessons into practice has yielded success—and how the unwillingness or inability to do so has led to failure. This book aims to teach you these lessons and put you and your team on the road to success. We believe the lessons here are valuable for everyone from engineering staffs to product staffs, including every level from the individual contributor to the CEO.

Part I

Staffing a Scalable Organization

Chapter 1: The Impact of People and Leadership on Scalability	9
Chapter 2: Roles for the Scalable Technology Organization	21
Chapter 3: Designing Organizations	41
Chapter 4: Leadership 101	71
Chapter 5: Management 101	99
Chapter 6: Relationships, Mindset, and the Business Case	119

Chapter 1

The Impact of People and Leadership on Scalability

Fighting with a large army under your command is nowise different from fighting with a small one;
it is merely a question of instituting signs and signals.

—Sun Tzu

Here's a multiple-choice pop quiz. Which of the following is the most important factor in ensuring the long-term scalability of the products you produce: people, process, or technology? Was your answer "people"? If so, we commend you. If it wasn't, then consider the following: To our knowledge, the "Robot Apocalypse" has not yet happened, so our products do not build themselves. Any value or liability within our products is a result of human endeavors. People design, code, and configure the systems that run our products and are solely responsible for any defects in design, code, or configuration. People alone are responsible for the product architectures that either successfully process incredible transaction volumes or fail miserably, crumpling beneath the weight of consumer demand. All too often, however, people are the things that we as technologists and engineers overlook when striving to create scalable solutions or solve problems related to scalability. People are woefully overlooked and underappreciated. Organizational reviews of employees are once-a-year, check-the-box exercises in which cursory assessments are written in haste. Managers and leaders are frequently untrained or undertrained in the performance of their duties. In this chapter, we explain why the people, structure, management, and leadership in your organization all have an enormous impact on your ability to scale.

The Case Method

Throughout this second edition of *The Art of Scalability*, we present stories or cases from real-world companies that help to breathe life into the topics we discuss. Some

of the examples we use were developed through first-hand experience working with clients in our firm, AKF Partners. Other cases were developed based on extensive research into public documents and/or interviews with people who have first-hand information from interesting companies.

The first edition of *The Art of Scalability* eschewed real-world cases in favor of a fictional company, AllScale. This fictional company was an aggregation of many of our clients, their challenges, and their successes. While AllScale provided value in highlighting the lessons we hoped to convey, it was nevertheless a piece of fiction with limited ability to connect users to real-world successes and failures.

In this second edition, sometimes the stories we present are vignettes about an individual, as in the case of Steve Jobs at Apple or Jeff Bezos at Amazon. In other stories, we outline the success or failure of a company broadly and then point to key elements of that success or failure to help make a point.

Why People?

From our perspective, people aren't simply important to scalability, they are *the single most important* piece to get right if you hope to scale a product. The upside is that without people, you'll never have scalability problems; the downside is that you'll never develop a product that *needs* to scale. People architect the systems, develop or choose the software, and deploy the software that runs your product. People configure (or write the scripts that configure) the servers, databases, firewalls, routers, and other devices. Through acts of omission or commission, people make the decisions on which pieces of the product will succeed or fail under intense demand. People design the processes that companies use to identify current and future scalability issues. Initiatives aren't started without people, and mistakes aren't made without people. People, people, people...

All of the greatest successes in building scalable products have at their heart a great set of people making many great decisions and every once in a while a few poor choices. Ignoring the people element of scale is a very large mistake—one that is often found at the root of products that fail to meet end-user demands.

Given that people are the heart and brains behind scalability, it makes sense that we should spend a great deal of time and effort on attracting and retaining the best folks available. As we will discuss in Chapter 5, Management 101, this quest is not just about finding the people with the right and best skills for the amount you are willing to pay. Rather, to be truly successful, you must have the right person, with the right behaviors, in the right job, at the right time.

The *right person* speaks to whether the individual has the right knowledge, skills, and abilities. Putting this person in the right job at the right time is about ensuring that he or she can be successful in that position and create the most shareholder

value possible while tending to his or her career. The *right behaviors* mean that the person works and plays well with others while adhering to the culture and values of the company. Bad behaviors are as valid a reason for removing a person from the team as not having the requisite skills, because bad behavior in any team member creates a vicious cycle of plummeting morale and productivity.

Perhaps no story better emblemizes the traits of the right person and the right behavior at the right time than both the firing and the subsequent rehiring of Silicon Valley legend Steve Jobs. Walter Isaacson's masterful biography, *Steve Jobs*, paints a picture of a man who, in the spring of 1985 at the age of roughly 30, was destroying the company with his childish, rude, selfish, and often disruptive behavior. Morale was plummeting within Apple. While clearly John Sculley (Jobs's replacement) was not the answer, Jobs's lack of teamwork and constantly changing views of an appropriate product left the company devoid of focus and identity. After leading the launch of the world-changing Macintosh, Jobs apparently refused to believe or accept that the Apple II continued to drive a majority of the company's sales. His derisive behaviors sowed discontent within the product teams—dissent that ran counter to a culture of learning from mistakes and capitalizing on successes throughout the company. Jobs's *behaviors*, at the time, were simply not correct for what the company needed as a leader.

Now fast forward a little more than a decade. In 1996, Jobs was brought back in as Apple's CEO, through the acquisition of NeXT and the subsequent ouster of Gil Amelio. Perhaps as a result of his experiences at both NeXT and Pixar, Jobs had matured. While he retained the maniacal focus on his vision of the perfect product, he had at least tempered many of the liabilities that resulted in his abrupt dismissal in 1985. Jobs returned to Apple as a more disciplined man, with all of the innate skills he had as a brash 30-year-old, but also with many more of the leadership attributes necessary to turn around a company. This is an example of how (as we describe later in this book) leaders can, in fact, be made—or at least made better. While Jobs was the *right person* in 1985, he did not have the *right behaviors* at that time. When he returned in 1996, he was clearly the *right person* with the *right behaviors* in the *right job* at the *right time*. Note that the "right behaviors" here doesn't mean that Jobs was a nice guy; in fact, Isaacson indicates he was anything but a nice guy. Rather, the right behaviors means that the net results of his behaviors were generally a positive influence on Apple's growth.

Why Organizations?

It should follow that if people are the most important element to the scalability of a system, the way in which we structure them to get the job done is also important. To be successful in designing an organization, we must first understand our desired

outcomes. Any organizational structure has benefits and drawbacks relative to the goals we hope to achieve. Here we offer a few questions to consider regarding how organizational structure can positively or negatively impact desired outcomes:

- How easily can I add or remove people to/from this organization? Do I need to add them in groups, or can I add individual people?
- Does the organizational structure help or hinder the development of metrics that will help measure productivity?
- Does the organizational structure allow teams to own goals and feel empowered and capable of meeting them?
- Which types of conflict will arise, and will that conflict help or hinder the mission of the organization?
- How does this organizational structure help or hinder innovation within my company?
- Does this organizational structure help or hinder the time to market for my products?
- How does this organizational structure increase or decrease the cost per unit of value created?
- Does work flow easily through the organization, or is it easily contained within a portion of the organization?

The question of how easily people are added is an obvious one. It is very difficult to increase the amount of work done by an organization if the structure does not allow for bringing in more people to perform additional work. Great organizational structures support the addition of people in small or large numbers, allowing new teams to be formed or people to be added to existing teams. They also enable the organization to flex “down” when and if the company falls on hard times.

The question regarding metrics is important because organizational output is a function both of size (the number of people doing work) and efficiency (the output per individual or team). Sometimes the means of achieving greater output isn’t “more people” but rather “more work per person” or “more total output at the same team size.” Without understanding individual or organizational efficiency (metrics such as throughput, availability, time to market, and cost per unit produced), how can we determine if it makes sense to spend money on productivity tools or additional headcount? Most of us wouldn’t consider driving a car any considerable distance if that car didn’t at least have a gas gauge and a speedometer. The same should be true about our organizations: we shouldn’t run them without key performance indicators that help us achieve our desired results. Management means measurement, and a failure to measure means a failure to manage.

If the structure of your organization makes it difficult to establish measurements for individual performance, you will not be able to measure output. If you cannot measure the output and quality of the organization's work, you can't react to sudden and rapid deteriorations in output.

Team members' perception of whether they "own" goals and are empowered to deliver on them speaks to the autonomy of the team in achieving and owning its goals or, conversely, to its reliance upon other teams to meet its goals. Empowered teams generally experience higher morale, lower employee turnover, and faster time to market than teams that do not feel empowered. Fundamental to empowerment is the ability to own the decisions necessary to achieve a goal. To feel real ownership of their work, teams must feel they have the tools and capabilities to act upon their decisions. The teams that report the highest level of empowerment and ownership over goals are those that are cross-functional in nature and have all the skill sets necessary to achieve the goals set for their teams.

The question about the types of conflict that arise within this organizational structure, and whether that conflict helps or hinders the mission, addresses the two primary types of conflicts within an organization (cognitive and affective) and their relationships with the organization's outcomes. Affective ("bad") conflict is role- or control-based conflict and often arises between teams. Affective conflict is often about "who" does something or "how" something will be done. In organizational structures where multiple groups are necessary to deliver a product, such as "Operations," "Development," and "Quality Assurance," the real question may be about "who" gets to define when something is ready for launch or "how" it should be developed to meet customer needs. How much input does the infrastructure team have into the way in which applications will interact with databases? How much input does the software team have into the definition of the deployment architecture? Who should make those decisions? Affective conflict rarely adds value to products; instead, it almost always delays the launch of products and increases their costs. Further, when left unhandled, it decreases employee morale, increases employee turnover, and diminishes the level of innovation within a company.

Cognitive conflict, in contrast, if handled properly, is often called "good conflict." Most often cognitive conflict is related to "why" something must happen or "what" a company needs to do achieve some desired outcome. Imagine some of the best brainstorming sessions in which you've ever participated, or a genuinely effective prioritization meeting where a team really came together, made difficult decisions, and felt good about the results. Cognitive conflict expands our range of strategic possibilities. It increases the probability that we will make the right decisions by relying on diverse knowledge, skill sets, and experiences to cover the intersection of that which is imaginable and possible. More than likely you've had at least one experience where you've been in a room with someone who lacked your

technical knowledge and expertise, but who nevertheless asked a question that completely changed your approach to a problem.

These two types of conflict, when joined together, help to define how we should think about forming teams. To reduce affective conflict, we want cross-functional teams capable of owning the outcome to a goal in its entirety. That means embedding each of the necessary skill sets into the team tasked with delivering some solution. This same approach—that is, the embedding of diverse skill sets within a team—also allows us to increase the levels of cognitive or positive conflict within a team. Whenever we can create such cross-functional teams, we know that morale increases, empowerment increases, employee turnover decreases, time to market decreases, and innovation increases.

The next two questions “How does this structure help or hinder innovation?” and “How does this structure help or hinder time to market?” are at least partially informed by the previous answers. Everyone might agree that they are important questions to answer, but their answers are a bit involved. We will address these issues in Chapter 3, Designing Organizations, during a more in-depth discussion of organizational structure.

The relationship between organizational structure and the cost per unit of value created brings up the issue of what we like to call the “organizational cost of scale.” In his book *The Mythical Man-Month*, Fred Brooks indicated that there is a point in the life of a software project when adding people to a project can actually cause the project to be delayed further (i.e., be delivered even later). Brooks points out that one reason for this further delay is the communication overhead associated with the addition of each new team member to the project. As the team size grows, the increase in the overhead per team member is not linear. In other words, the more people you add to a project or task, the greater the overhead *per person* in communication and coordination costs.

Have you ever been in an organization where you receive hundreds of internal emails each day and potentially dozens of meeting invitations/requests each week? If so, you have undoubtedly spent valuable time deleting the emails and declining requests that aren’t relevant to your job responsibilities. The more people within your company, group, or team, the more time you must spend reading and deleting emails as well as going to meetings rather than doing “real” work. This scenario perfectly illustrates why adding people can often decrease the output of each individual within an organization. As in the preceding example, when you add people, the email volume grows and communication time correspondingly increases. Figure 1.1 depicts an engineering team attempting to coordinate and communicate. Table 1.1 shows the increase in overall output, but the decrease in individual output, as the team size increases from one to three. In Table 1.1, the individual loss of productivity due to communication and coordination is 0.005, which represents 2.4 minutes of coordination activity in

One Person
Very Little Coordination
and Communication
Overhead



Three People Communicating
and Coordinating Tasks



Figure 1.1 *Coordination Steals Individual Productivity*

an 8-hour day. This isn't a lot of time, and most of us intuitively would expect that three people working on the same project will spend at least 2.4 minutes each day coordinating their activities even with a manager! One person, in contrast, need not perform any of this coordination. Thus, even as individual productivity drops, the team output increases.

You can offset but not completely eliminate this deterioration in a number of ways. One possibility is to add management to limit nonessential coordination. Another possibility is to limit the interactions between individuals by creating smaller self-sufficient teams. Both of these approaches have benefits and drawbacks,

Table 1.1 *Individual Loss of Productivity as Team Size Increases*

Organization Size	Communication and Coordination Cost	Individual Productivity	Organization Productivity
1	0	1	1
3	0.005	0.995	2.985
10	0.010	0.99	9.9
20	0.020	0.98	19.6
30	0.05	0.95	28.5

as we will discuss in Chapter 3. Many other approaches are possible, of course, and anything that increases individual throughput without damaging innovation should be considered.

“Does work flow easily and quickly through the organization, or is it easily contained within a portion of the organization?” is meant to focus on the suitability of your organizational design to the type of work you do. Does work flow through your organization as efficiently as a well-defined assembly line? Waterfall processes resemble assembly lines; if your company employs such a process, having functional teams that hand off work to one another can make a lot of sense. Do your product architecture and the process you employ (such as Agile) allow the work to start and end within a single functionally diverse team? To achieve such a structure, the components of our architecture must work independently with minimal communication overhead between teams. An important point here—and a lesson that we will explore later in the book—is that our organizations, processes, and technology must work well together. Agile processes and functionally oriented teams working on monolithic, highly interdependent architectures can run into as many problems as waterfall processes with cross-functional teams working on largely independent and horizontally scaled product architectures.

One well-known company, Amazon, clearly demonstrates the value of ensuring the answers to our questions align to our end goals. You have likely heard of the “Two-Pizza Team” concept made famous at Amazon. In one off-site retreat, the executives at Amazon were discussing the need for better communication among teams. Jeff Bezos, the founder and CEO, jumped into the discussion by saying, “No, communication is terrible!”¹ Bezos was recognizing the cost of communication overhead as described in Table 1.1 and which Fred Brooks identified in *The Mythical Man-Month*. Bezos’s fix was to create the “Two-Pizza Team” rule: no team should be larger than can be fed with two pizzas. The idea here is that communication occurs primarily in the team and, as a result, communication overhead is significantly reduced. To be successful, however, the teams must be empowered to achieve their goals. Furthermore, each team must be given one or more key performance indicators (KPIs) that are indicative of overall success. The teams are often staffed cross-functionally so they have the skills necessary to achieve their KPIs without outside help.²

Bezos’s hope in creating the “Two-Pizza Team” rule was to engender higher levels of innovation, faster time to market, and lower levels of affective conflict within

1. Alan Deutschman. “Inside the Mind of Jeff Bezos.” *Fast Company*. <http://www.fastcompany.com/50106/inside-mind-jeff-bezos>.

2. Stowe Boyd. “Amazon’s Two Pizza Teams Keep It Fast and Loose.” *GigaOm*. <http://research.gigaom.com/2013/08/amazons-two-pizza-teams/>.

teams. He explicitly designed the teams with KPIs in mind for the purposes of achieving measurability. The solution is highly scalable. Need more capacity? Add more teams dedicated to specific aspects of the product. Finally, work is easily contained within each team. With one simple construct, all of our key questions are addressed.

Having discussed why organization design is important to scale, let's now turn our attention to why management and leadership are so important.

Why Management and Leadership?

Most STEM (science, technology, engineering, and math) college graduates have never had academic or foundational exposure to the concepts of management and leadership. Few universities offer such classes, unless you happen to have been a management major or attended an MBA program with a management curriculum. As a result, most of the managers within product organizations learn how to manage and how to lead informally. That is, they watch what others do in peer positions and positions of greater responsibility, and they make their own decisions about what works and what doesn't. Over time, these folks start to develop their own "toolboxes." Some add tools from their professional readings or discard tools as the approaches age and potentially become less relevant to managing generations of employees. This general "life as a lab" approach is how we've developed managers for years. Despite the benefits of this approach, it is unfortunate that management and leadership don't get more respectful and thorough treatment in structured curricula both within universities and within larger corporations.

Management and leadership can either multiply or detract from your ability to scale organizations in growth environments. Although they are often perceived as existing within the same context, they are really two very different disciplines with very different impact on scalability. Many times, the same person will perform the functions of both a leader and a manager. In most organizations, a person tends to progress from a position of an individual contributor into a primarily management-focused role; over time, with future promotions, that person will take on increasing leadership responsibilities.

In general, we like to think of management activities as "pushing" activities and leadership as "pulling" activities. Leadership sets a destination and "waypoints" toward that destination. Management then gets you to that destination. Leadership would be stating, "We will never have a scalability-related downtime in our systems"; management would be ensuring that this scenario never happens. Every company absolutely needs both management and leadership, and needs to do both well. Table 1.2 is a non-exhaustive list of some activities and their classification as either leadership or management oriented.

Table 1.2 Example Leadership and Management Activities

Leadership Activities	Management Activities
Vision setting	Goal evaluation
Mission setting	KPI measurement
Goal setting	Project management
Culture creation/culture setting	Performance evaluation
Key performance indicator (KPI) selection	People coaching
Organizational inspiration	People mentoring
Standard setting	Standard evaluation

Far too often, we get caught up in the notion of a “management style.” We might believe that a person’s “management style” makes him or her more of a leader or more of a manager. This notion of style represents our perception of an individual’s bias toward the tasks that define either leadership or management. We might believe that a person is more operationally focused and therefore more of a “manager,” or we might perceive someone as being more visionary and therefore more of a “leader.” Although we all have a set of personality traits and skills that likely make us more comfortable or more capable with one set of activities relative to the other, there is no reason we can’t get better at both disciplines. Recognizing the distinction between the two disciplines is a step toward isolating and developing both our management and leadership capabilities to the benefit of our stakeholders.

As we have indicated, management is about “pushing.” Management ensures that people are assigned to the appropriate tasks and that those tasks are completed within the specified time interval and at an appropriate cost. It ensures that people get performance-oriented feedback in a timely manner and that the feedback includes both praise for great performance and information about areas needing improvement. Management focuses on measuring and improving everything that ultimately creates shareholder value, such as reducing the cost to perform an activity or increasing the throughput of an activity at the same cost. Management means communicating status early and often, and clearly identifying what is on track and where help is needed. Management activities also include removing obstacles or helping the team over or around obstacles where they occur on the path to an objective. Management is important to scale because it is how you get the most out of an organization, thereby reducing cost per unit of work performed. The definition of how something is to be performed is a management responsibility, and how something is performed absolutely impacts the scale of organizations, processes, and systems.

Management as it relates to people addresses the need to have the right person, in the right job, at the right time, with the right behaviors. From an organizational

perspective, it is about ensuring that the team operates cohesively and effectively, includes the proper mix of skills, and has appropriate experiences to be successful. Management as applied to an organization's work makes sure that projects are on budget, are on time, and deliver the expected results upon which their selection was predicated. Management means measurement, and a failure to measure is a failure to manage. Failing to manage, in turn, guarantees that you will miss your organizational, process, and systems scalability objectives—without management, no one is charged with ensuring that you are doing the things you need to do in the time frame required.

Leadership, by comparison, has to do with all the “pulling” activities necessary to be successful in any endeavor. If management is the act of pushing an organization up a hill, then leadership is about selecting that hill and surmounting it first, thereby encouraging the rest of the organization to follow suit. Leadership is about inspiring people and organizations to be better and do great things. It creates a compelling vision that tugs on the heartstrings of employees and “pulls” them to do the right thing for the company. Leadership identifies a mission that helps codify the vision and develops a causal mental roadmap that helps employees understand how what they do creates value for the shareholders. Finally, leadership defines the objectives to be met on the path toward the organization’s goals and establishes KPIs. Leadership is important to scale because it not only sets the direction (mission) and destination (vision), but also inspires people and organizations to journey toward that destination.

Any initiative lacking leadership (including initiatives meant to increase the scalability of your company), while not doomed to certain failure, will likely achieve success only through pure dumb luck and chance. Great leaders create a culture focused on ensuring success through highly scalable organizations, processes, and products. This culture is supported by incentives structured around ensuring that the company scales cost-effectively without experiencing user-perceived quality of service or availability issues.

Conclusion

We’ve asserted that people, organizations, management, and leadership are all important to scalability. People are the most important element of scalability: without people, there are no processes and no technology. The effective organization of your people will either get you to where you need to be faster or hinder your efforts in producing scalable systems. Management and leadership are the push and pull, respectively, in the entire operation. Leadership serves to inspire people to greater accomplishments, and management exists to motivate them in meeting those objectives.

Key Points

- People are the most important piece of the scale puzzle.
- The right person in the right job at the right time and with the right behaviors is essential to scale organizations, processes, and systems.
- Organizational structures are rarely “right or wrong.” Any structure is likely to have pros and cons relative to your needs.
- When designing your organization, consider the following points:
 - The ease with which additional units of work can be added to the organization. How easily can you add or remove people to/from this organization? Do you need to add them in groups, or can you add individual people?
 - The ease with which you can measure organizational success and individual contributions over time.
 - How easily goals can be assigned and owned by groups, and whether the groups will feel empowered to deliver these goals.
 - How conflict will emerge within and between groups, and whether it will help or hinder the company mission.
 - How the structure will help or hinder both innovation and time to market.
 - How the organizational structure will increase or decrease the cost per unit of value created.
 - How easily work flows through the organization.
- Adding people to organizations may increase the organizational throughput, but the average production per individual tends to decline.
- Management is about achieving goals. A lack of management is nearly certain to doom your scalability initiatives.
- Leadership is about goal definition, vision creation, and mission articulation. An absence of leadership as it relates to scale is detrimental to your objectives.

Chapter 2

Roles for the Scalable Technology Organization

When the general is weak and without authority; when his orders are not clear and distinct; when there are no fixed duties assigned to officers and men, and the ranks are formed in a slovenly haphazard manner, the result is utter disorganization.

—Sun Tzu

A common cause of failures in scalability and availability is lack of clarity in responsibilities of people. In this chapter, we start by taking a look at two very real examples of what might happen without role clarity and clearly defined responsibilities. We also discuss executive roles, example organizational responsibilities, and the roles of various individual contributors. We conclude the chapter by introducing a tool that can aid in responsibility definition and help to reduce role-based (or affective) conflict.

This chapter is meant for companies of all sizes. For large companies, it can serve as a checklist to ensure that you have covered all of the technology and executive roles and responsibilities as they relate to scale. For small companies, it can help jumpstart the process of ensuring that you have your scalability-related roles properly defined. For technology neophytes, it serves as a primer for how technology organizations should work; for seasoned technology professionals, it is a reminder to review organizational structure to validate that your scalability-related needs are covered. For all companies, it clearly defines why everyone in a company has a role in creating scalable products.

The Effects of Failure

Sometimes, a lack of clarity in roles and responsibilities means that some things won't get done. Consider, for instance, the case where no team or individual is

assigned the responsibility of capacity planning. In this context, capacity planning compares expected demand with a product's capabilities to fulfill that demand. This comparison should ultimately result in a set of proposed actions to ensure that the product capacity matches or exceeds the demand. The forecasted number of requests defines the expected demand placed on the product in question. The proposed set of actions may include requesting the purchase of additional servers, modifying software to be more efficient, or adding persistent storage systems such as databases or NoSQL servers.

In this case, the absence of a team or person responsible for matching expected demand to existing capacity and determining appropriate actions would be disastrous in an environment where demand is growing rapidly. Nevertheless, this failure happens all the time—especially in young companies. Even companies that have a person or organization responsible for capacity planning often fail to plan for their newest system additions.

At the other end of the spectrum from the “missing an owner” problem is the case where multiple organizations or people are given the same goal. Note that this is not the same thing as “sharing a goal.” Sharing a goal is good, because “sharing” implies cooperation. Rather, what we are describing here is the existence of multiple owners without clarity as to who owns which actions to achieve that goal and sometimes without understanding that another group or person is working on the same thing. If you work in a smaller company where everyone knows what everyone else is doing, this concern may seem a bit ridiculous to you. Unfortunately, this problem exists in many of our larger client companies—and when it happens, it not only wastes money and destroys shareholder value, but can also create long-term resentment between organizations and destroy employee morale. Few things will destroy morale more quickly than a group believing that it is fully responsible but failing to deliver on a task or project because the group members assumed someone else owned a part that did not complete. The multiple-owner problem is often at the heart of the role-based (“affective”) conflict that we described in Chapter 1, *The Impact of People and Leadership on Scalability*.

As an example, let’s assume that an organization is split between an engineering organization, which is primarily responsible for developing software, and an operations organization, which is primarily responsible for building, deploying, and operating databases, systems, and networks. Let’s further assume that we have a relatively inexperienced CTO who has recently read a book on the value of shared goals and objectives and has decided to give both teams the responsibility of scaling the platform to meet expected customer demand. The company has a capacity planner who determines that to meet next year’s demand, the teams must scale the subsystem responsible for customer contact management to handle at least twice the number of transactions it is capable of handling today.

Both the engineering and operations teams have architects who have read the technology section of our book, and both decide to make splits of the database supporting the customer contact management system. Both architects believe they are empowered to make the appropriate decisions without the help of each other, as they are unaware that multiple people have been assigned the same responsibility and were not informed that they should work together. The engineering architect decides that a split along transaction boundaries (or functions of a Web site, such as buying an item and viewing an item on an ecommerce site) will work best. In contrast, the operations architect decides that a split along customer boundaries makes the most sense, such that various groups of customers will reside in separate databases. Both architects go about making initial plans for the split, setting their teams to doing their work and then making requests of the other team to perform some work.

This example may sound a bit ridiculous to you, but it happens all the time. In the best-case scenario, the two teams will stop there and resolve the issue having “only” wasted the valuable time of two architects. Unfortunately, what usually happens is that the teams become polarized and waste even more time in political infighting, and the final, probably delayed result isn’t materially better than if a single person or team had the responsibility to craft the best solution with the input of other teams.

Defining Roles

This section gives an example of how you might define roles to help resolve role- and responsibility-based conflict. It provides examples of how roles might be defined at the top leadership level of the company (the executive team), within classic technology organizational structures, and at an individual contributor level.

These examples of executive and individual contributor responsibilities are not meant to restrict you to specific job titles or organizational structure. Rather, they are intended to help you outline the necessary roles within a company and define certain responsibilities or professional focus areas. We’ve chosen to define these roles by the organizations in which they have traditionally existed to make them easier to understand for a majority of our audience. For instance, you may decide that you want operations, infrastructure, engineering, and quality assurance (QA) to exist within the same teams, with each team dedicated to a specific product line (in fact, as you will see in the next chapter, we strongly suggest such an organization structure).

You may recall from our introductory discussion that there is no “right or wrong” answer when designing an organizational structure—there is simply a set of benefits and drawbacks for any decision. The important point is to remember that you should include all of the appropriate responsibilities in your organizational design

and that you should clearly define not only who is a responsible decision maker but also who is responsible for providing input to any decision, who should be informed of the decision and actions, and who is responsible for making the decision happen. Perhaps the most critical element in making the best decisions consistently is having the best decision-making process in place, as that ensures the right information is gathered by the right people to inform the ultimate decision maker. That applies to anyone who must make decisions in the company. We'll discuss this last point in a brief section on a valuable tool later in this chapter. The appropriate organizational structure can help limit some of the conflict that may otherwise result from a lack of clarity, although conflict can never be completely eliminated. At the very least, as a leader, you should be clear about the responsibilities and desired outcomes for any group within your organization.

A Brief Note on Delegation

Before launching into the proposed division of responsibilities within an organization, we thought it important to include a brief note on delegation. In defining roles and responsibilities within organizations, you are creating a blueprint of delegation. Delegation, broadly speaking, is the act of empowering someone else to act on your behalf. For instance, by giving an architect or architecture team the responsibility to design a system, you are delegating the work of creating that architecture to that team. You may also decide to delegate the authority to make decisions to that team depending upon their capabilities, the size of your company, and so on.

Here's a very important point: You can delegate anything you like, but you can *never* delegate the accountability for results. At best, the team (or individual) to which you delegate can inherit that responsibility and you can ultimately fire, promote, or otherwise reward or punish the team for its results, but you should always consider yourself accountable for the end result. Great leaders get this point intuitively, and they put great teams ahead of themselves when lauding successes and take public accountability when acknowledging failures. Poor leaders, in contrast, assume that they can "pass the buck" for failures and take credit for successes.

To codify this point in your mind, let's apply "the shareholder test." Assume that you are the CEO of a company. You have decided to delegate the responsibility for one of your business units to a general manager. Can you imagine telling your board of directors or your shareholders (whom the board represents) that you will not be held accountable for the results of that business? Going one step further, do you think the board will not hold you at least partially responsible if the business begins to underperform relative to expectations?

Again, this does not mean that you should make all the decisions yourself. As your company and team grow and scale, you simply won't be able to do that; indeed, in many cases, you might not be qualified to make the decisions. For instance, a nontechnical CEO should probably not be making architecture decisions and a CTO of a 200-person engineering organization

should not be writing the most important code; those managers are needed in other executive tasks. This reality highlights that you absolutely must have the best people possible to whom you can delegate and that you must hold those people to the highest possible standards. It also means that you should be asking the best questions possible about how someone came to his or her decisions on the most critical projects and systems.

Executive Responsibilities

The executives of a company as a team are responsible more than anyone else for creating a culture of scalability as defined in Chapter 1, The Impact of People and Leadership on Scalability.

Chief Executive Officer

The CEO is the chief scalability officer of the company. As with all other matters within the company, when it comes to scale, the CEO is the final decision maker and arbiter of all things related to scale. A good technology company CEO needs to be appropriately technically proficient, but that does not mean that he or she needs to be the technology expert or the primary technology decision maker.

It is hard to imagine that someone could rise to the position of CEO and not understand how to read a balance sheet, income statement, or statement of cash flow. That same person, unless he or she has an accounting background or is a former CFO, is not likely to understand the intricacies of each accounting policy—nor should this individual need to. The CEO's job is to ask the right questions, to get the right people involved, and to solicit the right outside help or advice to arrive at the right answer.

The same holds true in the technical world—the CEO's job is to understand some of the basics (the equivalent of the financial statements mentioned previously), to know which questions to ask, and to know where and when to get help. Here is some advice for CEOs and other managers responsible for technical organizations who have not been the chief technology officer or chief information officer of a company, do not have technical degrees, or have never been an engineer.

Ask Questions and Look for Consistency in Explanations

Part of your job is to be a truth seeker, because only with the truth can you make sound and timely decisions. Although we do not think it is commonplace for teams to lie to you, it is very common for teams to have different pieces and perceptions of the truth, especially when it comes to issues of scale. When you do not understand

something, or something does not seem right, ask questions. When you are unable to distinguish fact from perception, look for consistency in answers. If you can get over any potential ego or pride issues when asking what might seem to be ignorant questions, you will find that you not only quickly educate yourself but also create and hone a very important skill in uncovering truth.

This *executive interrogation* is a key ability shared by many successful leaders. The Bill Gates version of this skill was known as the “BillG Review.”¹ Knowing when to probe and where to probe and probing until you are satisfied with answers need not be limited to the CEO. In fact, managers and individual contributors should all hone this skill—and start early in their careers when doing so.

Seek Outside Help

Seek help from friends or professionals who are proficient and knowledgeable in the area of scalability. Don’t bring them in and attempt to have them sort things out for you—that can be very damaging. Rather, we suggest creating a professional or personal relationship with a technically literate firm or peer. Leverage that relationship to help you ask the right questions and evaluate the answers when you need to deep dive.

Improve Your Scalability Proficiency

Create a list of your weaknesses in technology—things about which you have questions—and get help to become smarter. You can ask questions of your team and outside professionals. Read blogs on scale-related issues relevant to your company or product, and attend workshops on technology for people without technical backgrounds. You probably already do this through professional reading lists on other executive topics—add technology in general and scalability in particular to that list. You do *not* need to learn a programming language, understand how an operating system or database works, or understand how “collision detection multiple access/carrier detect” is implemented. Rather, you just need to be able to get better at asking questions and evaluating the issues your teams bring to you. Scalability is a business problem, but to solve it, you need to at least be somewhat conversant in the technology portion of the equation.

More than likely, the CEO will decide to delegate authority to several members of his or her team, including the chief financial officer, individual business unit owners (also known as general managers), and the head engineering and technology executive (referred to as either the CTO or the CIO in our book).

1. See Joel Spolsky. “My First BillG Review.” <http://www.joelonsoftware.com/items/2006/06/16.html>.

Chief Financial Officer

Most likely, the CEO has delegated the responsibility for budgeting to the CFO, although this may not always be the case. Budgeting is informed by capacity planning and—as we saw in our previous example of how things can go wrong—capacity planning is a very large part of successfully scaling a system. A key portion of the budgeting officer’s responsibility is ensuring that the team and the company have a sufficient budget to scale the platform/product/system. The budget needs to be large enough to allow the company to scale to the expected demand by purchasing or leasing servers and hiring the appropriate engineers and operations staff. That said, the budget should not be so large that the company spends money on scale long before it truly needs that capacity, because such spending dilutes near-term net income for very little benefit. Purchasing and implementing “just in time” systems and solutions will optimize the company’s net income and cash flow.

The CFO is also not likely to come from a very technical background, but can benefit from asking the right questions and creating an appropriate network, just as the CEO can. Questions that the CFO might ask regarding scalability include which other scale alternatives were considered in developing the proposed budget for scale and which tradeoffs were made in deciding upon the existing approach. The intent here is to ensure that the team considered more than one option. A bad answer would be “This is the only way possible,” as that is rarely the case. (We want to say it is never the case, but on rare occasions only one route may really be possible.) For example, if the company is facing challenges in lowering its expenses, a CFO might decide that older equipment such as servers and networking gear can still be used because its depreciation has fallen off the books. In reality, however, older equipment is usually much less efficient and much more prone to failure, which will cause its total cost of ownership to increase significantly. A good answer, then, might be “Of the options we evaluated, this one allows us to scale horizontally at comparatively low cost while setting us up to scale even more cost-effectively in the future by creating a framework from which we can continue our horizontal scale.”

Business Unit Owners, General Managers, and P&L Owners

More than any other position, the business unit general manager or owner of the company’s or division’s profit and loss statement (P&L; also called the income statement) is responsible for forecasting the platform/product/system-dependent business growth. In small- to medium-sized companies, it is very likely that the business unit owner will be the CEO and that he or she might delegate this responsibility to some member of her staff. Nevertheless, demand projections are critical to the activity of determining what needs to be scaled so that the budget for scale doesn’t become too large ahead of the actual corporate need for it.

Very often, we run into situations in which the business unit owner claims that demand simply can't be forecasted. Here, *demand* means the number of requests that are placed against a system or product. This is a punting of responsibility that simply should not be tolerated within any company. In essence, the lack of ownership for forecasting demand by the business means that this responsibility gets moved to the technology organization, which is very likely less capable of forecasting demand than the business unit owner is. Yes, your forecasts will probably be wrong, especially in their infancy, but it is absolutely critical that you start the process early in the life cycle of the company and mature it over time.

Finally, like the other senior executive staff of the company, the business unit owner is responsible for helping to create a culture of scalability. This individual should make sure to ask the right questions of his or her peers (or subordinates) in the technology organization and try to ensure that those technology partners receive the funding and support to properly assist the business units in question. Both efforts are essential to the success of scalability within the company.

Chief Technology Officer/Chief Information Officer

Whereas the CEO is the chief scalability officer of the company, the chief technology executive is the chief technology, technical process, and technology organization scalability officer. In companies where the technology is the primary product for the customers, particularly Internet companies, this executive is often titled the chief technology officer (CTO). In companies where technology plays more of a supporting role in producing or delivering a product for customers, this executive is often titled the chief information officer (CIO). In such circumstances, if there is a CTO, that person is generally more focused on the science or technology of the product. We will use CTO and CIO interchangeably throughout this book to mean the chief technology executive of the company. This individual most likely has the best background and capabilities to ensure that the company scales cost-effectively ahead of the product/system or platform needs.

In essence, "the buck stops here." Although it is true that the CEO can't really "delegate" responsibility for the success of the platform scalability initiatives, it is also true that the chief technology executive inherits that responsibility and shares it with the CEO. A failure to properly scale will likely at least result in the termination of the chief technology executive, portions of his or her organization, and potentially even the CEO.

The CTO/CIO must create the technical vision for the company overall. For the purposes of our discussion, within a growth company that vision must include elements of scale. The chief technology executive is also responsible for setting the aggressive, measurable, and achievable goals that are nested within that vision and

for ensuring that his or her team is appropriately staffed to accomplish the associated scalability mission of the organization. The CTO/CIO is responsible for the development of the culture and processes surrounding scalability that will help ensure that the company always stays ahead of end-user demand.

The CTO/CIO will absolutely need to delegate responsibilities for certain aspects of decision making around scalability as the company grows. Of course, as we pointed out previously, such delegation never eliminates the executive's own responsibility to ensure that scalability is done correctly, on time, and on budget. Additionally, in hyper-growth environments where scale is critical to company survival, the CTO should never delegate the development of the vision for scale. The term "lead from the front" is never more important than in this context, and the vision does not need to be deeply technical.

Although the best CTOs we have seen have had technology backgrounds varying from once having been an individual contributor to having been a systems analyst or technical project manager, we have seen examples of successful CTOs without such backgrounds. With a nontechnical CTO/CIO, it is absolutely critical that this individual has some technical acumen and is capable of speaking the language and understanding the critical tradeoffs within technology, such as the relationship of time, cost, and quality. Calling upon a technology neophyte to lead a technical organization is akin to throwing a non-swimmer overboard into a lake; you may be pleased with your results assuming the person can swim, but more often than not you will need to find a new partner in your boat. Equally important with respect to a neophyte CTO is gaining trust from the technical staff; without this trust from below, the CTO cannot hope to lead effectively.

Equally important is that the CTO have some business acumen. Unfortunately, this can be as difficult to achieve as finding a chief marketing officer with a Ph.D. in electrical engineering (not that you'd necessarily want one)—such people exist, but they can be challenging to find. Unfortunately, most technologists do not learn about business, finance, or marketing in their undergraduate or graduate courses. Although the CTO does not need to be an expert on capital markets (that's likely the job of the CFO), he or she should understand the fundamentals of the business in which the company operates. For example, the CTO should be able to analyze and understand the relationships between the income statement, the balance sheet, and the statement of cash flow. In a technology-centric company like an Internet firm, the CTO will often have the largest budget in the company. Such an executive will often be charged with making very large investments in capital, telecommunications carriers, data center leases, and software licensing—and his or her lack of understanding of financial planning may expose the company to large risks in capital misappropriation, overpaying vendors, and unplanned and unforeseen expenses (such as periodic software license "true-ups"). The CTO should also understand

marketing basics at the level of at least a community college– or company-sponsored course on the subject. This is not to say that the CTO needs to be an expert in any of these areas; rather, a basic understanding of these topics is critical to making the business case for scalability and to being able to communicate effectively in the business world. We'll discuss these areas in later chapters.

Individual Contributor Responsibilities

Here we describe the roles of individual contributors that most companies will need as they grow and scale. These roles include ensuring that the company has people to cover the overall architecture of the product (architecture), the software engineering of the product (engineering), the monitoring and production handling of the product (operations), design and deployment of hardware for the product (infrastructure engineering), and the testing of the product (quality assurance).

Our goal here is not to define strict organizational or functional boundaries. As mentioned earlier, organizing by function is one way of structuring an organization; in particular, it is an effective approach when employing classical waterfall approaches to development. As we will describe in Chapter 3, Designing Organizations, when companies focus on developing products and need to do product discovery, we prefer organizations that are aligned with the products a company produces and processes that enable product discovery.

Architecture Responsibilities

Architects are responsible for ensuring that the *design* and *architecture* of the system allow for scale in the time frame appropriate to the business. Here, we clearly indicate a difference between the intended design and the actual implementation. Architects need to think well in advance of the needs of the business and should have thought through how to scale the system long before the business unit owners forecast demand exceeding the platform capacity at any given time. For instance, your architects may have developed an extensible data access layer (DAL) or data access object (DAO) that can allow for multiple physical databases to be accessed with varying schemas as user demand increases in any given area. The actual implementation may be such that only a single database is used, but with some cost-effective modification of the DAL/DAO and some work creating migration scripts, additional databases can be developed in the production environment in a matter of weeks rather than months should the need arise. Architects are also responsible for creating the set of architecture standards by which engineers design code and implement systems.

Architects are responsible for designing a system and having designs ready to solve any scale-related issue. In Part II, “Building Processes for Scale,” we identify a key process that the architecture team should adopt to help identify scale-related problems across all of the technology disciplines.

Architects may also be responsible for forming information technology (IT) governance, standards, and procedures, and for enforcing those standards through such processes as the Architecture Review Board discussed in Chapter 13, Joint Architecture Design and Architecture Review Board. When architects perform these roles, they do so at the request of the chief technology executive. Some larger companies may create process-engineering teams responsible for procedure definition and standards enforcement.

Companies use many different titles to designate the individuals who are focused on designing the technology architecture, including software architect, systems architect, scalability architect, enterprise architect, and even Agile architect. These roles, no matter the title used, tend to focus on one of two areas. The first is software architecture, which is primarily concerned with how the software is designed and structured. These individuals focus on areas such as service-oriented architecture frameworks or coding patterns that provide guidance and governance to developers. The second is system architecture, which is primarily concerned with how the software is deployed and supported on the hardware (or virtual machines). These individuals focus on areas such as eliminating single points of failure, fault isolation, and capacity planning. From the customer’s perspective, the scalability of a system is mostly concerned with the system architecture. This important role is covered in more depth in the discussion of individual contributor responsibilities later in this section.

Engineering Responsibilities

Engineers are “where the rubber meets the road.” Engineers are the chief implementers of the scalability mission and the chief tuners of the product platform. They take the architecture and create lower-level designs that they ultimately implement within code. They are responsible for adhering to the company’s architectural standards. Engineering teams are one of the two or three teams most likely to truly understand the limits of the system as implemented, given that they are one of the teams with the greatest daily involvement with that system. As such, they are key contributors to the process of identifying future scale issues.

DevOps Responsibilities

DevOps, which is a portmanteau of the words “development” and “operations,” is a recent phenomenon that encompasses almost anything that facilitates the interaction

between the software development teams and the technical operations teams. This term represents an acknowledgment of something that many of us have known for years—that a “system” or “service” requires both software and hardware. Thus the two skill sets of software development and systems administration are blended in DevOps.

In the Software as a Service (SaaS) and Web 2.0 worlds, DevOps is usually responsible for deploying, running, and monitoring the systems that create the company’s revenue. This team should be part of the Agile development efforts described in detail later in this book. If you do not employ either our Agile organizational structure or Agile development practices, you might still at least consider embedding DevOps personnel within engineering teams to help them better understand the deployment architectures of the solutions they create. Given that the team interacts with how the system runs every day and has daily insight into system utilization data, these team members are uniquely qualified to identify bottlenecks within the system and to design software with deployment in mind.

Often, DevOps is responsible for development/testing environments, build scripts, deployment scripts, monitoring solutions, log file aggregation, and development or testing tools. DevOps personnel might be responsible for creating reports that show trended availability over time, bucketing root cause and corrective actions, and determining mean time to resolution and mean time to restoration for various problems.

Regardless of the composition of the team, the organization responsible for monitoring and reporting on the health of systems, applications, and quality of service plays a crucial role in helping to identify scale issues. The processes that this group employs to manage issue and problem resolution should feed information into other processes that help identify scale issues in advance of major outages. The data that the operations organization collects is incredibly valuable to those performing capacity planning as well as to those responsible for designing away systemic and recurring issues such as scale-related events. The architecture and engineering teams rely heavily on DevOps to help them identify what should be fixed and when. We discuss some of these processes in Part II—more specifically in Chapter 8, Managing Incidents and Problems, and Chapter 13, Joint Architecture Design and Architecture Review Board.

Infrastructure Responsibilities

Infrastructure engineers are contributors whose skill sets are unique enough to not be needed on a day-to-day basis on the Agile teams, but rather are required across many Agile teams. Some large companies embed these teams into Agile teams to develop end-to-end solutions, but in most medium- and small-sized companies they work in centralized infrastructure teams that support multiple engineering teams.

Such a group can include database administrators, network engineers, systems engineers, and storage engineers. They are often responsible for defining which systems will be used, when systems should be purchased, and when systems should be retired. Whether embedded within smaller product teams or not, their primary responsibilities include architecting shared resources (such as network and transit resources), defining storage architectures globally or for a specific product team, and defining database or nonrelational solutions.

Within companies hosted in a cloud environment (e.g., Infrastructure as a Service [IaaS] providers like Amazon Web Services) the infrastructure team is often responsible for managing the virtual machine instance, the network (i.e., load balancers), and security (i.e., firewalls). Infrastructure engineers also aid in identifying capacity constraints on the systems, network devices, and databases that they support and help determine appropriate fixes for scale-related issues. Finally, they are responsible for making sure the systems they put in place can be managed and monitored at scale.

Quality Assurance Responsibilities

In the ideal scenario, the individuals who are primarily responsible for testing an application to ensure it is consistent with the company's desired product outcomes (KPIs) will also play a role in advanced testing for scale. New products, features, and functionality change the demand characteristics of a system, platform, or product. Most often, we are adding new functions that by definition create additional demand on a system. Ideally, we can profile that new demand creation to ensure that the release of our new functionality or features won't have a significant impact on the production environment. QA professionals need to be aware of all other changes going on around them so that they can ensure any scale-related testing done is updated in a timely fashion. These folks must also be focused on automated testing rather than manual testing: Not only must the products scale, but the process by which they are tested must scale to run efficiently as new features are added.

Capacity Planning Responsibilities

Individuals with capacity planning responsibilities can reside nearly anywhere, but they need access to up-to-date information regarding system, product, and platform performance. Capacity planning is a key to scaling efficiently and cost-effectively. When performed well, the capacity planning process results in the timely purchase of equipment where systems are easily horizontally scaled, the emergency purchase of larger equipment where systems cannot yet be scaled horizontally, and the identification of systems that should have a high priority on the list of scale-related problems to correct.

You might notice that we use the word *emergency* when describing the purchase of a larger system. Many companies take the approach that “scaling up” is an effective strategy. Our position, as we will describe in Chapters 21 through 25, is that if your scaling strategy relies on faster and bigger hardware, your solution does not scale. Rather, you are relying on the scalability of your providers to allow you to scale. Stating that you scale by moving to bigger and faster hardware is like stating that you become fast by buying a bigger, faster car. You have not worked to become faster, and you are only as fast as anyone else with similar wealth. Scalability is the ability to scale independently of bigger and faster systems or the next release of an application server or database.

What Happens When You Are Missing Roles and Skill Sets: An eBay Story

Have you ever wondered what would happen if you asked your carpenter to fix a plumbing problem in your house? Maslow’s Hammer (also known as “The Law of the Instrument”) predicts the outcome of such a situation: There is a good chance your carpenter will bring a hammer to get the job done. Chances are that you’ve heard of Maslow’s Hammer before—it is commonly described along the lines of “When all you have is a hammer, everything looks like a nail.”

This story starts with our partner Tom Keeven arriving at eBay in 2001. Tom walked into the operations center on one of his first days on the job as the VP of Infrastructure, pointed to some of the monitors on the wall displaying data, and casually asked, “What are those?”

One of the operators turned around, looked at Tom, and turned back to what he was doing. “Those are our schedulers,” replied the operator.

“And what is a ‘scheduler’?” asked Tom.

“They schedule requests coming in from end users to the Web servers,” replied the operator.

“Oh, so they are load balancers,” offered Tom.

“I don’t know what a load balancer is,” said the operator.

This may seem a bit wonky to you, but it is actually an accurate portrayal of an event that happened in 2001. The exact statements may be unintentionally altered, but the overall gist of the conversation is correct. Remember that products on the Internet had been around for only a few years and that not everyone understood many of the basic concepts in systems infrastructure that are widely taken for granted today.

Hardware load balancers—the devices to which Tom was referring—had been around for several years by 2001. They weren’t just used for Web products; they were used in all sorts of client-server architectures serving employees within corporations. The issue highlighted by this discussion will become clearer in a minute.

Tom decided to find out more about the “Schedulers” that eBay was using at the time. He discovered that they had been in operation for at least a handful of years; prior to that eBay had used DNS (domain name server) round-robin techniques to schedule traffic to Web servers. (Incidentally DNS round robin is a technique that we still find in some of our clients’ operations—seriously!) Moreover, the team that had selected the solution included primarily software engineers without significant infrastructure experience.

Tom knew quite a bit about both software and hardware load balancers. He knew, for instance, that hardware load balancers tended to process more requests per second with lower failure rates than their software cousins (“Schedulers”). While more expensive on a per-device or per-license basis, after traffic throughput was taken into account, the cost per transaction processed was typically lower on the hardware devices. Factor in their higher overall availability (lower device failure rate) and hardware load balancers were a clear winner when choosing a solution for load balancing requests.

Tom quite often displays magical tendencies when it comes to infrastructure architecture, but there was nothing magical or mystical about his insights here. They were born of more than 20 years of hard-won knowledge fighting in the trenches of infrastructure, operations, and solutions architecture. Tom reviewed the incidents associated with the software load balancers along with their license costs and quickly came to the realization that he could both increase availability and decrease costs by switching to hardware load balancers.

The lesson here is that eBay was missing a key skill set—someone with both deep and broad infrastructure experience and knowledge. This isn’t to say that eBay didn’t have some great infrastructure people before Tom; it did. But the skill sets present at eBay didn’t cover a broad-enough range at the time of a key decision (load balancing solution) to keep the company from making a mistake. That mistake could just have easily been scalability related, such as not having a person or group of people paying attention to capacity management of the eBay platform.

Tom’s decision did ultimately lower eBay’s costs per transactions slightly and helped to improve its availability. Neither outcome was earth shattering in its magnitude, but both shareholders (through cost reduction) and customers (through improvement of availability) ultimately benefited. And as we say in our business, “Every little bit helps.”

A Tool for Defining Responsibilities

Many of our clients use a simple tool to help them define role clarity for any given initiative. Often when we are brought in to help with scalability in a company, we employ this tool to define who should do what, and to help eliminate wasted work and ensure complete coverage of all scalability-related needs. Although it is technically a process, as this is a chapter on roles and responsibility, we felt compelled to include this tool here.

The tool we use most often is called RASCI. It is a responsibility assignment chart, and the acronym stands for Responsible, Accountable, Supportive, Consulted, and Informed.

- R: Responsible. This is the person responsible for completing the project or initiative.
- A: Accountable. This is the person to whom R is accountable and who must approve the work before it is okay to complete. When using RASCI as a decision-making tool, the A is sometimes referred to as the *approver* of any key decision.
- S: Supportive. These people provide resources to complete the project or initiative.
- C: Consulted. These people have data or information that can be useful in completing the project.
- I: Informed. These people should be notified, but do not need to be consulted or provide input to the project.

RASCI can be used in a matrix, where each activity or initiative is spelled out along the *y*-axis (vertical axis) of the matrix and the individual contributors or organizations are spelled out on the *x*-axis of the matrix. The intersection of the activity (*y*-axis) and the organization (*x*-axis) contains one of the letters R, A, S, C, or I, but may include nothing if that individual or organization is not part of the initiative.

Ideally, in any case, there will be a single R and a single A for any given initiative. This helps eliminate the issue we identified earlier in this chapter—that is, having multiple organizations or individuals believe that they are responsible for any given initiative. By holding a single person or organization responsible, you are abiding by the “one back to pat and one throat to choke” rule. A gentler way of saying this is that distributed ownership is ownership by no one.

This is not to say that others should not be allowed to provide input into the project or initiative. The RASCI model clearly allows and enforces the use of consultants or people within and outside your company who might add value to the initiative. An A should not sign off on an R’s approach until such time as the R has actually consulted with all of the appropriate people to develop the right course of action. Of course, if the company has the right culture, not only will the R want to seek those people’s help, but the R will make them feel as if their input is valued and value is added to the decision-making process.

You can add as many C, S, and I personnel whom you would like and who add value or who are needed to complete any given project. That said, guard against going overboard in terms of who exactly you will inform. Remember the earlier point about people getting bogged down with email and communication that does not concern them. Young companies often assume that everyone should feel involved in every decision or be informed of every decision. This information distribution

mechanism simply does not scale, however—and results in people reading emails rather than doing what they should be doing to create shareholder value.

A partially filled-out example matrix is included in Table 2.1. Based on our discussion thus far regarding different roles, let's see how we've begun to fill out this RASCI matrix.

Earlier, we indicated that the CEO absolutely must be responsible for the culture of scalability, or the *scale DNA* of the company. Although it is theoretically possible for the CEO to delegate this responsibility to someone else within the company from a practical perspective, and as you will see in the chapter on leadership, this executive must live and walk the values associated with scaling the company and its platform. As such, even with delegation and as we are talking about how the company "acts" with respect to scale, the CEO absolutely must "own" this responsibility. Therefore, we have placed an R in the CEO's column next to the Scalability Culture initiative row. The CEO is obviously responsible to the board of directors

Table 2.1 RASCI Matrix

	Business								Board of Directors
	CEO	Owner	CTO	CFO	Arch	Eng	Ops	Inf	QA
Scalability Culture	R								A
Technical Scalability Vision	A	C	R	C	S	S	S	S	I
Product Scale Design				A		R			
Software Scale Implementation				A			R	S	
Hardware Scale Implementation				A			S	R	
Database Scale Implementation				A			S	R	
Scalability Implementation Validation				A					R

and, as the creation of scale culture has to do with overall culture creation, we have indicated that the board of directors is the A.

Who are the S people in the Scalability Culture initiative? Who should be informed and who needs to be consulted? In developing your answer to this question, you are allowed to have people who are supportive (S) of any situation also be consulted (C) in the development of the solution. It is implied that C and S staff will be informed as a result of their jobs, so it is generally not necessary to include an I wherever you feel you need to communicate a decision and a result.

We've also completely filled out the row for Technical Scalability Vision. Here, as we've previously indicated, the CTO is responsible for developing the vision for scalability for the product/platform/system. The CTO's boss is very likely the CEO, so that executive will be responsible for approving the decision or course. Note that it is not absolutely necessary that the R's boss be the A in any given decision. It is entirely possible that the R will be performing actions on behalf of someone for whom he or she does not work. In this case, however, assuming that the CTO works for the CEO, there is very little chance that the CTO would actually have someone other than the CEO approve his or her scalability vision or scalability plan.

The CTO's peers are consultants to the scalability vision—the people who rely on the CTO for either the availability of the product or the back-office systems that run the company. These people need to be consulted because the systems that the CTO creates and runs are the lifeblood of the business units and the heart of the back-office systems that the CFO needs to do his or her job.

We have indicated that the CTO's organizations (Architecture group, Engineering team, Operations team, Infrastructure team, and QA) are all supporters of the vision, but one or more of them could also be consultants to the solution. The less technical the CTO, the more that executive will need to rely upon his or her teams to develop the vision for scalability. In Table 2.1, we have assumed that the CTO has the greatest technical experience on the team, which is obviously not always the case. The CTO may also want to bring in outside help in determining the scalability vision or plan. This outside help may take the form of a retained advisory services firm or potentially a technology advisory and governance board that provides the same governance and oversight for the technology team that a board of directors provides at a corporate level.

Finally, we have indicated that the board of directors needs to be informed (I) of the scalability vision. This might be a footnote in a board meeting or a discussion centered on what is possible with the current platform and how the company will need to invest to meet the scalability objectives for the coming years.

The remainder of the matrix has been partially filled out. Important points with respect to the matrix are that we have split up the tasks/initiatives to avoid overlaps in the R category. For instance, the responsibility for infrastructure tasks has been

split from the responsibility for software development or architecture and design tasks. This allows for clear responsibility in line with our “one back to pat and one throat to choke” philosophy. In so doing, however, the organization might tend to move toward designing in a silo or vacuum, which is counter to what you would like to have in the long term. Should you structure your organization in a similar fashion, it is very important that you implement processes requiring teams to design together to create the best possible solution. Matrix-organized teams do not suffer from some of the silo mentality that exists within teams built in silos around functions or organizational responsibility, but they can still benefit from RASCI. You should still have a single responsible organization, but you want to ensure that collaboration happens. RASCI helps enforce that philosophy through the use of the C attribute.

You should spend time working through the rest of the matrix in Table 2.1 until you feel comfortable with the RASCI model. It is a very effective tool in clearly defining roles and responsibilities and can help eliminate duplicated work, unfortunate morale-deflating fights, and missed work assignments.

One final note on roles and responsibilities: No team should ever cite roles and responsibilities as a barrier to getting things done. The primary responsibility of any person in the company is to create value for customers and stakeholders. There will certainly be times when people must perform roles and execute tasks that are outside of their defined responsibility. To the degree that such efforts help the company achieve its mission, personnel can and should be willing to step outside their boundaries to get the work done. The important message is that when those situations occur, they should work with leadership to identify where there are consistent gaps, and leaders should commit to getting those corrected for the future.

Conclusion

Providing role clarity is the responsibility of leaders and managers. Individuals as well as organizations need role clarity. This chapter provided some examples of how roles might be clearly defined to help the organization achieve its mission of attaining higher availability. These examples illustrate only a few of the many structures that might be created regarding individuals and organizations and their roles. The real solution for your organization might vary significantly from these structures, as the roles should be developed consistent with company culture and need. As you create role clarity, try to stay away from overlapping responsibilities, as these can create wasted effort and value-destroying conflicts. While we use role clarity as a way to ensure scalability within a company, it can be applied in many other aspects of your business as well.

We also introduced a tool called RASCI to help define roles and responsibilities within the organization. Feel free to use RASCI for your own organizational roles and for roles within initiatives. The use of RASCI can help eliminate duplicated work and make your organization more effective, efficient, and scalable.

Key Points

- Role clarity is critical for scale initiatives to be successful.
- Overlapping responsibility creates wasted effort and value-destroying conflicts.
- Areas lacking responsibility create vacuums of activity and failed scale initiatives.
- The CEO is the chief scalability officer of the company.
- The CTO/CIO is the chief technical scale officer of the company.
- RASCI is a tool that can help eliminate overlaps in responsibility and create clear role definition. RASCI is developed in a matrix in which
 - R stands for the person *responsible* for deciding what to do and running the initiative.
 - A is *accountable* for the initiative and the results or the *approver* in a decision-making process.
 - S stands for *supportive*, referring to anyone providing services to accomplish the initiative.
 - C stands for those who should be *consulted* before making a decision and regarding the results of the initiative.
 - I stands for those who should be *informed* of both the decision and the results.

Chapter 3

Designing Organizations

Management of many is the same as management of few. It is a matter of organization.

—Sun Tzu

This chapter discusses the importance of organizational structure and the ways in which it impacts a product's scalability. We will discuss two key attributes of organizations: size and structure. These two attributes go a long way toward describing how an organization works and where an organization will likely have problems as a group grows or shrinks.

Organizational Influences That Affect Scalability

Some of the most important factors that organizational structure can affect are communication, efficiency, standards, quality, and ownership. Let's take each factor and examine how an organization structure can influence it as well as why that factor is important to scalability. In this way, we can establish a causal relationship between the organization and scalability.

Communication and coordination are essential for any task requiring more than one person. A failure to communicate architectural designs, effects and causes of outages, sources of customer complaints, or reasons for and expected value from changes being promoted to production can all be disastrous. Imagine a team of 50 people with no demarcation of responsibilities or structural hierarchy. The chance that teammates know what everyone is working on is remote. This lack of knowledge is fine until you need to coordinate changes. Without structure, who do you ask for help? How do you coordinate your changes or know with whom you may be in conflict in making a change? These breakdowns in smooth communication, on most days, may cause only minor disruptions, such as having to spam all 50 people to get a question answered. But engineers will, over time, grow wary of the spam and start to ignore most messages. Ignoring messages will, in turn, mean that questions don't

get answered or—even worse—that critical functionality won’t work once merged into the code base. Was that failure the engineer’s fault for not being a superstar, or was it the organizational structure’s fault for making it impossible to communicate and collaborate clearly and effectively?

Organizational efficiency increases when an organizational structure streamlines workflow and decreases when projects get mired down in unnecessary organizational hierarchy where significant increases in communication are necessary to perform work. In the Agile software development methodology, product owners are often seated alongside engineers to ensure that product questions get answered immediately and efficiently. If the engineer arrives at a point in the development of a feature that requires clarification to continue, she has two choices. The engineer could guess which way to proceed or she could ask the product owner and wait for the answer. If she asks the product owner, she must wait for a response. Time waiting may be spent on another project or potentially on non-value-added tasks (like playing games). Substantial waiting time could even cause the team to miss its commitments and unnecessarily carry stories into future sprints, thereby delaying or diminishing the potential return on investment.

Having the product owner’s desk beside the engineer’s desk increases efficiency by getting those questions answered quickly. The alternative to colocation—that is, degraded efficiency vis-à-vis idle time—is a double-edged sword. First, the cost to create some value increases. Second, as your resource pool becomes fully utilized, the company starts favoring short-term customer-facing features at the expense of longer-term scalability projects. Quarterly goals may be met in the short term, but technical debt piles up for a lack of resources and ultimately forces the company, through downtime, to stall new product features.

Organizational efficiency is also driven by the adoption of standards. An organization that does not foster the creation, distribution, and adoption of standards in coding, documentation, specifications, and deployment is sure to suffer from decreased efficiency, reduced quality, and increased risk associated with significant production issues. To see how this behavior evolves, consider an organization that is a complete matrix, where very few engineers (possibly only one) reside on each team along with product managers, project managers, and business owners. Without the adoption of common standards, it would be very simple for teams to significantly diverge in best practices. Standards such as commenting code and the discipline to do so might slip with some teams as they favor greater throughput, but at the expense of future maintainability. To avoid this problem, great organizations help engineers understand the value of established guidelines, principles, and collective norms.

Here’s another example: Imagine your organization has an architectural principle requiring any service to run independently on multiple servers. A team that disregards this standard will both release a solution that will not scale horizontally and have intolerably low levels of availability. Think that doesn’t happen? Think again—we see

teams make this mistake all the time. The argument most commonly cited in support of this deviation from standards is that the service isn't critical to the functioning of the product. Of course, if that's the case, then why waste time building it in the first place? In this scenario, the team has inefficiently used engineering resources.

As described earlier, an organization that does not foster adherence to norms and standards in essence condones lowering the quality of the product being developed. A brief example of this would be an organization that has a solid unit test framework and process in place but is structured, through either size or team composition, so that it does not foster the acceptance and substantiation of this unit test framework. Perhaps one team might find it too tempting to disregard the parent organization's request to build unit tests for all features and forgo the exercise. This decision is very likely to lead to poorer-quality code and, therefore, to an increased number of major and minor defects. In turn, the higher defect rate might lead to downtime for the application or cause other availability-related issues. The resulting increase in bugs and production problems will take engineering resources away from coding new features or scalability projects, such as sharding the database. All too often, when resources become scarce in this way, the tradeoff of postponing a short-term customer feature in favor of a long-term scalability project becomes much more difficult.

Longhorn

The Microsoft operating system code-named Longhorn, which publicly became known as Vista, serves as an example of the failure of standards and quality in an organization. Ultimately, Vista became a successful product launch that achieved the ranking of being the second most widely adopted operating system, but not without significant pain for both Microsoft and its customers. The time period between the release of XP, the previous desktop operating system, and Vista marked the longest in the company's history between product launches.

In a front-page article on September 23, 2005, in *The Wall Street Journal*, Jim Allchin, Microsoft's co-president, admitted to telling Bill Gates, "It's not going to work," referring to Longhorn. Allchin described the development as "crashing to the ground" due to the haphazard methods by which features were introduced and integrated.¹

One factor that helped revive the doomed product was the decision to enlist the help of senior executive Amitabh Srivastava. Srivastava had a team of architects map out the operating system and established a development process that enforced high levels of code quality across the organization. Although this move drew great criticism from some of the individual developers, it resulted in the recovery of a failed product.

1. Robert A Guth. "Battling Google, Microsoft Changes How It Builds Software." *Wall Street Journal*, September 23, 2005. <http://online.wsj.com>.

Ownership also affects the scalability and availability of a product. When many people work on the same code and there is no explicit or implicit ownership of elements of the code base, no one feels as if he or she owns the code. When this occurs, no one takes the extra steps to ensure others are following standards, building the requested functionality, or maintaining the high quality desired in the product. Thus, we see the aforementioned problems with scalability of the application that stem from issues such as less efficient use of the engineering resources, more production issues, and poor communication.

Now that we have a clear basis for caring about the organization from a scalability perspective, it is time to understand the basic determinants of all organizations—size and structure.

Team Size

Consider a team of two people: The two know each other's quirks, they always know what each other is working on, and they never forget to communicate with each other. Sounds perfect, right? Now consider that they may not have enough engineering effort to tackle big projects, like scalability projects of splitting databases, in a timely manner; they do not have the flexibility to transfer to another team because each one probably has knowledge that no one else does; and they probably have their own coding standards that are not common among other two-person teams. Obviously, both small teams and large teams have their pros and cons. The key is to balance team size to get the optimal result for your organization.

An important point is that we are looking for the optimal team size for *your* organization. As this implies, there is not a single magic number that is best for all teams. Many factors should be considered when determining the optimal size for your teams, and the sizes may vary even from team to team within the same organization. If forced to give a direct answer to how many members should optimally be included in a team, we would provide a range and hope that suffices for a specific-enough answer. Although there are always exceptions even to this broad range of choices, our low boundary for team size is 6 and our upper boundary is 15. What we mean by *low boundary* is that if you have fewer than 6 engineers, there is probably no point in dividing them into separate teams. For the *upper boundary*, if there are more than 15 people on a single team, the size starts to hinder managers' ability to actively manage and communication between team members starts to falter. Having given this range, we ask that you recognize there are always exceptions to these guidelines; more importantly, consider the following factors as aligned with your organization, people, and goals. You might recall from Chapter 1 that Amazon calls this upper boundary the "Two-Pizza Rule": Never have a team larger than can be fed by two pizzas.

The first factor to consider when determining team size is the experience level of the managers. Managerial responsibilities will be discussed later in this chapter as a factor by itself, but for our purposes now we will assume that managers have a base level of responsibility, which includes the three following items: ensuring engineers are productive on value-creating projects, either self-directed or by edict of management; ensuring that administrative tasks, such as allocating compensation or passing along human resources information, are handled; and ensuring that managers stay current on the projects and problems they are running and can pass status information on the same along to upper management.

A junior manager who has just risen from the engineering ranks may find that, even with a small team of six engineers, the administrative and project management tasks consume her entire day. She may have time for little else because these tasks are new to her and they require a significant amount of time and concentration compared with her more senior counterparts. New tasks typically take longer and require more intense concentration than tasks that have been performed over and over again. A person's experience, therefore, is a key factor in determining the optimal size for a given team.

Tenure of the team is a second factor to consider. Long-tenured and highly experienced teams require less overhead from management and less communication internally to perform their responsibilities. Both experience (time) at the company and experience in engineering are important. Long-tenured employees generally require lower administrative overhead (e.g., signing up for benefits, getting incorrect paychecks straightened out, finding help with certain tasks). Likewise, experienced engineers need less help understanding specifications, designs, standards, frameworks, or technical problems.

Of course, every individual is different and the seniority of the overall team must be considered. If a team has a well-balanced group of senior, midlevel, and junior engineers, they can probably operate effectively in a moderate-sized team. By comparison, a team of all senior engineers, say on an infrastructure project, might be able to accommodate twice as many individuals because they should require much less communication items and be much less distracted with mundane engineering tasks. You should consider all of these issues when deciding on the optimal team size because doing so will provide a good indicator of how large the team can be and remain effective without causing disruption due to the overhead overwhelming the productivity.

As mentioned earlier, each company has different expectations of the tasks that a manager should be responsible for completing. We decided that a base level of managerial responsibilities includes ensuring the following:

- Engineers are productive on value-adding projects.
- Administrative tasks take place.
- Managers are current on the status of projects and problems.

Obviously, many more managerial responsibilities could be assigned to the managers, including one-on-one weekly meetings with engineers, coding of features by managers themselves, reviewing specifications, project management, reviewing designs, coordinating or conducting code reviews, establishing and ensuring adherence to standards, mentoring, praising, and performance reviews. The more of these tasks that must be handled by the individual managers, the smaller the team should be, so as to ensure the managers can accomplish all the assigned tasks. For example, if one-on-one meetings are required—and we believe they should be—an hour-long meeting with each engineer weekly for a team of 10 engineers will consume 25% of a 40-hour week. The numbers can be tweaked—shorter meetings, longer work weeks—but the point remains that just speaking to each engineer on a large team can take up a very large portion of a manager's time. Speaking frequently with team members is critical to be an effective manager and leader. Obviously, the number of tasks and the level of effort associated with these tasks should be considered as a major contributing factor when determining the optimal team sizes in your organization. An interesting and perhaps enlightening exercise for upper management is to survey the front-line managers and ask how much time they spend on each task for a week. As we indicated with the one-on-one meetings, it is surprisingly easy to fill up a manager's week with deceptively "quick" tasks.

The previous three factors—experience of management, tenure of team members, and managerial responsibilities—are all constraints. Each limits the size of the team with the intent to reduce overhead and maximize value creation time of the team.

Unlike the first three factors, our final factor—the needs of the business—works to increase team size. Business owners and product managers, in general, want to grow revenue, fend off competitors, and increase their customer bases. To do so, they often need more and increasingly complex functionality. One of the main problems with keeping team sizes small is that large projects require (depending on the product development life-cycle methodology employed) many more iterations or more time in development. The net result is the same: Projects take longer to get delivered to the customer. A second problem is that increasing the number of engineers on staff requires increasing the number of support personnel, including managers. Engineering managers may take offense at being called support personnel, but in reality that is what management should be—something that supports the teams in accomplishing their projects. The larger the teams, the fewer managers per engineer are required.

Obviously, for those familiar with the concepts outlined in *The Mythical Man-Month: Essays on Software Engineering* by Frederick P. Brooks, Jr., there is a limit to the amount a project can be subdivided to expedite delivery. Even with this consideration, the relationships remain clear: The larger the team size, the faster projects can be delivered and the larger the projects that can be undertaken.

Warning Signs

Let's turn our attention from factors affecting team size to signs that the team size is incorrect. Poor communication, lowered productivity, and poor morale are all potential indicators of a team that has grown too large. Poor communication could take many forms, including engineers missing meetings, unresponsiveness to emails, missed specification changes, or multiple people asking the same questions.

Decreased productivity can be another sign of a team that is too large. If the manager, architects, and senior engineers do not have enough time to spend with the junior engineers, these newest team members will not produce as many features as quickly. Without someone to mentor, guide, direct, and answer questions, the junior engineers will have to flounder longer than they normally would. The opposite scenario might also be the culprit: Senior engineers might be too busy answering questions from too many junior engineers to get their own work done, thereby lowering their productivity. Some signs of decreased productivity include missing release dates, lower function or story points (if measured), and pushback on feature assignment. Function points and story points are two different methods that attempt to standardize the measurement of a piece of functionality. Function points assume the user's perspective, whereas story points take on the engineer's perspective. Engineers by nature are typically overly optimistic in terms of what they think they can accomplish; if they are pushing back on an amount of work that they have done in the past, this reluctance might be a clear indicator that they feel their productivity slipping.

Both of the preceding problems—poor communication and decreased productivity due to lack of support—can lead to the third sign of a team being too large: poor morale. When a normally healthy and satisfied team starts demonstrating poor morale, this disgruntlement serves as a clear indicator that something is wrong. Although poor morale may have many causes, the size of the team should not be overlooked. Similar to how you approach debugging, look for what changed last. Did the size of the team grow recently? Poor morale can be demonstrated by a variety of behaviors, such as showing up late for work, spending more time in the game room, arguing more in meetings, and pushing back more than usual on executive-level decisions. The reason for this is straightforward: As an engineer, when you feel unsupported, left out of the communication loop, or unable to succeed in your tasks, that state weighs heavily on you. Most engineers love a challenge, even very tough ones that will take days just to understand the nature of the problem. When an engineer realizes that he cannot solve the problem, he falls into despair. This is especially true of junior engineers, so watch for these behaviors to occur first in the more junior team members.

Conversely, if the team is too small, indicators to look for include disgruntled business partners, micromanaging managers, and overworked team members. In

this situation, one of the first signs of trouble might be that business partners, such as product managers or business development, spend more time around the manager complaining that they need more products delivered. A team that is too small is just unable to deliver sizable features quickly. Alternatively, instead of complaining directly to the engineer or technology leadership, disgruntled business leaders might focus their energy in a more positive manner by supporting budget requests for more engineers to be hired.

A trend toward micromanagement from a normally effective manager is a second worrisome sign. Perhaps the manager's team is too small and she's keeping busy by hovering over her team members, second-guessing their decisions, and asking for status updates about the request for a status update. If this is the case, it represents a perfect opportunity to assign the manager some other tasks that will serve the organization, help professionally develop the manager by expanding her focus, and give her team some relief from her constant presence. Ideas for special projects that might be assigned in this scenario include chairing a standards committee, leading an investigation of a new software tool for bug tracking, or establishing a cross-team mentoring program for engineers.

The third sign to look for when a team is too small is overworked team members. Most teams are extremely motivated by the products they are working on and believe in the mission of the company. They want to succeed and they want to do everything they can to help. This includes accepting too much work and trying to accomplish it in the expected time frame. If the team members start to leave increasingly later each evening or consistently work on weekends, you might want to investigate whether this particular team includes enough engineers. This type of overworking behavior is expected and even necessary for most startup companies, but working in this manner consistently month after month will eventually burn out the team, leading to attrition, poor morale, and poor quality. It is much better to take notice of the hours and days spent working on tasks and determine a corrective action early, as opposed to waking up to the problem when your most senior engineer walks into your office to resign.

Ignoring the symptoms summarized here can be disastrous. If they go unchecked, it's almost inevitable that the company will experience unwanted attrition, slowing its ability to scale and deliver the product.

Growing or Splitting Teams

For the teams that are too small, adding engineers, although not necessarily easy, is straightforward. The much more difficult task is to split a team when it has become too large. Splitting a team incorrectly can have dire consequences—for example, confusion over code ownership, deterioration in communication, and increased

stress from working for a new manager. Every team and every organization is different, so there is no perfect, standard, one-size-fits-all way of splitting teams. Instead, some factors must be taken into account when undergoing this organizational surgery to minimize the impact and quickly restore the team members to a productive and engaged existence.

Some of the things that you should think about when considering subdividing a team include how to split the code base, who will be the new manager, what level of involvement will individual team members have, and how the relationships with the business partners will change.

The first item to concentrate on is based on the code or the work. As we will discuss in much more detail in Part III, “Architecting Scalable Solutions,” this might be a great opportunity to split the team as well as the code base into *failure domains*—that is, domains that limit the impact of failures by isolating services from one another.

The code that used to be owned by and assigned to a single team needs to be split between two or more teams. In the case of an engineering team, this division usually revolves around the code. The old team perhaps owned all the services around the administrative part of the application, such as account creation, login, billing, and reporting. Again, there is no standard way of doing this, but one possible solution is to subdivide the services into two or more groups: one group handling account creation and login, the other group handling billing and reporting services. As you get deeper into the code, you will likely hit base classes that require assignment to one team or the other. In these cases, we like to assign general ownership to one team or—even better—to one engineer; alerts can then be set up through the source code repository that inform the other team if anything changes in that particular file or class, keeping everyone aware of changes in their sections of the code.

The next item to consider is the identity of the new manager. This is an opportunity to hire someone new from the outside or to promote someone internally into the position. Each option has both pros and cons. An external hire brings new ideas and experiences, whereas an internal hire provides a manager who is familiar with all the team members as well as the processes. Because of the various advantages and disadvantages associated with each option, this is a decision you do not want to make lightly and might want to ponder for a long time. Making a well-thought-out decision is absolutely the correct thing to do, but taking too much time to identify a manager can cause just as many problems. The stress of the unknown can be dampening to employee morale and cause unrest. Make a timely decision; if that involves bringing in external candidates, do so as openly and quickly as possible. Dragging out the selection process and wavering between an internal candidate and an external candidate simply increases the stress on the team.

The last of the big three items to consider when splitting a team is how the relationship with the business will be affected. If a one-to-one relationship exists

between the engineering team, quality assurance, product management team, and business team, it will obviously change when a team is split. A discussion with all the affected leaders should take place before a decision is reached on splitting the team. Perhaps all the counterpart teams will split simultaneously, or perhaps individuals will be reassigned to interact more directly along team lines. There are many possibilities for restructuring, but the most important consideration is that an open discussion should take place beyond the engineering and technology teams.

So far, we have covered the warning signs associated with teams that are too large and too small. We have also addressed the factors to consider when splitting teams. One of the major lessons that should be gleaned from this section is that the team size and changes to it can have tremendous impacts on every aspect of the organization, from morale to productivity. In turn, it is critical to recognize team size's importance as a major determining factor of how effective the organization is in relation to scalability of the application.

Checklist

Optimal team size checklist:

1. Determine the experience level of your managers.
2. Calculate each engineer's tenure at the company.
3. Look up or ask each engineer how long he or she has been in the industry.
4. Estimate the total effort for managerial responsibilities.
 - a. Survey your managers to determine how much time they spend on tasks.
 - b. Make a list of the core managerial responsibilities that you expect managers to accomplish.
5. Look for signs of disgruntled business partners and managers who are bored to indicate teams that are too small.
6. Look for losses in productivity, poor communication, and degrading morale to indicate teams that are too large.

Splitting a team checklist:

1. Determine how to separate the code base:
 - a. Split by services.
 - b. Divide base classes and services as evenly as possible with only one owner.
 - c. Set up alerts in your repository to ensure everyone knows when items are being modified.

2. Determine who will be the new manager.
 - a. Consider internal versus external candidates.
 - b. Set an aggressive timeline for making the decision and stick to it.
3. Analyze your team's interactions with other teams or departments.
 - a. Discuss the planned split with other department heads.
 - b. Coordinate your team's split with other teams to ensure a smoother transition.
 - c. Use joint announcements for all departments to help explain all of the changes simultaneously.

Organizational Structure

The organizational structure refers to the actual layout or how teams relate to each other within an organization. This includes the separation of employees into departments, divisions, and teams as well as the management hierarchy that is used for command and control of the forces. Although there are as many different structures as there are companies, two basic structures have been in use for years—functional and matrix—and a new structure has recently come into favor—Agile. By understanding the pros and cons of the two time-honored structures as well as the new kid on the block, you will be able to choose one or the other. Alternatively, and perhaps as the more likely scenario, you can create a hybrid that best meets the needs of your company. This section covers the basic definition of each structure, summarizes the benefits and drawbacks of each, and offers some ideas on when to use each one. The most important lesson to be drawn here, however, is how to choose parts of one versus the other structure, and how to plan an evolution from one structure to another as your teams mature.

Functional Organization

The functional organizational structure is the original structure upon which armies and industries were based. This structure, as seen in Figure 3.1, separates departments and divisions by their primary purpose or function. This strategy was often called a silo approach because each group of people was separated from other groups just as grain or corn would be separated into silos based on the type or grade of crop. In a technology organization, a functional structure results in the creation of separate departments to house engineering, quality assurance, operations, project management, and so forth. Along with this, there exists a management hierarchy

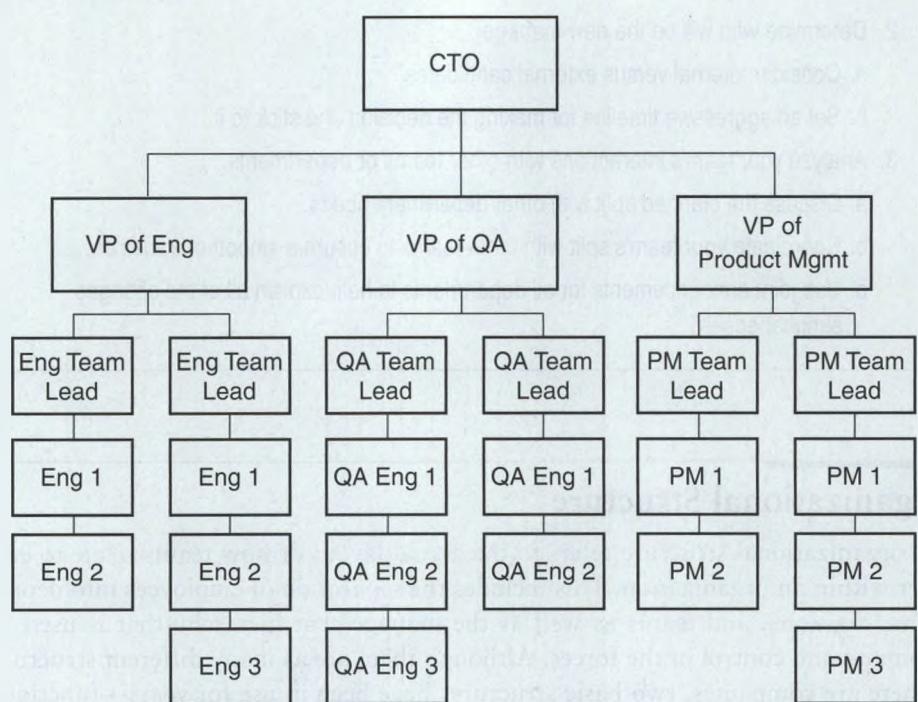


Figure 3.1 Functional Organization Chart

within each department. Each group has a department head, such as the VP of engineering, and a structure within each department that is homogeneous in terms of responsibilities. Reporting to the VP of engineering are other engineering managers such as engineering directors, and reporting to them are engineering senior managers and then engineering managers. This hierarchy is consistent in that engineering managers report to other engineering managers and quality assurance managers report to other quality assurance managers.

The functional or silo organizational structure offers numerous benefits. Managers almost always rise through the ranks; thus, even if they are not good individual performers, they at least know what is entailed in performing the job. Unless there have been major changes in the field over the years, there is very little need to spend time explaining to bosses the more arcane or technical aspects of the job because they are well versed in it. Team members are also consistent in their expertise—that is, engineers work alongside engineers. With this structure, peers who are usually located in the next cube can answer questions related to the technical aspects of the job quickly. The entire structure is built along lines of specificity.

To use an exercise analogy, the functional organizational structure is like a golfer practicing on the driving range. The golfer wants to get better and perform well at

golf and therefore surrounds himself with other golfers, and perhaps even a golf instructor, and practices the game of golf, all very specific to his goal. Keep this analogy in mind because we will use it to compare and contrast the functional organizational structure with the matrix structure.

Other benefits of the functional organizational structure, besides the homogeneity and commonality of management and peers, include simplicity of responsibilities, ease of task assignment, and greater adherence to standards. Because the organizational structure is extremely clear, almost anyone—even the newest members—can quickly grasp who is in charge of which team or phase of a project. This simplicity also allows for very easy assignment of tasks. In a waterfall software development methodology, the development phase is clearly the responsibility of the engineering team in a functional organization. Because all software engineers report up to a single head of engineering and all quality assurance engineers report to a single quality assurance head, standards can be established, decreed, agreed upon, and enforced fairly easily. All of these factors explain why the functional organization has for so long been a standard in both the military and industry.

The problems with a functional or silo organization include the lack of a single project owner and poor cross-functional communication. Projects almost never reside strictly within the purview of a single functional team. Most custom software development projects, especially services delivered via the Internet such as Software as a Service (SaaS) offerings, always require tasks to be accomplished by individuals with different skill sets. Even a simple feature request must have a specification drafted by the product owner, design and coding performed by the engineers, testing performed by the quality assurance team, and deployment by the operations engineers. Responsibility for all aspects of the project does not reside with any one person in the management hierarchy until you reach the head of technology, who has responsibility over the product managers, engineering, quality assurance, and operations staffs. This shifting of responsibility can be even further exacerbated when product management doesn't report through the CTO. Obviously, having the CTO or VP of technology be the lowest-level person responsible for the overall success of the project can be a significant drawback. With this structure, when problems arise in the projects, it is not uncommon for each functional owner to place the blame for delays or cost overruns on other departments.

As simple as the functional organization is to understand, communication can prove surprisingly difficult across departments. As an example, suppose a software engineer wants to communicate to a quality assurance engineer about a specific test that must be performed to check for the proper functionality. The software engineer may spend precious time tracking up and down the quality assurance management hierarchy looking for the manager who is assigning the testing of this feature and then requesting the identity of the person to whom the testing work will be assigned

so that the information can be passed along. More likely, the engineer will rely on established processes, which attempt to facilitate the passing along of such information through design and specification documents. As you can imagine, writing a line in a 20-page specification about testing leads to much more burdensome communication than a one-on-one conversation between the development engineer and the testing engineer.

Another challenge with the functional organizational structure is the conflict that arises between teams. When these organizations are tasked with delivering and supporting products or services that require cross-functional collaboration, conflict between teams inevitably arises. “This team didn’t deliver something on time” and “That other team delivered the wrong thing” are common refrains heard when functionally organized teams attempt to work together. Rarely is there an appreciation of each functional team’s challenges or contributions. For an engineer, both self-identity and social identity are tied in part to being seen as belonging to the “tribe” of engineers. Engineers want to belong and be accepted by our peers. Others who are different (quality assurance, product management, or even technical operations) are often seen as outsiders, not trusted, and sometimes the target of open hostility. This conflict between groups of individuals who perceive one another as different is witnessed at one of the most extreme levels through discrimination.

One classic example of the ease with which individuals can turn against others seen as different is the “blue-eyed/brown-eyed” exercise. In 1968, after the assassination of Martin Luther King, Jr., a third-grade schoolteacher in Iowa conducted an exercise in which she divided her class into groups of blue-eyed and brown-eyed students. She made up “scientific” evidence showing that eye color determined intelligence. The teacher told one group that they were superior, gave them extra recess, and had them sit in the front of the room; conversely, she did not allow the other group to drink from the water fountain. The teacher observed that the children deemed “superior” became arrogant, bossy, and unpleasant to their “inferior” classmates. She then reversed her explanation and told the other group that they were really superior. Elliott reported that this new group did taunt the other group similarly to how they were persecuted but were much less nasty in doing so.

Conflict

Practitioner experience and scholarly research agree that there are good forms of conflict and bad forms of conflict. The good conflict—referred to as cognitive conflict—is the healthy debate that teams get into regarding what should be or why something should be done; it involves a wide range of perspectives and experiences. Bad conflict—referred to as affective conflict—is role based and often involves how to do something or who should be doing

something. Of course, not all discussions over roles are "bad"; rather, lingering role-based discussions that are perceived as political or territorial can be unhealthy for an organization if not handled properly.

Good or cognitive conflict helps teams open up the range of possibilities for action. Diverse perspectives and experiences work together to attack a problem or an opportunity from multiple angles. Brainstorming sessions and properly run postmortems are examples of controlled cognitive conflict intended to generate a superior set of alternatives and actions. Team norms have been shown to have a positive effect on developing cognitive conflict; a culture of acceptance and respect for diverse opinions is more likely to generate more alternatives. Emotionally and socially intelligent leaders also may help create positive cognitive conflict within teams. Unfortunately, if cognitive conflict is left unresolved, it can escalate to affective (bad) conflict.

Bad or affective conflict results in physical and organizational trauma. Physically, it can leave us drained, because the sympathetic nervous system (the same system involved in the fight-or-flight syndrome, which is kicked off by the hypothalamus) releases stress hormones such as cortisol, epinephrine, and norepinephrine. As a result, blood pressure and heart rate increase. Over time, constant affective conflict leaves us feeling exhausted. Organizationally, teams experiencing this type of conflict fight over ownership and approaches. Organizations become fractured, and scholarly research shows that the result is a limiting of our options from a tactical and strategic perspective. The fighting closes our minds to options, meaning our results may potentially be suboptimal.

Why is this important? By understanding the sources and results of conflict, we as leaders can drive our teams to have healthy debates and can quickly end value-destroying affective conflict. Our job is to create a healthy environment suitable to the maximization of shareholder value. By creating an open, caring, and respectful culture, we can both maximize cognitive conflict and minimize affective conflict. By setting clear roles and responsibilities, we can limit the sources of affective conflict. Moreover, by hiring a diverse group of people with complementary skills and perspectives, we can minimize groupthink, maximize strategic options, and encourage our organizations to grow quickly.

Brown-Eyed/Blue-Eyed

On April 4, 1968, Martin Luther King, Jr., was assassinated in Memphis, Tennessee, only two days after delivering his "I've Been to the Mountaintop" speech. The next morning in rural Riceville, Iowa, Jane Elliott, a third-grade schoolteacher, asked her students what they knew about black people. She then asked the children if they would like to feel how a person of color was treated in America at the time. Elliott's idea was an exercise based on eye color instead of skin color.

She first designated the brown-eyed children as the “superior” group. Elliott reported that at first there was resistance to this assignment; to counter it, she lied to the children, telling them that melanin was responsible for making children brown-eyed and was linked to higher intelligence and learning ability. Following this, the children deemed “superior” became arrogant, bossy, and unpleasant to their “inferior” classmates. Elliott gave the brown-eyed children extra privileges, such as second helpings at lunch, access to the new jungle gym, and five extra minutes at recess. The brown-eyed children sat in the front of the classroom. Elliott would not allow brown-eyed and blue-eyed children to drink from the same water fountain.

The brown-eyed children’s performance on academic tasks improved, while the blue-eyed children became timid and subservient. One brown-eyed student asked Elliott how she could be a teacher with blue eyes. Another brown-eyed student responded, “If she didn’t have blue eyes, she’d be the principal or the superintendent.”

The next Monday, Elliott reversed the exercise, making the blue-eyed children superior. While the blue-eyed children did taunt the brown-eyed children in similar ways, Elliott reports that their behavior was much less intense. On Wednesday of the following week, Elliott halted the exercise and asked the children to write down what they had learned. These reflections were printed in the local newspaper and then subsequently picked up by the Associated Press. Elliott received quite a bit of notoriety, including an appearance on *The Tonight Show Starring Johnny Carson*. Reactions to her experiment were mixed, ranging from outrage at her treatment of children to an invitation to speak at the White House Conference on Children and Youth.²

The benefits of the functional organization include commonality of managers and peers, simplicity of responsibility, and adherence to standards. The drawbacks include no single project owner and poor communications. Given these pros and cons, the scenarios in which you would want to consider a functional organizational structure are ones in which the advantages of specificity outweigh the problems of overall coordination and ownership. For example, organizations that follow waterfall processes can often benefit from functionally aligned organizations. The functional alignment of the organization neatly coincides with the phase containment inherent to waterfall methods.

Matrix Organization

In the 1970s, organizational behaviorists and managers began rethinking organizational structure. As discussed previously, although there are certain undeniable

2. Stephen G. Bloom. “Lesson of a Lifetime.” *Smithsonian Magazine*, September 2005. <http://www.smithsonianmag.com/history/lesson-of-a-lifetime-72754306/?no-ist=&page=1>. Accessed June 24, 2014.

benefits to the functional organization, it also has certain drawbacks. In an effort to overcome these disadvantages, companies and even military organizations began experimenting with different organizational structures. The second primary organizational structure that evolved from this work was the matrix structure.

The principal concept in a matrix organization is the two dimensions of the hierarchy. As opposed to a functional organization, in which each team has a single manager and thus each team member reports to a single boss, the matrix includes at least two dimensions of management structure, whereby each team member may have two or more bosses. Each of these two bosses may have different managerial responsibilities—for instance, one (perhaps the team leader) handles administrative tasks and reviews, whereas the other (perhaps the project manager) handles the assignment of tasks and project status. In Figure 3.2, the traditional functional organization is augmented with a project management team on the side.

The right side of the organization in Figure 3.2 looks very similar to a functional structure. The big difference appears on the left side, where the project management organization resides. Notice that the project managers within the Project Management Organization (PMO) are shaded with members of the other teams. Project Manager 1 is shaded light gray along with Engineer 1, Engineer 2, Quality Assurance Engineer 1, Quality Assurance Engineer 2, Product Manager 1, and Product Manager 2. This light gray group of individuals constitutes the project team that is working together in

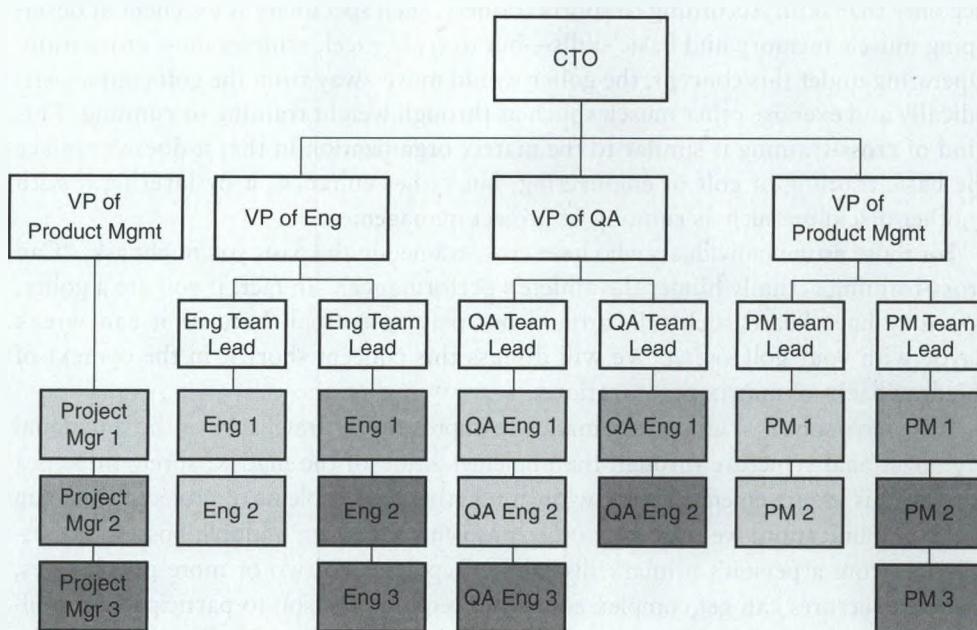


Figure 3.2 Matrix Organization Chart

a matrixed fashion. The light gray team project manager might have responsibility for the assignment of tasks and the timeline. In larger and more complex matrix organizations, many members of each team can belong to project teams.

Continuing with the project team responsible for implementing the new billing feature, we can start to realize the benefits of such a structure. The three primary problems with a functional organization are no project ownership, poor cross-team communication, and inherent affective conflict between organizations. In the matrix organization, the project team fixes all of these problems. We now have a first-level manager, Project Manager 1, who owns the billing project. This project team will likely meet weekly or more often, and will certainly have frequent email dialogues, which solves one of the problems facing the functional organization: communication. If the software engineer wants to communicate to the QA engineer that a particular test should be included in the test harness, doing so is as simple as sending an email or mentioning the need at the next team meeting. Thus this structure alleviates the need to trudge through layers of management in search of the right person.

We can now return to the golf analogy that we used in the discussion of the functional organization. Recall that we described a golfer who wants to get better and perform well at golf. To that end, he surrounds himself with other golfers, and perhaps even a golf instructor, and practices the game of golf—all activities very specific to his goal. This is analogous to the functional team where we want to perform a specific function very well, so we surround ourselves with others like us and practice only that skill. According to sports trainers, such specificity is excellent at developing muscle memory and basic skills—but to truly excel, athletes must cross-train. Operating under this concept, the golfer would move away from the golf course periodically and exercise other muscles such as through weight training or running. This kind of cross-training is similar to the matrix organization in that it doesn't replace the basic training of golf or engineering, but rather enhances it by layering it with another discipline such as running or project management.

For those astute individuals who have cross-trained in the past, you might ask, "Can cross-training actually hinder the athlete's performance?" In fact, if you are a golfer, you may have heard such talk around not playing softball because it can wreak havoc with your golf swing. We will discuss this concept shortly, in the context of the drawbacks of matrix organizations.

If we have solved or at least dramatically improved the drawbacks of the functional organizational structure through the implementation of the matrix, surely there is a cost for this improvement. In fact, while mitigating the problems of project ownership and communication, we introduce other problems involving multiple bosses and distraction from a person's primary discipline. Reporting to two or more people—yes, matrix structures can get complex enough to require a person to participate on multiple teams—invariably causes stressors because of differences in direction given by

each boss. The engineer trapped between her engineering manager telling her to code to a standard and her project manager insisting that she finish the project on time faces a dilemma fraught with stress and the prospect of someone not being pleased by her performance. Additionally, the project team requires overhead, as does any team, in the form of meetings and email communications. This overhead does not replace the team meetings that the engineer must attend for her engineering manager, but rather takes more time away from her primary responsibility of coding.

As you can see, while solving some problems, the matrix organizational structure introduces new ones. This really should not be too shocking because that is typically what happens—rarely are we able to solve a problem without triggering consequences of another variety. The next question, given the cons of both the matrix organization and the functional organization, is clear: “Is there a better way?” We believe the answer is “yes” and it comes in the form of an “Agile Organization.”

Agile Organization

Developing unique and custom SaaS solutions or enterprise software is a complex process requiring cross-functional collaboration between multiple skill sets. Within functionally organized teams delivering SaaS products, conflict and communication issues are inevitable. While the matrix organizational structure solved some of these issues by creating teams with diverse skill sets, it created other problems, such as individual contributors reporting to multiple managers who often have differing priorities. The unique requirements of SaaS, the advent of Agile development methodologies, and the drawbacks of both functional and matrixed organizations has led to the development of a new organizational structure that we call the Agile Organization.

In February 2001, 17 software developers, representing practitioners in various document-driven software development methodologies, met in Snowbird, Utah, to discuss a lightweight methodology.³ What emerged was the *Manifesto for Agile Software Development*, which was signed by all of the participants. This manifesto, contrary to some opinions, is not anti-methodology, but rather a set of 12 principles that attempt to restore credibility to the term *methodology* by outlining how building software should be intensely focused on satisfying the customer. One of the principles focused on how teams should organize: “The best architectures, requirements, and designs emerge from self-organizing teams.” This idea of an autonomous, self-organizing team opened people’s minds to the possibility of a new organizational structure that wasn’t role based, but rather focused on satisfying the customer.

3. Kent Beck et al. “Manifesto for Agile Software Development.” Agile Alliance, 2001. Retrieved June 11, 2014.

Centralized hosting of business applications is not something new. Indeed, it dates back to the 1960s, with time-sharing on mainframes. Fast-forward to the 1990s and the rapid expansion of the Internet, and we find entrepreneurs marketing themselves as application service providers (ASPs). These companies hosted and managed applications for other companies, with each customer getting its own instance of the application. The reputed value was reduced costs for the customers due to the ASP being extremely skilled at hosting and managing a particular application. By the early 2000s, another shift had come about—the emergence of Software as a Service. Supposedly this term first appeared in an article called “Strategic Backgrounder: Software as a Service,” internally published in February 2001 by the Software & Information Industry Association’s (SIIA) eBusiness Division.⁴ Like most things in the technology world, the definition of SaaS can be debated, but most agree that it includes a subscription-pricing model whereby customers pay based on their usage rather than a negotiated licensing fee, and that the architectures of these applications are usually multitenant, meaning that multiple customers use the same instance of the software.

With this shift toward providing a service rather than a piece of software, technologists began thinking about being service providers rather than software developers. Along with this evolving mindset came other ideas about the expected quality and reliability of delivery of these services. Traditionally, when we think of a service, we conjure up ideas of household services such as water, sanitation, and electricity. With such services, we have very high expectations of their quality and dependability. When we turn a faucet, we expect clear, potable drinking water to spew forth every time. When we flip a light switch, we expect electricity to flow, with very little variation in current at our disposal. Why shouldn’t we expect the same from software services? As customer expectations related to SaaS increased, technology companies began to react by attempting to offer more reliable services. Unfortunately, their traditional organizations kept getting in the way of meeting this standard. What resulted was even greater conflict between functional teams and slower delivery.

The last piece of the puzzle that allowed the Agile Organization to be conceptualized was the realization that the organizational structure of a technology team matters greatly to the quality, scalability, and reliability of the software. The authors of this book arrived at this conclusion only after several years of scalability engagements with clients. As technologists, when we started consulting, we were certain that our efforts would be focused on technology and architecture. After all, couldn’t every technology problem be solved through technology? Yet, in engagement after engagement, the conversation kept returning to organizational issues such as conflict between teams or individuals reporting to multiple managers and not understanding their priorities.

4. *Strategic Backgrounder: Software as a Service*. Washington, DC: Software & Information Industry Association, February 28, 2001. <http://www.slideshare.net/Shelly38/software-as-a-service-strategic-backgrounder>. Accessed April 21, 2015.

Eventually, we remembered that *people* develop technology and, therefore, *people* are important to the process. Thus began our journey to understanding that a truly scalable system requires the alignment of architecture, organization, and process. The culmination of this epiphany was the first version of *The Art of Scalability*, published in 2009. We certainly don't claim to have been the first to recognize the importance of organizations, nor do we mean to suggest that this book was the most influential in propagating this message. However, given that both the authors had previously held executive-level operating roles at high-tech companies, our lack of understanding of this key point serves as an indicator of what the technology community in general thought during the early to mid-2000s.

The result of these three factors was technology companies testing various permutations of organizational structures in an attempt to improve the quality and reliability of the software services they were offering. As with the functional and matrix organizations, many variations of this new organization structure are possible. For simplicity, we label any organization that is cross-functional and aligned to the architecture of the services that are provided as an Agile Organization. The Agile Organization, as shown in Figure 3.3, creates teams that are completely autonomous and self-contained. These teams own a service throughout the entire life cycle—from idea inception, to development, to support of the service in production. Directors or VPs of cross-functional, Agile teams have replaced the typical managerial roles such as VP of Engineering.

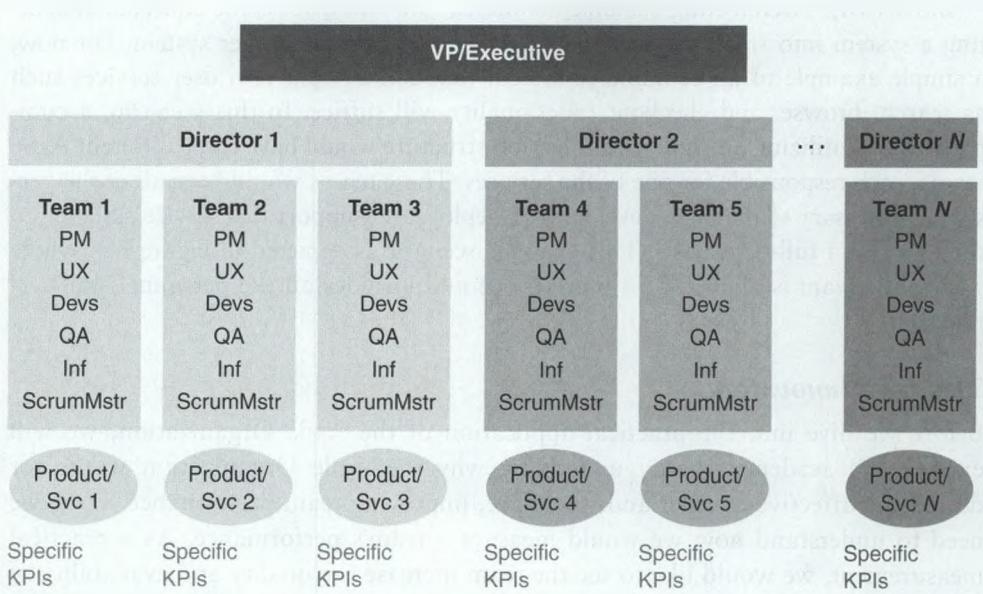


Figure 3.3 Agile Organization Chart

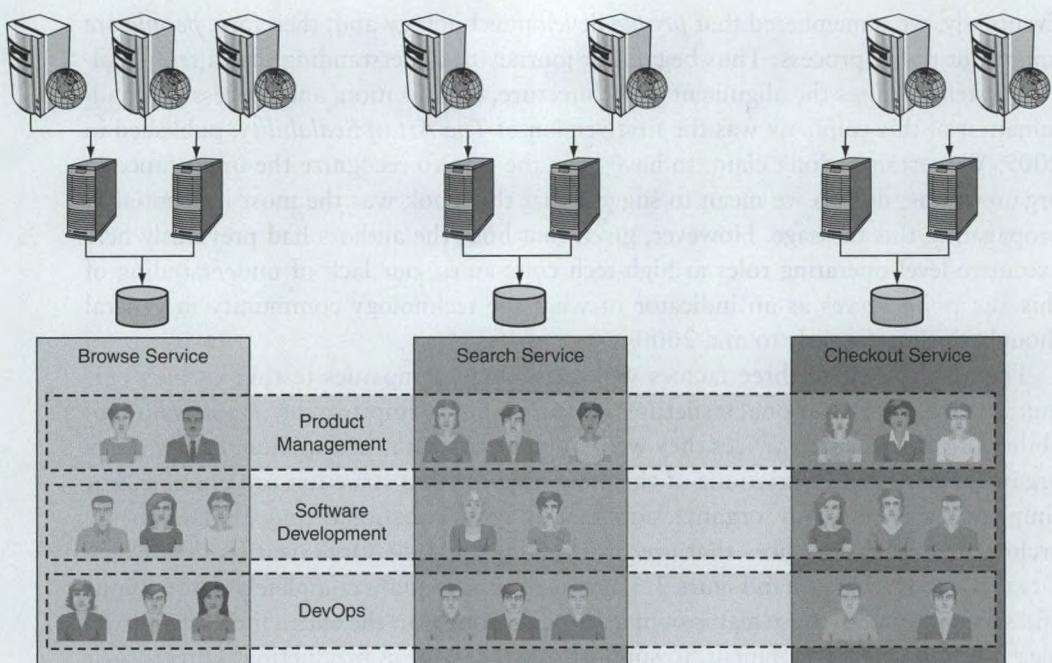


Figure 3.4 Agile Organization Aligned to Architecture

In Part III, “Architecting Scalable Solutions,” we will discuss the concepts of splitting a system into small services that together make up the larger system. For now, a simple example of an ecommerce system that can be split into user services such as search, browse, and checkout functionality will suffice. In this scenario, a company that is utilizing an Agile Organization structure would have three different Agile teams, each responsible for one of the services. These teams would have all of the personnel necessary to manage, develop, test, deploy, and support this service assigned to the team on a full-time basis. This example scenario is depicted in Figure 3.4, where each Agile team is aligned to a user service and includes all the personnel skill sets required.

Theory of Innovation

Before we dive into the practical application of the Agile Organization, we will explore the academic theory underlying why the Agile Organization works for decreasing affective conflict and, therefore, improving team performance. First, we need to understand how we would measure a team’s performance. As a practical measurement, we would like to see the team increase the quality and availability of the services that it provides. In academic research, we can use the term *innovation*

to represent the value-added output of a team. Innovation has been identified as a criterion that encompasses effective performance. The question that many of us have asked for years is this: “Which factors help teams increase their innovation?” Through extensive qualitative and quantitative primary research, triangulated with extant research, we now know some of the factors that drive innovation. In the discussion that follows, we will explore each of these factors in turn.

As discussed previously, conflict can be either productive or destructive, depending on the type. Cognitive conflict brings diverse perspectives and experiences together. Brainstorming sessions are often seen as examples of cognitive conflict, whereby teams attempt to gather a set of alternatives superior to what the team members could arrive at in isolation. It’s likely that each of us has participated in at least one brainstorming session that was incredibly productive. The session probably started with a leader setting the agenda, making sure everyone knew each other, establishing some ground rules around respect and time limits, and so on. What ensued might have been a 60- to 90-minute session in which people built upon each other’s ideas. Not everyone agreed, but ideas were exchanged in a respective manner such that roadblocks to solutions were raised and ways around them were discussed. This collaboration resulted in creative and innovative ideas that likely no one would have generated on their own. Everyone probably left the meeting feeling glad they had participated and that the time investment was well worth it. We also probably wished that all of our other meetings that week went as smoothly.

The opposite of cognitive conflict is affective conflict. This so-called bad or destructive conflict is role based and revolves around the questions of “who” or “how” a task should be done. Affective conflict places physical and emotional stress on team members. Whereas cognitive conflict can increase a team’s innovation, affective conflict decreases a team’s innovation.

Affective and cognitive conflicts are not the only factors that influence innovation. If we think back to our recollected brainstorming session and mentally look around the meeting room, the participants probably represented diverse backgrounds. Perhaps one person was from engineering, while another individual came from product management. Some of the individuals were only a few years out of college, while others had been in the workforce for decades. This experiential diversity can increase both affective and cognitive conflict. Sometimes people with different backgrounds tend to butt heads because they approach problems or opportunities from such different perspectives. If this were always the case, we would staff teams with people of similar backgrounds—but alas, dynamics between people aren’t that simple. Diversity of experience also can promote diversity of thoughts, leading to ideas and solutions that go far beyond what a single person could ever achieve. Thus experiential diversity increases both affective and cognitive conflict. The key for us as leaders attempting to increase our teams’ innovation is to minimize how

experiential diversity impacts the affective components and to maximize how it impacts the cognitive aspects.

Another type of diversity that is important in terms of influencing innovation is network diversity, which is a measure to what extent individuals on a team have different personal or professional networks. Network diversity becomes important with regard to innovation because almost all projects run into roadblocks. Teams with diverse networks are better able to identify these potential roadblocks or problems early in the project. We have all probably been on a team where one individual, who came from a different background than the rest and still had friends from that past, was able to provide clear advice on potential problems. Perhaps you were working on an IT project to implement a new software package in a manufacturing plant. One of your team members who had been a summer intern on the manufacturing floor was able to give the team a heads-up that one of the manufacturing lines could be shut down to install the new software only on certain days of the week. This type of network diversity on a team brings greater knowledge to the project. When a team actually does encounter roadblocks, those teams with the most diverse networks are better prepared to find resources outside of their teams to circumvent those obstacles. Such resources might take the form of upper-level managers who can provide support or even additional QA engineers who can help test the software.

Another factor that is widely credited with increasing innovation is the team's sense of empowerment. If a team feels empowered to achieve a goal, it is much more likely to achieve it. An interesting counterexample can be seen in many military selection courses, where decreasing empowerment is intended to decrease a candidate's motivation to complete tasks. Decreasing a team's or individual's sense of empowerment to accomplish a goal tests one's mettle or fortitude. One technique for achieving this aim is to move the goal line.

Imagine yourself in one of the competitive military selection courses, such as officer candidate school, which selects and trains enlisted soldiers to become commissioned officers to lead soldiers on the battlefield. To get to this level, you must have first established yourself as a stellar enlisted soldier, achieving the highest marks on your reviews. You must have also competed in a variety of psychological, physical, and mental tests over the course of months or even years. You're mentally sharp and physically fit; you're confident that you can overcome any obstacle and accomplish any task put in front of you.

This morning you wake up before the sun comes up, don your physical training uniform, lace up your shoes, and head out for formation. On the agenda is a physical test of your running endurance. Being a pretty decent runner, you're thinking this test should be a breeze. The wrench that the instructors throw at you is not telling you where the finish line is located. You run out a couple of miles and turn around, heading back to the starting line. Most people would think that the finish line would

be where the start line was. However, upon arriving back to the start line, you turn around and head back out on a different path. After a few more miles, you again turn around and head back to the start line. Surely, this will be the finish line. Not so fast: As the instructors approach the start line, they turn and head back out on a yet another path.

It's at this point that the candidates begin to break and throw in the towel. Not knowing where the finish line or goal is, people become disheartened and start doubting their ability. The opposite of this is, of course, true as well: When individuals or teams believe they are empowered with the resources they need to accomplish something, their innovation increases. Returning to our officer candidate school example, if the soldiers believe they are empowered by being in the proper physical shape, outfitted in the proper running attire, and armed with a clear understanding of their goals, they are much more likely to achieve them.

There is one more factor that we need to understand in our model of innovation—a factor that deals directly with the organizational structure. This factor is organizational boundaries, which simply means individuals are on different teams. Between all teams are boundaries that separate them. Some boundaries are narrow, such as those between similar teams, while others are quite large, such as those found between very different teams (e.g., product managers and system administrators). In our model of innovation, organizational boundaries, across which collaboration must happen, increase affective conflict. Therefore, the more organizational boundaries that a team must cross to coordinate with others for the accomplishment of a goal, the less innovation that the team will demonstrate. We touched upon the reason for this outcome earlier. As engineers, our self-identity is tied in part to being seen as belonging to the “tribe” of engineers. We want to belong and be accepted by our peers. Others who are different (quality assurance, product management, or even technical operations personnel) are often seen as outsiders and people who should not be trusted. It has been hypothesized that survival strategies may constitute a *homo homini lupo*⁵ situation in which outsiders are distrusted as hostile competitors for scarce resources. Distrust toward outsiders forces individuals into rigid in-group discipline.⁶ This sort of emotional aloofness and distrust of outsiders has been observed in many groups.⁷

5. “Man is a wolf to [his fellow] man.”

6. Christian Welzel, Ronald Inglehart, and Hans-Dieter Klingemann. *The Theory of Human Development: A Cross-Cultural Analysis*. Irvine, CA: University of California—Irvine, Center for the Study of Democracy, 2002. <http://escholarship.org/uc/item/47j4m34g>. Accessed June 24, 2014.

7. Peter M. Gardner. “Symmetric Respect and Memorate Knowledge: The Structure and Ecology of Individualistic Culture.” *Southwestern Journal of Anthropology* 1966;22:389–415.

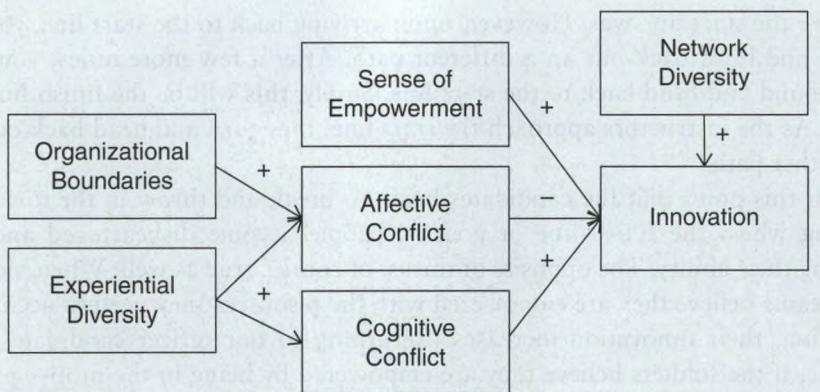


Figure 3.5 *Theory of Innovation Model*

This brings us to our final theoretical model for team innovation. Figure 3.5 depicts the complete model, in which the factors of network diversity, sense of empowerment, and cognitive conflict increase innovation. Affective conflict decreases innovation. Experiential diversity increases both affective and cognitive conflict, while organizational boundaries increase just the affective conflict. Now, armed with this fully detailed model, we can clearly articulate why the functional and matrix organizational structures decrease innovation whereas the Agile Organization increases it.

In a functional organization, individuals are organized by their skill or specialty. Almost all projects require coordination across teams. This is especially true of SaaS offerings, where the responsibilities of not only developing and testing the software but also hosting and supporting it fall on the company's technology team. With shrink-wrapped software or software that the customer installs and supports, part of this responsibility is shared with the customer. In today's more popular SaaS model, the entirety of this responsibility belongs to companies' technology teams. As shown in Figure 3.6, this causes affective conflict between teams. Organizational boundaries increase the affective conflict, resulting in a decrease in innovation.

In matrix organizations, where individuals have multiple managers, often with different priorities, we see the "moving goal line" problem. Attempting to please two masters often leads to unclear goals and teams with a reduced sense of empowerment.

Agile Organizations have neither of these problems. They break down the organizational boundaries that functional organizations struggle with, and they empower teams, eliminating the problem that matrix organizations face.

Pros of an Agile Organization The primary benefit of an Agile Organization is the increased innovation produced by the team. In a SaaS product offering, this increased

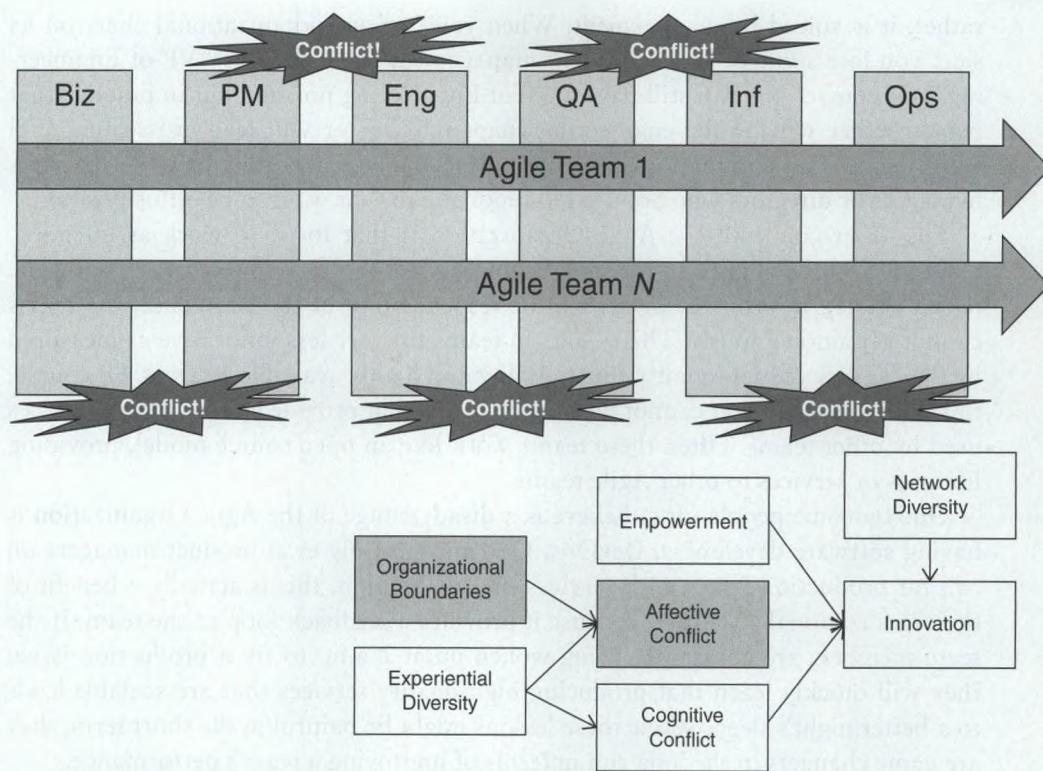


Figure 3.6 Functional Organization Increases Affective Conflict

innovation is often measured in terms of faster time to market with features, greater quality of the product, and higher availability. Such benefits are often realized by teams that embrace the Agile Organization structure. As expected, the increased innovation is driven by the improvement in factors such as conflict, empowerment, and organizational boundaries.

When teams align themselves according to services, are autonomous, and have cross-functional composition, the result is a significant decrease in affective conflict. The team members have shared goals and no longer need to argue about who is responsible or who should perform certain tasks. The team wins or loses together. Everyone on the team is responsible for ensuring the service they provide meets the business goals, which include high-quality, highly available functionality.

While the Agile Organization does a great job at improving a team's innovation, there are downsides to this organizational structure. We'll discuss these drawbacks next.

Cons of an Agile Organization The primary complaint that we hear about an Agile Organization doesn't come from engineers, product managers, or any team members;

rather, it is voiced by management. When you turn an organizational chart on its side, you lose some of the traditional management roles, such as “VP of Engineering.” Of course, you can still have a VP of Engineering position but in practice that person either will be the engineering chapter leader or will lead cross-functional Agile teams. This elimination of the upper-level management role sometimes bothers managers or directors who are used to reporting to—or want to be—this position.

The other con with the Agile Organization is that for it to work as intended, teams need to be aligned to the user-facing services in the architecture. When Agile teams overlap in terms of ownership or responsibility of the code base, the teams cannot act autonomously. This results in teams that are less innovative as measured by the delivery of high-quality functionality and highly available services. Of course, this does not mean you cannot have Agile teams that provide common core services used by other teams. Often these teams work like an open source model, providing libraries or services to other Agile teams.

One outcome people often believe is a disadvantage of the Agile Organization is having software developers, DevOps, QA, and possibly even product managers on call for production issues with services. In our opinion, this is actually a benefit of the organizational structure, because it provides a feedback loop to the team. If the team members are constantly being woken up at 2 a.m. to fix a production issue, they will quickly learn that producing high-quality services that are scalable leads to a better night’s sleep. While these lessons might be painful in the short term, they are game changers in the long run in terms of improving a team’s performance.

While no organization structure is perfect, we believe that the Agile Organization is an ideal choice for many companies struggling with poor service delivery, high amounts of conflict, unmotivated employees, and a general lack of innovation.

Spotify Organizational Structure

A real-world example of a service-oriented, cross-functional team can be found at Spotify. Its organizational structure is not functionally aligned, but rather consists of small Agile teams, which the company refers to as squads. These self-contained teams are organized around a deliverable or service that the business provides. The following descriptions are taken from Henrik Kniberg and Anders Ivarsson’s October 2012 paper, “Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds.”

A squad is similar to a Scrum team, and is designed to feel like a mini-startup. Members of a squad sit together, and they have all the skills and tools needed to design, develop, test, and release to production. They are a self-organizing team and decide their own way of working—some use Scrum sprints, some use Kanban, some use a mix of these approaches. Each squad has a long-term mission based on a service, which that team supports.

Squads have a dedicated product owner who prioritizes the work and takes both business value and technology aspects of the product into consideration. Squads also have an Agile coach who helps them identify impediments and encourages them to continuously improve their process. The squad has a long-term mission that everyone knows, with stories on the backlog related to the mission.

A tribe is a collection of squads that work in related areas. The tribe can be seen as the “incubator” for the squad mini-startups. Tribes enjoy a fair degree of freedom and autonomy. Each tribe has a tribe lead, who is responsible for providing the best possible habitat for the squads within that tribe. The squads in a tribe are all physically located in the same office, normally right next to each other, and the lounge areas nearby promote collaboration between the squads. Tribes include fewer than 100 people in total.

A chapter comprises a small group of people having similar skills and working within the same general competency area, within the same tribe. Each chapter meets regularly to discuss its area of expertise and specific challenges. The chapter lead serves as the line manager for the chapter members, with all the traditional responsibilities, such as developing people and setting salaries. However, the chapter lead is also part of a squad and is involved in the day-to-day work, which helps this individual stay in touch with reality.

A guild is a more organic and wide-reaching “community of interest”—a group of people who want to share knowledge, tools, code, and practices. Whereas chapters are always local to a tribe, a guild usually cuts across the whole organization. Some examples are the Web technology guild, the tester guild, and the Agile coach guild.

Conclusion

In this chapter, we highlighted the factors that an organizational structure can influence and showed how they are also key factors in application or Web services scalability. We established a link between the organizational structure and scalability to point out that, just like hiring the right people and getting them in the right roles, building a supporting organizational structure around them is important. We discussed the two determinants of an organization: team size and team structure.

In regard to the team size, size truly matters: A too-small team cannot accomplish enough; a too-large team has lower productivity and poorer morale. Four factors—management experience, team member tenure in the company and in the engineering field, managerial duties, and the needs of the business—must be taken into consideration when determining the optimal team size for your organization. A variety of warning signs should be monitored to determine if your teams are too large or too small. When teams are too large, poor communication, lowered productivity, and poor morale may emerge as symptoms. When teams are too small, disgruntled

business partners, micromanaging managers, and overworked team members may be apparent. Growing teams is relatively straightforward, but splitting up teams into smaller teams entails much more. When splitting teams, topics to be considered include how to split the code base, who will be the new manager, which level of involvement individual team members will have, and how the relationship with the business partners will change.

The three team structures discussed in this chapter were functional, matrix, and Agile Organization. The functional structure—the original organizational structure—essentially divides employees based on their primary function, such as engineering or quality assurance. The benefits of a functional structure include homogeneity of management and peers, simplicity of responsibilities, ease of task assignment, and greater adherence to standards. The drawbacks of the functional structure are the lack of a single project owner and poor cross-functional communication. The matrix structure starts out much like the functional structure but adds a second dimension, consisting of a new management structure. It normally includes project managers as the secondary dimension. The strengths of the matrix organization are its resolution of the project ownership and communication problems; its weaknesses include the presence of multiple bosses and distraction from a person's primary discipline. Finally, the Agile Organization improves teams' innovation as measured by time to market, quality of features, and availability of services.

Key Points

- Organizational structure can either hinder or help a team's ability to produce and support scalable applications.
- Team size and team structure are the two key attributes with regard to organizations.
- Teams that are too small do not provide enough capacity to accomplish the priorities of the business.
- Teams that are too large can cause a loss of productivity and degrade morale.
- The two traditional organizational structures are functional and matrix.
- Functional organizational structures provide benefits such as commonality of management and peers, simplicity of responsibilities, ease of task assignment, and greater adherence to standards.
- Matrix organizational structures provide benefits such as project ownership and improved cross-team communication.
- Agile Organization structures, especially those aligned to services and architecture, provide increased innovation as measured by faster time to market, higher-quality functionality, and higher availability of services.

Chapter 4

Leadership 101

A leader leads by example, not by force.

—Sun Tzu

Why does a book on scalability have a chapter on leadership? The answer is pretty simple: If you can't lead or fail to make the right leadership decisions, you won't scale your solutions in time to meet customer demand. In our experience as executives, consultants, and advisors, leadership failures are the most common reasons for a failure to scale. All too often companies get caught in the trap of focusing solely on product functionality, without realizing what is necessary to achieve appropriate uptime and response time under increasing user demands. Then, as the first really big rush of demand comes, the product begins to slow down and finally fails. Don't believe us? No story is more emblematic of this common type of failure than the story of Friendster.

Jonathan Abrams founded Friendster in 2002. He had previously founded a somewhat successful company in 1998 called Hotlinks. Abrams initially envisioned Friendster as a site focused on a new and better way to "date"—connecting friends of friends with one another. Having preceded Facebook in its inception, Friendster ultimately became one of the first successful examples of a non-work-oriented social networking site. Users had profile pages and could create links to other profile pages, invite other friends to join, create testimonials about each other, and so on. The product was so successful that it reached 1 million users within the first 4 months of launch—truly impressive growth!

The engineers at Friendster were insanely focused on an interesting problem within computer science that they called the "friend-graph" (the "f-graph," for short). View any person's page, and the f-graph would show how you were connected to that person with initially up to four degrees of separation, and then ultimately with unlimited degrees of separation. A user could visit Kevin Bacon's page and see exactly how many friend relationships separated him from the actor—six friends would be six degrees of separation. But the f-graph proved computationally

complex, especially when the four degrees of separation limitation was lifted, and response times started to increase significantly as the servers tried to calculate relationships with each view of a page. Soon, Friendster started to have response time and availability issues that impacted the willingness of users to use its product.¹

Friendster didn't correct its failures soon enough. When the company proved intent upon solving and delivering the business functionality of the f-graph rather than correcting the issues resulting from not designing a scalable solution, users flocked to alternative solutions. The result? In 2009, a Malaysian company acquired Friendster for \$26.4 million. Facebook—a company founded nearly a year later than Friendster—had an initial public offering and, when this second edition was being written, had a market capitalization exceeding \$170 billion.

Friendster ceded the battlefield to Facebook by not creating a compelling vision that pulled the organization to do the right thing for its customers in creating a highly scalable and available solution. Leaders missed or failed to address the issues of scale, and they allowed engineers to focus on solutions that the market was not demanding (Facebook doesn't have an equivalent of the f-graph more than 10 years later). Would Friendster have beaten Facebook had it properly focused on scalability and made the right leadership decisions (as Facebook obviously has) early in its product development life cycle? Maybe. Maybe not. But it certainly would have lived to fight another day, and probably would have gone on to have a much larger valuation and play a much larger part in social networking within the United States.

What Is Leadership?

For the purposes of this book, we will define leadership as “influencing the behavior of an organization or a person to accomplish a specific objective.” Leadership is perhaps most easily thought of along the lines of “pulling activities.” The establishment of a vision that inspires an organization “pulls” that organization toward the goal or objective. The setting of specific, measurable, achievable, realistic, timely goals along the way to that vision creates milestones that help the organization correct its course as it journeys to that destination. Had Friendster’s leadership, for instance, focused early on availability and scalability within their vision and measured progress in these areas, they likely would have given up on the f-graph earlier and focused on creating an acceptable experience.

Leadership applies to more than just an individual’s direct reports or organization. You can lead your peers, people in other organizations, and even your

1. Michael Fisher, Martin Abbott, and Kalle Lyytinen. *The Power of Customer Misbehavior*. London: Palgrave Macmillan, 2013.

management. Project managers, for instance, can “lead” a project team without being the person responsible for writing the reviews of the members of that team. Role models within an organization are leaders as well. As we will see, leadership is about what you do and how you influence the behaviors of those people around you, for better or worse.

Leadership is very different from management, however, and not everyone is good at both endeavors. That said, everyone can get better at both with practice and focus. Both leadership and management are also necessary to varying degrees in the accomplishment of most objectives. Consider the general goal of ensuring that your product, platform, or system can scale to meet incredible increases in end-user demand. You will obviously want to establish a vision that meets or exceeds that goal but that also recognizes the real-world budgetary constraints that we all face. In addition, you will want to make sure that every dollar you spend creates shareholder value (accretive) rather than destroys it (dilutive). In the business world, it’s not enough to be within budget—you need to ensure that you are not spending your precious funds significantly ahead of near-term demand, because idle equipment and unreleased code mean dilution of shareholder value.

Leaders: Born or Made?

No discussion of leadership, even in a brief chapter, would be complete without at least addressing the topic of whether leaders are born or made. Our response to this conundrum is that the question is really irrelevant when you consider our conceptual model (described next).

A person’s ability to lead is a direct result of his or her ability to influence behavior in individuals and organizations. That ability to influence behavior is a result of several things, some of which are traits with which a person is born, some of which are a result of the environment, and some of which are easily modified tools and approaches the person has developed over time.

When people say that someone is a “born leader,” they are probably talking about the person’s charisma and presence and potentially his or her looks. The importance assigned to the last element (good looks) is unfortunate in many respects (especially given the appearance of the authors), but it is an absolute truth that most people would rather be around “good-looking” people. That’s why there aren’t a whole lot of ugly politicians (remember, we’re talking about looks here and not political views). By comparison, charisma, personality, and presence are all things that are developed over time and are very difficult to change. While we typically believe that people are “born with” those characteristics, these elements are probably a result of not only genetics but also our environment. Regardless, we pin those characteristics on the person as if they are a birthright. We then jump to the conclusion that a leader must be “born,” because having good looks, a good personality, great presence, and great charisma absolutely help when influencing the behaviors of others.

In fact, looks, charisma, presence, charm, and personality are just a few of the many components of leadership; although they help, other components are just as important in establishing a leader's influence. As we will discuss in the next section, many attributes of leadership can and should be constantly honed to gain proficiency and improve consistently.

Leadership: A Conceptual Model

Let's first discuss a model we can use to describe leadership and to highlight specific components of leadership and explore how they affect scale. We believe that the ability to lead or influence the behaviors of a person or organization in regard to a goal is a function of several characteristics. We'll call these characteristics the "parameters" of our function. When arguments are provided to the parameters of the leadership function, it returns a score that's indicative of a person's ability to effect change and influence behaviors. This isn't the only way to think about leadership, but it's an effective conceptual model that illustrates how improvements in any given area can not only improve your ability to lead, but also offset things that you may not be able to change.

Some of the parameters of the leadership function represent attributes that the person "has." Potentially, the person may have been born with them, such as good looks and an outgoing personality. It's unlikely, without significant time spent under a surgeon's scalpel, that you will dramatically change the way you look, so we usually discount spending much time or thought on the matter. Likewise, if you are of modest intelligence, it's unlikely that you will ever be as good at immediately solving difficult problems on the spot as someone who is a super-genius. Neither of these relative deficiencies indicates that you can't be an effective leader; rather, each simply means that you are at a comparative disadvantage in a single element of the leadership equation.

Some of the leadership function parameters may be products of a person's upbringing or environmental issues—for example, charm, charisma, personality, and presence. These characteristics and attributes can and should be addressed, but they are difficult and tend to need constant attention to make significant and lasting change. For instance, a person whose personality makes her prone to disruptive temper tantrums probably needs to find ways to hold her temper in check. A person who is incapable of displaying anger toward failure probably needs to at least learn to "act" upset from time to time, as the power of showing displeasure in a generally happy person has a huge benefit within an organization. For instance, at some point you've probably said something like, "I don't think I've ever seen John mad like

that,” and more than likely this behavior made an impression that lasted for quite some time and helped influence your actions.

Some parameters of leadership have to do with what you can create through either innovation or perseverance. Are you innovative enough to create a compelling vision, or can you persevere to make enough attempts to create a compelling vision (assuming that you are consulting with someone to try it out on that person)? Innovation here speaks to the ability to come up with a vision on the fly, whereas perseverance to continue trying will take longer but can yield exactly the same result.

Other parameters of leadership relate to how you are perceived by others. Are you perceived to be a giver or a taker? Are you selfish or selfless? Are you perceived to be a very ethical person or morally bankrupt? Two important points arise here: the issue of perception and the effect of that perception on the team you are leading.

Everyone has probably heard the statement “Perception is reality,” and you have more than likely also heard the statement “Leaders are under a microscope.” When you’re a leader, everyone is watching you all of the time, and they will see you in your weakest moment and form an opinion of you based on that. Get used to it. That situation may not be fair, but it’s the way things work. If someone sees you accept free tickets to the Super Bowl from a vendor, he is very likely to jump to the conclusion that you are morally bankrupt or, at the very least, have questionable ethics. Why, after all, would you accept tickets from a vendor who is obviously trying to influence you to purchase the company’s products? Someone is inevitably going to catch you doing something at a low point in your day, and that something might not even be “bad” but simply taken out of context. The only thing you can do is be aware of this reality, and attempt as best you can to limit the number of “low points” that you have in any given day.

As to the effect of perception, we think the answer is pretty clear. Returning to the example of the Super Bowl tickets, the perception that you are willing to allow vendors to influence your decisions will undoubtedly have a negative impact on your ability to influence behaviors. Every vendor-related discussion you have with your team will likely be tainted going forward. After a meeting in which you indicate you want to include the vendor who supplied the Super Bowl tickets in some discussion, just imagine the team’s comments when you depart! You may have desired the inclusion of that vendor for all the right reasons, but that point just doesn’t matter. Your prior behavior caused significant damage to your ability to lead the team to the right answers.

The point of describing leadership as an equation is to drive home the message that although you may not be able to change some things, you can definitely work on many other things to become a better leader. More importantly, there is no maximum boundary to the equation! You can work your whole life to become a better leader and reap the benefits along the way. Make life your leadership lab, and become a lifelong student. By being a better leader, you will get more out of

your organization, and your organization will make decisions consistent with your vision and mission. The result is greater scalability, more benefit with less work (or rework), and happier shareholders.

Taking Stock of Who You Are

Most people are not as good at being leaders as they think. We make this assertion from our personal experience, and while relying on the Dunning–Kruger effect. In their studies, David Dunning and Justin Kruger witnessed that we often overestimate our abilities and noted that the overestimation is most severe where we lack experience or have a high degree of ignorance.² Given that very little formal leadership training is available in our universities and workplaces, we believe that leadership ignorance abounds and that, as a result, many people overestimate their leadership skills.

Few people are formally trained in how to be leaders. Most, however, have seen so many poor leaders get promoted for all the wrong reasons that they emulate the very behaviors they despise. Think we have it wrong? How many times in your career have you found yourself saying, “I will never do what he did just now if I have his job”? Now think through whether anyone in your organization is saying that about you. The answer is almost definitely “yes.” That situation has happened to us in the past, it will likely continue to happen to us over time, and we can almost guarantee that it is happening to you.

But that’s not the end of it. You can lead entirely the wrong way and still be successful, which encourages you to (mistakenly) associate that success with your leadership approach. Sometimes success happens by chance; your team just happens to accomplish the right things despite your wrong-headed approach. Sometimes success happens because you get great performance out of individuals for a short period of time by treating them poorly, but over the long term your behaviors result in high turnover and an inability to attract and retain the best people whom you need to accomplish your mission.

At the end of the day, to reap the scalability benefits that great leadership can offer, you need to measure where you are today. In their book *Resonant Leadership*, Richard Boyatzis and Annie McKee identify the three components necessary

2. Justin Kruger and David Dunning. “Unskilled and Unaware of It: How Difficulties in Recognizing One’s Own Incompetence Lead to Inflated Self-Assessments.” *Journal of Personality and Social Psychology* 1999;77(6):1121–1134. doi:10.1037/0022-3514.77.6.1121. PMID 10626367; David Dunning, Kerri Johnson, Joyce Ehrlinger, and Justin Kruger. “Why People Fail to Recognize Their Own Incompetence.” *Current Directions in Psychological Science* 2003;12(3):83–87.

for change in individuals as mindfulness, hope, and compassion.³ Mindfulness here is knowledge of oneself, including feelings and capabilities, whereas hope and compassion help to generate the vision and drivers for change. Unfortunately, as the Dunning–Kruger effect would argue, you probably aren't the best person to evaluate where you are today. All of us have a tendency to inflate certain self-perceived strengths and potentially even misdiagnose weaknesses.

Elite military units strip a potential leader down to absolutely nothing and force him to know his limits. They deprive the person of sleep and food and force the person to live in harsh climates, with the goal of getting the person to truly understand his strengths, weaknesses, and limitations. You likely don't have time to go through such a process, nor do the demands of your job likely require that you have that level of self-awareness. Your best option is a good review by your boss, your peers, and—most importantly—your subordinates. This approach is often referred to as a *360-degree* review process.

Ouch! An employee review sounds like a painful process, doesn't it? But if you want to know what you can do to get better at influencing the behavior of your team, what better place to go than to your team to ask that question? Your boss will have some insights, as will your peers. Nevertheless, the only people who can tell you definitively how you can improve their performance and results are the people whom you are trying to influence. Moreover, if you want good information, the process must be anonymous. People's input tends to be sanitized if they believe that there is the potential that you will get upset at them or potentially hold their comments against them. Finally, if you are really willing to go this far (and you should), you need to act on the information. Sitting down with your team and saying, "Thanks for the input; here's what I have heard on how I can improve," will go a long way toward creating respect. Adding the very necessary step of saying, "And here is how I am going to take action to improve myself," will go even further.

Of course, a self-evaluation that does not result in a plan for improvement is a waste of both your time and the time of your organization and management. If leadership is a journey, the review process described should help set your starting point. Now you need a personal destination and a plan (or route) to get there. Some books suggest that you rely upon and build your strengths; others suggest that you eliminate or mitigate your weaknesses. We think that your plan should include both a reliance on and strengthening of your strengths and a mitigation of your weaknesses. Few people fail in their objectives because of their strengths, and few people win as a result of their weaknesses. We must reduce the dilutive aspects of our leadership

3. Richard Boyatzis and Annie McKee. *Resonant Leadership*. Cambridge, MA: Harvard Business School Press, 2005.

by minimizing our weaknesses and increase our accretive aspects by multiplying and building upon our strengths.

Having discussed a model for leadership, along with the need to be mindful of your strengths and weaknesses, we will now look at several characteristics shared by some of the greatest leaders with whom we've had the pleasure of working. These characteristics include setting the example, leading without ego, driving hard to accomplish the mission while being mindful and compassionate about the needs of the organization, timely decision making, team empowerment, and shareholder alignment.

Leading from the Front

We've all heard the phrase "Set the example," and if you are a manager, you may even have used it during performance counseling sessions. But what does "Set the example" really mean, how do you do it, and how does it affect scalability?

Most people would agree that employees with an optimal work environment or culture produce much more than employees at a company with a less than desirable work environment or culture. Producing more with a similar number of employees is an element of scale, as the company producing more at a comparable cost structure is inherently more "scalable." Terrible cultures can rob employees of productivity; in such toxic environments, employees gather around the water cooler and gossip about the recent misdeeds of the boss or complain about how the boss abuses her position.

Evaluate the cultural norms that you expect of your team and determine once again whether you are behaving consistently with these cultural norms. Do you expect your organization to rise above the temptations of vendors? If so, you had best not take any tickets to a Super Bowl or any other event. Do you expect your organization to react quickly to events and resolve them quickly? If so, you should display that same behavior. If a person is up all night working on a problem for the betterment of the organization and the company, do you still expect that individual to be at work the next day? If so, you had better pull a few all-nighters yourself.

It's not enough to say that you wouldn't have your team do anything you haven't done. When it comes to behaviors, you should show your team that you aren't asking them to do anything that you don't do now! People who are perceived to be great leaders don't abuse their position, and they don't assume that their progression to their position allows them certain luxuries not afforded to the rest of the organization. You likely already get paid more—that's your compensation.

Having your administrative assistant get your car washed or pick your kids up from school might seem to be an appropriate job-related perk to you, but to the rest of the company it might appear to be an abuse of your position. You may not

care about such perceptions, but they destroy your credibility with your team and impact the result of the leadership equation. This destruction of the leadership equation causes employees to waste time discussing perceived abuses and may even make them think it's acceptable to abuse their positions in similar fashion, which wastes time and money and reduces organizational scale. If such benefits are voted on by the board of directors or approved by management as part of your compensation package, you should request that they be paid for separately and should not rely on company employees to perform the functions.

The key here is that everyone can increase the value of his or her leadership score by "leading from the front." Act and behave ethically and do not take advantage of your position of authority. Behave exactly as you expect your organization to behave. If you abide by these rules, you will likely notice that your employees emulate your behaviors and that their individual output increases, thereby increasing overall scale of the organization.

Checking Your Ego at the Door

There is simply no place for a big ego in any position within any company. Certainly, there is a high degree of correlation between passionate inspirational leaders and people who have a need to talk about how great, intelligent, or successful they are. We would argue, however, that those people would be even more successful if they kept their need for publicity or public recognition to themselves. The concept isn't new; in fact, it is embodied in Jim Collins's concept of Level 5 Leadership in his wonderful book *Good to Great*.

CTOs who need to talk about being the "smartest person in the room" and CEOs who say, "I'm right more often than I'm wrong," simply have no place in a high-performing team. Put bluntly, they are working as hard as they can to destroy the team by making such statements. Focusing on an individual rather than the team, regardless of who that individual is, is the antithesis of scale; scale is about growing cost-effectively, and a focus on the one rather than the many is clearly a focus on constraints rather than scale. Such statements alienate the rest of a team and very often push the very highest-performing individuals—those actually getting stuff done—out of the team and out of the company. These self-aggrandizing actions and statements run counter to building the best team and over time will serve to destroy shareholder value.

The best leaders give of themselves selflessly in an ethical pursuit of creating shareholder value. The right way to approach your job as a leader and a manager is to figure out how to get the most out of your team so as to maximize shareholder wealth. You are really a critical portion of that long-term wealth creation cycle only

if your actions evolve around being a leader of the team rather than an individual. Take some time to evaluate yourself and your statements through the course of a week. Identify how many times you reference yourself or your accomplishments during your daily discussions. If you find that you are doing it often, step back and redirect your thoughts and your statements to things that are more team oriented than self-oriented.

It is not easy to make this type of change. Too many people around us appear to be rewarded for being egoists and narcissists, and it is easy to reach the conclusion that humility is a character trait embodied by the unsuccessful business person. To counter this perception, all you need to do is reflect on your career and identify the boss to whom you had the greatest loyalty and for whom you would do nearly anything; that boss most likely put the shareholders first and always emphasized the team. Be the type of person who thinks first about how to create shareholder value rather than personal value, and you will succeed!

Mission First, People Always

As young leaders serving in the U.S. Army, we were introduced to an important concept in both leadership and management: Leaders and managers accomplish their missions through their people. Neither getting the job done at all costs nor caring about your people makes a great leader; great leaders know how to do both, even in light of their apparent contradictions. Broadly speaking, as public company executives, managers, or individual contributors, “getting our jobs done” means maximizing shareholder value. We’ll discuss maximizing shareholder value in the section on vision and mission.

Effective leaders and managers get the mission accomplished; great leaders and managers do so by creating a culture and environment in which people feel appreciated and respected and wherein performance-related feedback is honest and timely. The difference here is that the latter leader—the one who creates a long-term nurturing and caring environment—is leading for the future and will enjoy the benefits of greater retention, loyalty, and long-term performance. Caring about people means giving thought to the careers and interests of your employees; it means giving timely feedback on performance and, in so doing, recognizing that even stellar employees need feedback regarding infrequent poor performance (how else can they improve). Great leaders ensure that those persons who create the most value are compensated most aggressively, and they ensure that people get the time off that they deserve for performance above and beyond the call of duty for their individual positions.

Caring about people does *not* mean creating a sense of entitlement or lifetime employment within the organization. We will discuss this point more in the

management chapter. Caring also does *not* mean setting easy goals, as in so doing you would not be accomplishing your mission of creating shareholder value.

It is very easy to identify “Mission First” leaders because they are the ones who are getting the job done even in the face of adversity. It is not so easy to identify “Mission First, People Always” leaders because it takes a long time to test whether the individual leader has created a culture that inspires people and makes high-performance individuals want to follow the person from job to job because he or she is a caring individual. The easiest “People Always” test to apply for a seasoned leader is to find out how many direct reports have followed that manager consistently from position to position within successful organizations. “Mission First, Me Always” leaders find that their direct reports will seldom work for them in more than one or two organizations or companies, whereas “Mission First, People Always” leaders seldom have problems in getting their direct reports to follow them through their careers.

“Mission First, Me Always” leaders climb a ladder with rungs made of their employees, stepping on them as they climb to the top. “Mission First, People Always” leaders build ladders that all of the stellar performers can climb.

Making Timely, Sound, and Morally Correct Decisions

Your team expects you to help resolve major issues quickly and with proper judgment. Rest assured that you are going to make mistakes over time: Welcome to humanity. But on average, you should move quickly to make the best decision possible with the proper input without wasting a lot of time. Be courageous and make decisions. That’s what being a leader is entirely about.

Why did we add “morally correct” in this point? Few things destroy shareholder value or scale more quickly than issues with your personal ethics. We asserted earlier that leaders are always under a microscope and that you will undoubtedly be caught or seen doing things you wish you hadn’t done. Our hope is that those behaviors are nodding off at your desk because you’ve been working too hard or running out to your car to perform a personal errand during work hours because you worked too late at night to perform it. Ideally, such behaviors will not include things like accepting tickets for major sporting events for the reasons we’ve previously indicated. We also hope that they don’t include allowing others within your team to do the same.

One of our favorite quotes goes something like “What you allow, you teach, and what you teach becomes your standard.” Here, *allow* refers to either yourself or others. Nowhere does that ring more true than with ethical violations, large and small. We’re not sure how corruption at companies like Tyco or Enron ultimately starts. Nor are we certain how a Ponzi scheme as large as Bernard Madoff’s criminal activity, which destroyed billions of dollars of wealth, can possibly continue

for so many years. We do know, however, that they could have been stopped long before the problems grew to legendary size. We also know that each of these events destroyed the size and scale of the companies in question, along with a great deal of shareholder value.

We don't believe that people start out plotting billion-dollar Ponzi schemes, and we don't believe that people start off by misstating tens or hundreds of millions of dollars of revenue or embezzling tens of millions of dollars from a company. We're fairly certain that this behavior starts small and slowly progresses. People get closer and closer to a line they shouldn't cross, and then they take smaller and smaller steps into the abyss of moral bankruptcy until it is just too late to do anything.

Our answer is to never start. Don't take company pens, don't take favors from vendors who wish to sway your decisions, and don't use the company plane for personal business unless it is authorized by the board of directors as part of your compensation package. Few things will destroy your internal credibility and, in turn, your ability to influence an organization as thoroughly as the perception of impropriety. There is no way you can align lying, cheating, or stealing with the creation of shareholder wealth.

Empowering Teams and Scalability

Perhaps no leadership activity or action impacts an organization's ability to scale more than the concept of team empowerment. Empowerment is the distribution of certain actions, accountability, and ownership. It may include giving some or all components of both leadership and management to an individual or an organization.

The leadership aspects of empowerment come from the boost that individuals, teams, leaders, and managers get out of the feeling and associated pride of ownership. Individuals who believe they are empowered to make decisions and own a process in general are more productive than those who believe they are merely following orders. Mechanically, the leader truly practicing empowerment multiplies his or her organization's throughput, as he or she is no longer the bottleneck for all activities.

When empowerment happens in statement only, such as when a leader continues to review all decisions for a certain project or initiative that has been given to an empowered team, the effects are disastrous. The leader may indicate that he is empowering people, but in actuality he is constraining the organizational throughput by creating a chokepoint of activity. The teams immediately see through this ruse; rather than owning the solution, they feel as though they are merely consultants to the process. Worse yet, they may feel absolutely no ownership and as a result neither experience the gratification that comes with owning a solution or division nor feel the obligation to ensure that the solution is implemented properly or the division run well. The net result is that morale, throughput, and trust are destroyed.

This is not to say that in empowering individuals and teams, the leader is no longer responsible for the results. Although a leader may distribute ownership, that individual can never abdicate the responsibility to achieve results for shareholders. When delegating and empowering teams, one should be clear as to what the team is empowered to do. The trick is to give the team or individual enough room to maneuver, learn, and create value, while still providing a safety net and opportunities to learn. For instance, a small corporation is likely to limit budgetary decisions for managers to no more than several thousand dollars, whereas a division chief of a *Fortune 500* company may have latitude within the budget up a few million dollars. These limitations should not come as a surprise in the empowerment discussions, as most boards of directors require that they approve large capital expenditures.

Alignment with Shareholder Value

Everything you do as a leader and manager needs to be aligned with shareholder value. Although we probably shouldn't absolutely need to say this, we've found in our practice that this concept isn't raised enough within companies. Put simply, if you are in a for-profit business, your job is to create shareholder wealth. More importantly, your job is to *maximize* shareholder wealth. You will probably get to keep your job if your actions and the actions of your team make your shareholders wealthier. You will be considered the best if you make shareholders more money than anyone else. Even if you are in a nonprofit organization, you are still responsible for creating a type of wealth. The wealth in these organizations is more often the emotional wealth creation of the people who donate to the organization if it is a charity, or the other type of "good" that you do for the people who pay you if the organization is something other than a charity. Whatever the reason, if you aren't attempting to maximize the type of wealth the company sets out to make, you shouldn't be in the job.

As we discuss concepts like vision, mission, and goals, a test that you should apply is this simple question: "How does this create and maximize shareholder wealth?" You should be able to answer this question in relatively simple terms; later in this chapter, we will discuss what we call the "causal roadmap" and its impact on individuals within nearly any job. Moreover, you should find ways to ask the shareholder-wealth-creation question in other aspects of your job. Why would you ever do anything in your job that doesn't somehow help create shareholder wealth or keep natural forces like growth and the resulting instability of your systems from destroying it? Why would you ever hire a person who isn't committed to creating shareholder wealth? Why would you ever assign a task not related to the creation of shareholder wealth?

Transformational Leadership

A great deal of research has examined how the most effective leaders with the greatest results tend to engage with their teams. The most effective leaders influence organizations through ideals, put aside their own self-interest in favor of that of the team, provide intellectual stimulation for the team and its members, and display honest and personal consideration for the team members' well-being and their careers.⁴ These characteristics stand in stark contrast to the characteristics of how many leaders operate, trading value for results in a *quid pro quo* or transactional basis. Sometimes called "transactional" or *quid pro quo* leadership, this horse trading comes in the form of "If you accomplish goal X, I will give you reward Y." From an operational perspective, it could be stated as "Deliver 99.99% availability for me and I'll promote you to VP"; from an engineering perspective, it may be cast as "Deliver single sign-on and the next-generation shopping cart and you will max out your bonus." Hang on—don't give up too quickly. We have all given into transactional leadership from time to time. The question isn't whether you do it, but how much you do it in comparison to the loftier leadership that is characterized by idealized influence and often called "transformational leadership."

Transformational leadership simply scales better. Rather than focusing on transactions with individuals, it focuses on the team. Rather than emphasizing oneself, it discusses the greater good. What might otherwise become a discussion with a single individual now becomes a discussion about what the team should aspire to be and accomplish. Whereas transactional leaders focus on the completion of individual tasks and goals, transformational leaders concentrate on presenting a compelling vision that pulls individuals to the right results even when the leader isn't present. In addition, transformational leadership has been shown to reduce affective conflict (the bad role-based conflict described earlier in this book) and increase value-creating cognitive conflict.⁵

Vision

It is our experience that, in general, leaders don't spend enough time on vision and mission. Both of these elements tend to be something that is addressed as a result of a yearly planning session, with the leader, with or without the team, spending

4. B. M. Bass. "Two Decades of Research and Development in Transformational Leadership." *European Journal of Work and Organizational Psychology* 1999;8(1):9–32.

5. Marty Abbott. "Mitigating a Conflict Paradox: The Struggle between Tenure and Charisma in Venture Backed Teams." <http://digitalcase.case.edu:9000/fedora/get/ksl:waeadm382/waeadm382.pdf>.

perhaps an hour or two discussing it. Think about that for a minute: You spend an hour or two talking about the thing that will serve as the guiding light for your company or organization. Does that sound right? Your vision is your destination; it's something that should inspire people within your company, attract outside talent, help retain the best of your current talent, and help people understand what they should be doing in the absence of a manager standing over their shoulders. You probably spend more time planning your next vacation than you do pondering the destination for your entire company. We hope we've convinced you that the creation and communication of a compelling vision is something that's incredibly important to the success of your company and nearly any initiative.

Vision is the description of a destination for a company, an organization, or an initiative. It is a vivid description of the ideal future—the desired state or outcome in a clearly defined, measurable, and easily committed to memory package. Ideally, the vision will prove inspirational to the team and can stand alone as a beacon of where to go and what to do in the absence of further leadership or management. It should be measurable and testable, meaning that there is an easy way to determine whether you are actually at that point when you get there. It should also incorporate some portion of your beliefs so that it has meaning to the organization.

U.S. Pledge of Allegiance

The U.S. Pledge of Allegiance has an inspirational destination or vision as one of its components. "One Nation under God, indivisible, with Liberty and Justice for all" is a vivid description of an ideal future.

The Pledge is certainly testable at any given point in time by dividing it into its subcomponents and determining whether they have been met. Are we "one nation"? Apparently as we are still governed by a single government and the attempt during the Civil War to split the country was not successful. Are we "under God"? This is debatable, but certainly not everyone believes in God if that were the intent of the passage; we'll discuss this portion of the Pledge in the next paragraph. Do we have liberty and justice for all? We have laws, and everyone is theoretically governed by those laws and given some set of what we call "inalienable rights." Of course, intense debate is ongoing as to whether the laws are effective and whether we apply them equally based on color, sex, beliefs, and so on, but the U.S. system in general appears to work as well as any system anywhere else. At the very least, you can agree that there is a test set up in this vision statement—the debate would be about whether everyone who applies the test will come up with the same answer.

An interesting side note here is that the phrase "under God" was added in the 1950s after years of lobbying by the Knights of Columbus and other religious orders. Regardless of your religious beliefs, the modification of the Pledge of Allegiance illustrates that visions can be modified over time to more clearly define the end goal. Vision, then, can be modified as the

desired outcome of the company, organization, or initiative changes. It does not need to be static, but should not be so fluid that it is modified on a whim.

The entire Pledge of Allegiance hints at a shared belief structure; elements such as the unity of the nation, a nod toward religious belief, and the equal distribution of “liberty” and “justice.” As to value creation, the value created here is really in supporting the shared values of the members or citizens of the United States. Although not universally true even at the time of the creation of the Pledge, most people within the United States would agree that equality in liberty and justice is value creation in and of itself.

The U.S. Declaration of Independence is an example of vision, though a majority of the document describes the vision in terms of the things we do not want to be. Much of this document focuses on actions taken by the British king, which were not desirable in the idealized future. Although such an approach can certainly be inspirational and effective in motivating organizations, it takes too much time to define an end state based on what the end state is not. For this reason, we suggest that vision be brief and that it define the desired end state.

The beginning of the Preamble to the U.S. Constitution is another example of a vision statement: “We the People of the United States, in Order to form a more perfect Union.” Although this passage certainly describes an ideal future, it is difficult to determine whether you can really “know” that you are there when you reach that future. What exactly is a “perfect union”? How do you test “perfection”? Perfection certainly would create value—who wouldn’t want to live in and contribute to a “perfect union”? The Preamble gets high marks for inspiration, memorizability, and value creation, but relatively low marks for measurability. It’s hard to describe how vivid it is, because it is difficult to determine how you would know you are truly there at the end state. Finally, the Preamble hints at certain beliefs but does not explicitly incorporate them.

Perhaps the easiest way to envision a vision statement is to view it within the context of giving someone directions. One of the things you are likely to do when giving directions is to provide criteria by which to determine whether the journey to the destination was a success. Assume that a couple is asking us to give them directions to a department store where they can find nearly anything they need at a reasonable price for their new apartment. We decide that the best place to send them is the local Walmart store. Let’s further assume that the people we are directing have never been to a Walmart store (maybe they are space aliens or perhaps they just emerged from a religious cult’s underground bomb shelter).

One of the first things we are likely to do in this case is give these rather strange people an indication of how they will know they have been successful in their journey. Perhaps we might say something like “The best place for you to go is Walmart,

because it has the lowest prices within the 10 miles surrounding this area and I know you don't want to travel far. The local Walmart is a big white building with blue trim and huge letters on the top that spell out WALMART. The address is 111 Sam Walton Street. The parking lot is huge, and within the parking lot you will find several other stores such as a McDonald's restaurant and a gas station. Across the street you will see two more gas stations and a strip mall." Such a description is not only vivid, but also outlines a set of tests that will indicate to our travelers exactly when they have arrived at their destination.

We've accomplished everything we wanted to do in creating our vision statement. We gave our travelers an inspiration—"lowest prices within the 10 miles surrounding this area" is inspiring because it meets their needs of goods being inexpensive and the location not being too distant. We've provided a vivid description of the ideal state—arriving at Walmart and giving the travelers an indication of what the site will look like when they arrive there. In summary, we've given our travelers a set of tests to validate that their initiative was successful, in the form of the vivid description and the beliefs implicitly identified in the need for low prices. The vision is easily committed to memory. The couple can look at the street address and determine for certain that they've definitely arrived at the appropriate location.

We suggest that you research other statements of vision and use the rules we have identified before creating your own. Apply these rules or tests and figure out which ones work for you. As a reminder, a vision statement should meet the following criteria:

- Vivid description of an ideal future
- Important to value creation
- Measurable
- Inspirational
- Incorporate elements of your beliefs
- Mostly static, but modifiable as the need presents itself
- Easily remembered

Mission

If vision is the vivid description of the ideal future or the end point of our journey, mission is the general path or actions that will get us to that destination. The mission statement focuses more on the present state of the company, as that present state is important to get to the desired state or "vision" of the company. It should incorporate a sense of purpose, a sense of what needs to be done *today*, and a general

direction regarding how to get to that vision. Like a vision statement, the mission statement should also be testable. The test for the mission statement should include the determination of whether, if properly executed, the mission statement will help drive the initiative, organization, or company toward the vision of the company.

Let's return to our analysis of the Preamble of the U.S. Constitution to see if we can find a mission statement. The passage "establish Justice, ensure domestic Tranquility, provide for the common defence, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution for the United States of America" appears to us to be the mission statement of the Preamble. The entire remainder of the quoted passage serves to implicitly establish an existing state. By their need to "establish" these things, the founding fathers are indicating that they do not exist today. The purpose of the United States is also explicitly called out in the establishment of these things. These actions also serve to identify the creation of the vision: a "perfect Union." Testability, however, is weak and suffers from the subjective analysis necessary to test any of the points. Has the United States ensured domestic tranquility after more than 200 years of existence? We still suffer from domestic strife along the boundaries of race, belief structure, and sex. The country has certainly spent a lot of money on defense and in general has probably performed well relative to other nations, but have we truly met the initial goals? What about general welfare? Does the rising cost of health care somehow play into that definition? By now, you've probably gotten the point.

Now, we return to the directions that we were giving our rather strange travelers who have never seen a Walmart. After providing the vision statement describing where the couple need to go, and consistent with our definition of a mission statement, we need to give them general directions or an approach for how to get there. We need to indicate the present state, a sense of purpose, and a general direction. The mission statement could then simply be "To get to Walmart from here, drive mostly southwest roughly 7 miles."

That's it; we've accomplished everything within a single sentence. We've given the travelers purpose by stating "To get to Walmart." We've given them an indication of the current position in the form of "from here," and we've given them a general direction to the vision, "drive mostly southwest roughly 7 miles." The whole mission is testable, as the couple can clearly see where they are (although most mission statements should be a bit more descriptive), they already know how to understand their destination, and they have a direction and limit to determine when they are out of bounds.

As a review, a mission statement should meet the following criteria:

- Be descriptive of the present state and actions.
- Incorporate a sense of purpose.

- Be measurable.
- Include a general direction or path toward the vision.

Now you might state, “But that doesn’t really get them to where they need to go!” You are correct, and that’s a perfect segue into a discussion of goals.

Goals

If vision is our description of where we are going and mission is a general direction on how to get there, goals are the guideposts or mile markers to ensure that we stay on track during our journey. In our minds, the best goals are achieved through the SMART acronym.

SMART goals are

- Specific
- Measurable
- Attainable (but aggressive)
- Realistic
- Timely (or contain a component of time)

Going back to the Constitution, we can look at many of the amendments as goals that Congress desired to achieve en route to its vision. As an example, consider the abolition of slavery in the 13th Amendment. This amendment was obviously meant as a goal on the path to the equality promised within the vision of a perfect union and the mission of securing the blessings of liberty. The text of this amendment is as follows:

Section 1. Neither slavery nor involuntary servitude, except as a punishment for crime whereof the party shall have been duly convicted, shall exist within the United States, or any place subject to their jurisdiction.

Section 2. Congress shall have the power to enforce this article by appropriate legislation.

The 13th Amendment is specific in terms of “who,” “what,” and “where,” and it implies a “when.” Congress (“who”) has the power to enforce the article, and everyone (another “who”) is subject to the article’s rule. The “where” is the United States, and the “what” is slavery or any involuntary servitude except as punishment for crimes.

The amendment is measurable in its effect, because the presence of slavery is binary: Either it exists or it does not. The result is attainable, as slavery is abolished, although from time to time strange pockets of people practicing enslavement

of others pop up and are handled by law enforcement personnel. The goal is realistic and it was time bounded as the amendment took immediate effect.

Returning now to our example of guiding our friends to their Walmart destination, how might we provide the couple with goals? Recall that goals don't tell someone *how* to do something, but rather indicate whether that person is on track. In our example, we defined a vision or ideal end state for our travelers, which was the local Walmart store. We also gave them a mission, or a general direction to get there: "To get to Walmart from here, drive mostly southwest roughly 7 miles." Now we need to give them goals to ensure that they keep on track in their journey to the final destination.

We might give this couple two goals: one identifying the end state or vision and an interim goal to help them on their way. The first goal might look like this: "Regardless of your path, you should be in the center of town as identified by the only traffic light within 10 minutes of leaving here. That is roughly halfway to Walmart." This goal is specific, describing where the travelers should be. It is measurable in that they can easily tell that they have achieved it. It is both attainable and realistic because if we expect the couple's car to move at an average speed of 30 miles per hour and travel a distance of 7 miles, they should be able to travel a mile every 2 minutes and anything over 10 minutes (5 miles) would put them somewhere other than where they should be. The inclusion of a 10-minute time interval means that our goal is time bounded.

The last goal we give our bargain-hunting friends will deal with the end vision itself. It is also a simple goal, as we've already described the location: "You should arrive at the Walmart located at 111 Sam Walton Street in no more than 20 minutes." Our statement is specific, measurable, achievable, realistic, and time bounded. Bingo!

Have we given the travelers everything they need to be successful? Are we missing anything? You are probably wondering why we haven't given them turn-by-turn directions. What do you think the reason is? Okay, we won't make you wait. You might recall our definition of leadership versus management, where leadership is a "pulling" activity and management is a "pushing" activity. Explaining or defining a path to a goal or vision is a management activity, and we'll discuss that in the next chapter.

Putting It All Together

We've now spent several pages describing vision, mission, and goals. You might again be asking yourself what, exactly, this has to do with scalability. Our answer is that it has *everything* to do with scalability. If you don't define where you are

going and you don't provide a general direction to get there and a set of goals to help identify that your team is on the right path, then you are absolutely doomed. Yes, we took a bit of time defining leadership, some characteristics of a good leader, and some points to consider, but all of those things were absolutely important to the success of any initiative, and we felt obliged to discuss them if even just at a high level.

Now that we've done so, let's see how we can put these things to use in the development of a vision, mission, and set of goals for a company from the authors' work history. Quigo was a privately held advertising technology company founded in 2000 in Israel by Yaron Galai and Oded Itzhak. By 2005, the company had completed a round of financing led by Highland Capital Partners and Steamboat Ventures and had moved its operations to New York City. The company's first product was technology that enabled it to analyze pages of content and suggest words that would be appropriate to "buy" on auction-based advertising networks such as Google's AdWords/AdSense and Yahoo's search advertising platforms. This technology, known colloquially as search engine marketing (SEM), was offered by Quigo as a service under the name FeedPoint to companies wishing to buy search and contextual advertising.

By 2005, Quigo's founders and a relatively newly hired CEO, Mike Yavonditte, decided to leverage aspects of FeedPoint's technology to offer a competitively differentiated contextual advertising product similar to Google's AdSense. Unlike AdSense, Quigo's product (known as AdSonar) would have a constrained set of keywords, whereas Google's list of keywords was nearly limitless. Keywords are the words that a customer "buys" so that its advertisement shows up when a search term uses that keyword or a page is determined (through contextual and semantic analysis) to be relevant to that keyword. Further, the AdSonar product would allow individual pages to be purchased, and the product would be presented under the publisher's brand rather than Quigo's brand. The idea driving this "white label" product was that it would allow publishers to extract the maximum value for their own brand, rather than Quigo's brand. The theory behind the idea was that advertisers would pay more for the keyword "football" on ESPN than they would on a site catering to weather information. As such, publishers like ESPN should make more money off of their premium branded sites and content than they would through another, non-white-label product.

When AdSonar launched, the theory seemed to hold true. Advertisers appeared to be willing to spend more on premium and vertically focused sites than they were on keywords within competing products. Unfortunately, Quigo's product was not initially designed to meet the onrush of demand from those publishers adopting AdSonar. The site would stall or become unavailable and in so doing would sometimes cause publisher sites not to render pages. Publishers complained that if something were not done soon, they would go back to their prior solutions.

The authors joined the company in the summer of 2005 and were welcomed with a new product that was crumbling beneath ever-increasing demand. What was even scarier than the fact that customers were threatening to quit is that the largest customers in the pipeline hadn't even launched yet! It was clear that the platform needed both to be more highly available and to scale to meet demand that would exceed 10 times the current volume of traffic—all within a year. To explain to the team, the board, and the remainder of the company executives what we (jointly) needed to accomplish, we needed to create a vision around our initiative. Recall our points about vision statements:

- Vivid description of an ideal future
- Important to shareholder value creation
- Measurable
- Inspirational
- Incorporate elements of your beliefs
- Mostly static, but modifiable as the need presents itself
- Easily remembered

We decided on something like the following vision for the product's availability and scalability (we've modified this slightly from the original):

AdSonar will all achieve 99.95% availability as measured by impact to expected publisher/partner revenue. We will never have a customer-impacting event from an inability to scale on time.

We then focused on creating a mission for the team around availability and scalability. Recall that an effective mission has the following components:

- Is descriptive of the present state and actions
- Incorporates a sense of purpose
- Is measurable
- Includes a general direction or path toward the vision

Our mission, as it related to scale and availability (our biggest roadblocks to success) became something like this:

To move from a crisis of availability to our vision of a highly available and scalable solution within two quarters by evolving to a fault-tolerant implementation.

The mission was measurable (it referenced our vision and therefore the 99.95% availability), had a component of time to achieve it (two quarters), and explained roughly how we would achieve it (implementing fault-tolerant designs—something we discuss in Chapter 21, Creating Fault-Isolative Architectural Structures).

Now we needed to set goals along the route to our mission. Recall that goals should be

- Specific
- Measurable
- Attainable (but aggressive)
- Realistic
- Timely (or contain a component of time)

Unfortunately, the operational processes at Quigo didn't afford us a great deal of data on the sources and causes of outages. Goal 1, then, was to create a goal for the head of operations to implement daily service meetings and robust incident and problem management to resolve these issues within 30 days. We didn't specify exactly how to do this—just that we needed to understand the impact of incidents (e.g., duration, revenue) and the root causes of the incidents. All of the SMART attributes were achieved.

The data that we did have available indicated that network components and email services were the primary source (problems) of many of the undesirable incidents. The systems were built “on the cheap,” but not in a good way. Many network devices were singletons; thus, when they failed, they would bring the entire product down. Goal 2, then, was to make all critical network servers redundant within 60 days, which allowed for sufficient hardware lead times within a small company. Because we didn't have a great deal of information on the services failures, we also set a goal to identify the top five sources of incidents within a period of 30 days and to develop a plan to fix each.

A fair amount of information also pointed to time-to-respond as a major contributor to downtime. Specifically, for any problem, no one would know there was an issue until a customer complained. Often, for problems happening overnight, this would mean several hours of impact before any action would be taken. Clearly we needed better monitoring, in the form of monitoring from a third-party service that tested not only our servers but also our customer's ability to connect to those services through Internet public transport services. The goal, then, was to reduce time-to-know for all failures of advertising services transaction availability before a customer called and to implement such a capability within 30 days.

Lastly, we wanted an overarching goal that generally helped show the path to success. While the company had not previously tracked availability consistently and absolutely did not do so by customer impact, we believed that Quigo needed to implement monthly increasing goals on the path to 99.95%. We didn't know where Quigo stood, but we knew that for the benefit of its customers, its business, and its shareholders, the company needed to get there within 6 months. As such, we set the

first month's goal at 99.90%, the second month's goal at 99.91%, the third month's goal at 99.92%, the fourth month's goal at 99.93%, and so on.

It would be ridiculous (and completely untrue) if we ended the Quigo story with "And that's all we had to do to turn the company around and sell it for a tidy profit to the shareholders." In fact, a lot of heavy lifting was required on the part of a lot of people to ultimately make Quigo a success. We'll cover some of that heavy lifting in the process and technology sections of this book. For now, we emphasize that this initial setting of vision, mission, and goals was absolutely critical to the company's ultimate success.

The Causal Roadmap to Success

One of the best and easiest things you can do as a leader to help ensure success within an organization is help people understand how what they do every day contributes to the vision of the organization and, as a result, to the creation of shareholder value. We call this creation of understanding the *causal roadmap to success*. The causal roadmap is relatively easy to create for people, and if it isn't, there is a good reason to question whether the job should exist.

Let's take a look at how we might create this causal roadmap to success for some different skill sets and teams within a company:

Operations teams (sometimes called network operations teams or applications operations teams) are responsible for helping to ensure that solutions or services are available, thereby keeping the company from experiencing lost opportunity (downtime that might mean a reduction in revenue or the loss of customers). Ensuring that services are "always on" contributes to shareholder value by supporting the ongoing delivery of revenues, which in turn maximizes profits. Increasing profits increases the price that shareholders should be willing to pay per share and, therefore, increases shareholder value.

Quality assurance professionals help reduce lost opportunities associated with the deployment of a product *and* the cost of developing that product. By ensuring that the product meets the needs of customers (including scalability needs), these professionals help ensure happier and more productive customers. This, in turn, engenders greater usage of the product and higher customer retention. Happier customers also refer more sales from other customers. Furthermore, because QA professionals are focused on testing solutions, software engineers are freed up to spend more time on engineering (rather than testing) solutions. As QA professionals tend to come at a discount to engineers, the total cost per unit developed goes down. Lower costs of production mean greater profits; happier customers mean more purchases and higher revenues. The two net out to bigger profit margins and happier shareholders.

Although we've painted some of the picture here of how the causal roadmap works in general, it is (and should be) a little more complicated than just blurting out a paragraph describing how an organization impacts the company's profitability. One-on-one discussions should take place between a leader and each member of the team about this relationship. We do not mean that a CEO should talk to each of the 5,231 people in the company, but rather that a manager of individual contributors should speak to each and every one of his or her employees. The director, in turn, should speak to each of his or her managers, the VP to each of his or her directors, and so on. Each conversation should be personal and tailored to the individual. The underlying reasons for engaging in the discussion will not change, but the message should be tailored to exactly what that individual does.

Furthermore, people need to be reminded of what they do and how it affects the maximization of shareholder wealth. This isn't a one-time conversation. It's also not really focused on delivering performance feedback (although you can certainly work that in), but rather on ensuring that the person has purpose and meaning within his or her job. The impact on retention from such reminders is meaningful, and it can potentially help employees produce more. A happy employee, after all, is a productive employee—and a productive employee is producing more to help you scale more!

Conclusion

Leadership is the influencing of an organization or person to accomplish a specific objective. Our mental model for thinking about leadership is a function consisting of personal characteristics, skills, experiences, and actions. Becoming a better leader starts with knowing where you are weak and where you are strong within the leadership function.

Leadership can impact the scalability of your team and your company in many ways. Poor leadership puts limits on the growth and output of your company and your team. Great leadership, in contrast, serves as an accelerator for growth, allowing organizations to grow in total size and in output per individual. By becoming a better leader, you can increase the capabilities and capacity of your organization and your company.

Successful leaders are characterized by certain behaviors, including leading by example, leaving your ego at the door, leading selflessly, and accomplishing the mission while taking care of your team. Leaders who focus on the greater good (transformational leadership), rather than engaging in transactions with their employees, generally get better results. For all these reasons, you should always be thinking of how to lead ethically and recognize that everything you do should be aligned with shareholder value.

The components of vision, mission, and goals all play into the leader's role, and the SMART acronym can be used to advantage in the creation of scalability goals. Finally, the causal roadmap to success emphasizes helping your organization to tie everything that it does back to what is important to the company: the maximization of shareholder value.

Key Points

- Leadership is influencing the behavior of an organization or a person to accomplish a specific objective.
- Leaders, whether born or made, can get better; in fact, the pursuit of getting better should be a lifelong goal.
- Leadership can be viewed as a function consisting of personal characteristics, skills, experiences, actions, and approaches. Increasing any aspect increases your leadership "quotient."
- The first step in getting better as a leader is to know where you stand. To do so, get a 360-degree review from your employees, your peers, and your manager.
- Lead as you would have people follow—abide by the culture you wish to create.
- There is no place for ego when leading teams—so check your ego at the door.
- Leadership should be a selfless endeavor.
- Mission First, People Always. Get the job done on time, but ensure you are doing it while taking care of your people.
- Always be morally straight. What you allow, you teach, and what you teach becomes your standard.
- Align everything you do with shareholder value. Don't do things that don't create shareholder value.
- Transformational leaders focus on the team overall, not on individuals. Don't engage in leadership transactions with individuals (*quid pro quo*), but rather focus the team on collaboration and team outcomes.
- Vision is a vivid description of an ideal future. It includes the following components:
 - Vivid description of an ideal future
 - Important to shareholder value creation
 - Measurable
 - Inspirational
 - Incorporate elements of your beliefs

- Mostly static, but modifiable as the need presents itself
- Easily remembered
- Mission is the general path or actions that will get you to your vision. It includes the following components:
 - Descriptive of the present state and actions
 - A sense of purpose
 - Measurable
 - General direction or path toward the vision
- Goals are the guideposts to your vision and are consistent with the path of your mission. SMART goals are
 - Specific
 - Measurable
 - Attainable (but aggressive)
 - Realistic
 - Timely (or contain a component of time)
- The causal roadmap to success will help you frame your vision, mission, and goals, and will help employees understand how they contribute to those goals and aid in the creation of shareholder value.

Chapter 5

Management 101

In respect of the military method, we have, firstly, Measurement; secondly, Estimation of quantity; thirdly, Calculation; fourthly, Balancing of chances; fifthly, Victory.

—Sun Tzu

Scott Cook cofounded Intuit with Tom Proulx in 1983. Scott's experience at Proctor & Gamble made him realize that personal computers would ultimately replace pencil-and-paper accounting.¹ That realization, coupled with a desire to create something to help his wife with the mundane task of "paying the bills," became the inspiration for the company's first product, the home finance software solution Quicken.² As Intuit grew, the company continued to improve Quicken, which competed against other consumer financial packages like Microsoft's Money. In 1992, the company launched QuickBooks—a peer to Quicken focused on small- and medium-sized business bookkeeping. In 1993, the company went public and then merged with Chipsoft, the makers of TurboTax, through a stock swap.³ Since 1993, the company has made dozens of acquisitions focused on both consumer and small-business financial solutions. As of 2014, Intuit was a highly successful global company with revenues of \$4 billion, net income of \$897 million, and 8,000 employees.

Experience tells us that at the heart of every success story are a number of critical moments—moments where leaders, managers, and teams either do or do not rise to and capitalize on the challenges and opportunities with which they are presented. Intuit's story begins with shifts in consumer purchase behaviors in the late 1990s. Intuit's first offerings were all desktop- and laptop-based products, but by the late

-
1. S. Taylor and K. Schroeder. *Inside Intuit*. Cambridge, MA: Harvard Business Review Press, September 4, 2003.
 2. "6 Ways Wealth Made This Billionaire an Amazing Human Being." *NextShark*, March 15, 2014. <http://nextshark.com/6-ways-wealth-made-this-billionaire-an-amazing-human-being/>.
 3. "Intuit and Chipsoft to Merge." *The New York Times*, September 2, 1993. <http://www.nytimes.com/1993/09/02/business/intuit-and-chipsoft-to-merge.html>.

1990s consumer behavior started to change: Users began to expect the products they once ran on their systems to instead be delivered as a service online (Software as a Service). Identifying this trend early, Intuit developed Web-enabled versions of many of its products, including Quicken, TurboTax, and QuickBooks.

While Intuit's business shifted toward SaaS, its development practices remained stagnant. The Product and Technology group, led by senior VP and CTO Tayloe Stansbury, was a growing team of individuals who were responsible for defining the infrastructure upon which software solutions would run. The company continued to develop online solutions in this fashion until 2013.

"Then we had a bit of an epiphany," stated Stansbury. "We were doing rather well in several dimensions, but we seemed to have the same types of small but nevertheless painful problems occurring over and over again. The software and the infrastructure folks were glaring at each other across this chasm and then it hit us—the world had changed around us but we really hadn't changed the way we produced our products. We were still building software but we were *selling* services. We really needed to change our approach to recognize that in the new world order, software and hardware are the raw materials consumed to create a service. We needed to rethink our organization, processes, incentives, and project management."

As mentioned previously, great companies and great managers, when presented with significant challenges and opportunities, rise to the occasion. Intuit is still around and thriving, so how did it address this opportunity? This chapter outlines how to take a vision (an element of leadership) and break it down into the concrete tasks necessary for implementation (management). We'll discuss staff teams for success and explore how to coach them across the goal line of a compelling vision, whether to meet the scalability challenges associated with incredible growth or for nearly any other reason.

In this chapter, we define management and outline characteristics of great managers. We also describe the need for continually improving and upgrading one's team, and provide a conceptual framework to accomplish that task. In addition, we illustrate the importance of metrics and provide a tool to tie metrics to the creation of shareholder wealth (the ultimate metric). We end the chapter by emphasizing the need for management to remove obstacles for teams so that they can reach their objectives.

What Is Management?

Merriam-Webster Dictionary defines management as "the conducting or supervising of something" and as the "judicious use of means to accomplish an end." We'll make a minor modification to this definition and add ethics to the mix, ending with "the judicious *and ethical* use of means to accomplish an end." Why ethics? How

can ethics negatively influence scalability? Consider cases where intellectual property is pirated, licenses for third-party products duplicated, and erroneous public statements about the scalability and viability of a platform are made. Each of these cases involves an ethical issue—the commission of theft and lies intended to garner personal gain at the expense of someone else. Ethics plays a role in how we treat our people, how we incent them, and how we accomplish our mission. If we tell an underperformer that he is performing acceptable work, we are behaving unethically; we are cheating both the shareholders and the employee because he deserves to know how he is performing. The way in which a mission is accomplished is every bit as important as the actual accomplishment of that mission.

AKF's Definition of Management

Management is the judicious and ethical use of means to accomplish an end.

As we've previously stated, management and leadership differ in many ways, but both are important. If leadership is a promise, then management is action. If leadership is a destination, then management is the direction. If leadership is inspiration, then management is motivation. Leadership pulls and management pushes. Both are necessary to be successful in maximizing shareholder wealth. Recall the comparison of leadership and management from Chapter 1, revisited in Table 5.1

Management includes the activities of measuring, goal evaluation, and metric creation. It also includes the personnel responsibilities of staffing, personnel evaluation, and team construction (including skills and other characteristics of team members). Finally, management includes all the activities one might typically consider “project management,” such as driving the team to work completion, setting aggressive dates, and so on.

Table 5.1 Example Leadership and Management Activities

Leadership Activities	Management Activities
Vision setting	Goal evaluation
Mission setting	KPI measurement
Goal setting	Project management
Culture creation/culture setting	Performance evaluation
Key performance indicator (KPI) selection	People coaching
Organizational inspiration	People mentoring
Standard setting	Standard evaluation

What Makes a Good Manager?

Leadership and management are so significantly different that it is rare to find someone who is really good at both disciplines. The people who excel at both were likely good at one aspect and worked at becoming good at the other. As with leadership, we view management as a function with several parameters. Some parameters are shared with the leadership function. Being personable and having a great sense of humor, for instance, are useful for leaders and managers, as they can help influence individuals and organizations to get tasks done.

Other parameters are unique to management. The best managers have an eye for detail and are incredibly task and goal oriented. Upon being given a task or goal, they can unpack that task or goal into everything that needs to happen for it to be successful. This activity involves more than actions; it requires communication, organizational structure, appropriate compensation, logistics, and capital to be successful. Very often, this detail orientation is at odds with the innovative qualities that allow people to generate compelling visions. Sometimes people can adjust themselves to do both, but only with extensive time and effort.

Great managers also develop a variety of people skills that help them get the most out of the individuals within their organizations. The very best managers don't describe themselves as having a specific "style," but rather understand that they might need to employ any number of approaches to motivate certain individuals. Some people respond best to terse descriptions and matter-of-fact approaches, whereas others prefer a bit of nurturing and emotional support. Some people need a taskmaster; still others require a den mother.

Finally, the very best managers recognize the need for continual improvement and realize that the only way to improve things is to measure them. "Measurement, measurement, measurement" is the phrase by which they live. They measure the availability of their systems, the output of their organizations, and the efficiency of everything.

Project and Task Management

Good managers get projects done on time and on budget, and meet the expectations of shareholder value creation in the completion of their projects. Great managers do the same thing even in the face of adversity. Both achieve their goals by decomposing them into concrete supporting tasks. They then enlist appropriate help both inside and outside the organization and continually measure progress. Although this isn't a book on project management, and we won't go into great detail about how to effectively manage scale projects, it is important to understand the necessary actions in those projects to be successful.

Let's return to our Intuit story and see how the CTO, Tayloe Stansbury, and his team broke down the opportunity of changing the business into the component

tasks necessary to be successful. Intuit's epiphany that it was building software but selling services led to a reorganization that moved the internal corporate technology teams under the purview of the CTO. Furthermore, infrastructure individual contributors, such as storage engineers and database administrators, were divided up and allocated to become part of the product teams with which they worked. Product infrastructure professionals were expected to think of themselves as part of the product teams. Each business-oriented product team shared goals including revenue generation, availability, and time to market regardless of whether it was software or hardware focused. Hardware and software teams would now work together. "No more 'you are a customer of mine' mindset," explained Stansbury. "One team, one Intuit, one customer."

The teams further determined that they weren't paying attention to all of the metrics they should be monitoring. A by-product of the "software mindset" was that Intuit wasn't as focused on the availability of its products as many other SaaS companies. Under Stansbury's leadership, this situation quickly changed. The combined teams started a morning service delivery call to jointly discuss the quality of service from the previous day and any open issues across the product lines. Problems associated with incidents were aggressively tracked down and resolved. The teams started reporting on a number of new metrics across the enterprise, including the mean time to resolve incidents, the mean time to repair problems, and the average time to identify incidents in production. Aggressive availability and end-user response time targets were created for each team and shared between the software and infrastructure disciplines. Stansbury's view: "Shared goals are critical to the success of creating a one team, one Intuit, one customer mindset."

Stansbury and his team also felt that Intuit's past organizational structure and approach had left the company with large amounts of technical debt: "When two teams that are necessary to build a product don't collaborate from the beginning in the creation of that product, you can imagine that you don't always get the best designs." Stansbury believed that certain elements of the holistic product were overbuilt, whereas others were underbuilt relative to the individual product needs: "We needed to move our holistic product architecture to more purpose-built designs—designs wherein the software and infrastructure leverage the other's strengths and mitigate the other's weaknesses." In turn, Stansbury and his team planned and budgeted to redesign each product group of services, beginning with the tax division. Incorporating elements of fault-isolative architectures, products would be capable of operating independently of each other and in so doing incorporate software and hardware specific to their needs and desired outcomes. Precise project plans were created and daily meetings held to identify barriers to success within the plans.

While not an all-inclusive list, Intuit's activities are indicative of what it takes to be successful in managing teams toward a vision. Stansbury and his team identified the gap in approach necessary to be successful and broke it down into its component

parts. Teams weren't working as well together as they could, so they reorganized to create cross-functional, product-oriented teams. This endeavor reflects the management aspect of executing the organizational structures described in Chapter 3, Designing Organizations. Collectively, Intuit's teams weren't as focused on availability as they could be, so the company created metrics and structure around availability, including a daily review of performance. Lastly, the combined teams identified architectural improvements around service offerings; in turn, they created structure, budget, and project plans and assigned owners to ensure the successful delivery of those new solutions.

Intuit's service availability increased significantly as a result of its focus on purpose-built designs, implementation of aggressive goals with respect to availability, and rigorous problem and incident management. None of this, however, would have been possible without multiple levels of management focused on decomposing goals into their component parts and the organizational realignment of teams to service offerings.

Here we will transition briefly to a discussion of one of the most often-overlooked aspects of project management—a focus on contingencies. Helmut Von Moltke was a Prussian general famous for several versions of the phrase, “No plan survives contact with the enemy.” Nowhere is this phrase more correct than in the management of complex product development. Myriad things can go wrong in the delivery of complex products, including delays in equipment delivery, problems with the interaction of complex components and software, and critical staff members falling ill or leaving in the middle of a project. In our entire careers, we have never seen a project run exactly as initially envisioned within the first iteration of a project plan. The initial delivery dates may have been met, but the path taken toward those dates was never the original path laid out with such optimism by the planners of the project.

This experience has taught us that while project planning is important, the value it creates lies not in the initial plan but rather in the exercise of thinking through the project and the possible paths. Unfortunately, too many of our clients see the original plan as the only path to success, rather than as just one of many possible paths. Moltke’s quote warns us away from this focus on initial plan execution and moves us toward recognition of the value of contingency planning. Instead of adopting a laser-like focus on the precision of our plan, we should instead consider how we might take any one of a number of paths to project success. As such, we at AKF practice what we call the “5-95 Rule”: Spend 5% of your time developing an adequate, defensible, and detailed plan—but also recognize that this plan will not survive contact with the enemy and devote the remaining 95% of your time to “war gaming” the plan to come up with contingencies. What will you do when network equipment does not arrive on time for your data center build-out? How will you handle the case in which your most critical resource calls in sick during a launch

date? What happens when your data access object hasn't included an important resource necessary for correct execution of an application?

The real value in project plans lies in the cerebral exercise of understanding your options in execution on the path to shareholder value creation. Remember the AKF 5-95 Rule and spend 5% of your time planning and 95% of your time addressing contingencies.

The AKF 5-95 Rule

Most teams spend too much time developing an initial plan, and too little time planning for contingencies and "war gaming" the plan. Ask yourself how often your first plan has remained unchanged throughout the life of an initiative. For us, the answer is "seldom to never."

The 5-95 Rule was born from Helmut Von Moltke's perceptive comment that "No plan survives contact with the enemy."

- Spend 5% of your planning time building a good, defensible plan.
- Spend 95% of your time planning for contingencies when things don't go the way you expect.
- Don't increase your total time spent planning, but rather reallocate your time according to the 5-95 Rule.

Reallocating planning time to the AKF 5-95 Rule will help you be better prepared for the realities of product development.

Building Teams: A Sports Analogy

Professional football team coaches know that having the right team to accomplish the mission is critical to reaching the Super Bowl in any given season. Furthermore, they understand that the right team today might not be the right team for next season: Rookie players enter the sport stronger and faster than ever before; offensive strategies and needs change; injuries plague certain players; and salary caps create downward pressure on the total value of talent that can exist within any team in any year.

Managing team skill sets and experience levels in professional sports is a dynamic job requiring the continual upgrading of talent, movement of personnel, management of depth and bench strength, selection of team captains, recruiting new talent, and coaching for skills.

Imagine a coach or general manager who is faced with the difficult task of recruiting a new player to fill a specific weakness in the team and whose collective team

salaries are already at or near the designated salary cap. The choices in this case are to release an existing player, renegotiate one or more players' contracts to make room for the new player's salary, or not hire the necessary player for the critical position. What do you think would happen to the coach who decides to take no action and not hire the new player? If the team owners find out, they are likely to fire the coach. If they don't find out sooner or later, the team is likely to atrophy and consistently turn out substandard seasons, resulting in lower ticket sales and unhappy shareholders (owners).

Our jobs as managers and executives are really no different than the jobs of coaches of professional football teams. Our salary caps are the budgets that are developed by the executive management team and are reviewed and approved by our boards of directors. To ensure that work is completed cost-effectively with the highest possible levels of throughput and quality, we must constantly look for the best talent available at a price that we can afford. Yet most of us don't actively manage the skills, people, and composition of our teams, ultimately short-changing our company and our shareholders. Scalability in professional sports means scaling the output of individuals. Professional football, for instance, will not allow teams to add a twelfth player to the on-field personnel. In your organization, scaling individuals might mean the same thing. The output of your organization depends on both individual output and the team size. Efficiency—a component of cost-effective scaling—is a measurement of getting more for the same amount of money or (better yet) more for less money. Scaling with people, then, is a function both of the individual people, the number of people, and the organization of people.

Consider a coach who refused to spend time improving the team's players. Can you imagine such a coach keeping his or her job? Similarly, can you imagine walking into your next board of directors meeting and stating that part of your job is *not* to grow and maintain the best team possible? Think about that last point for a minute. In Chapter 4, Leadership 101, we made the case that everything you do needs to be focused on creating shareholder value. Here, we have just identified a test to help you know when you are not creating shareholder value. With any major action that you take, would you be prepared to present it to the board of directors as something that *must* be done? Remember that a decision to *not do* something is the same as deciding to *do* something. Furthermore, ignoring something that should be done is a decision not to do it. If you have not spent time with the members of your team for weeks on end, you have *decided* not to spend time with them. That course of action is absolutely inexcusable and not something that you would likely feel comfortable discussing with your board of directors.

The parallels in professional sports to the responsibilities of team building for corporate executives are clear but all too commonly ignored. To get our jobs done, we must have the best talent (i.e., the best people) possible given our board-authorized

budgets. We must constantly evaluate and coach our team to ensure that each member is adding value appropriate to his or her level of compensation, find new and higher-performing talent, and coach the great people we have to achieve even higher levels of performance.

Upgrading Teams: A Garden Analogy

Even a novice gardener knows that gardening is about more than just raking some soil, throwing some seeds, and praying for rain. Unfortunately, rake, throw, and pray is exactly what most managers do with their teams. Our team is a garden, and our garden expects more of us than having manure thrown on it from time to time. Just as importantly, the scalability of our organization (as we described in our sports metaphor) is largely tied to how great our talent is on a per-person basis and how consistent their behaviors are with our corporate culture.

Gardens should be thoughtfully designed, and the same is true of our teams. Designing our teams means finding the right talent that matches the needs of the organization's vision and mission. Before planting new seeds or inserting new seedlings in our garden, we evaluate how the different plants and flowers will interact. We should do the same with our teams. Will certain team members steal too many nutrients? Will the soil (our culture) properly support their needs? Should the garden be full of only bright and brilliant flowers or will it be more pleasing with robust and healthy foliage to support the flowers?

Managers in hyper-growth companies often spend a lot of time interviewing and selecting candidates, but usually very little time on a per-candidate basis. Even worse, these managers often don't take the time to determine where they've gone wrong with past hiring decisions and what they've done well in certain decisions. Finding the right individual for a particular job requires paying attention to and correcting past failures and repeating past hiring successes. All too often, though, we interview for skills but overlook critical items like cultural or team fit. Ask yourself these questions: Why have you had to remove people? Why have people decided to leave?

Candidate selection also requires paying attention to the needs of the organization from a productivity and quality perspective. Do you need another engineer or product manager, or do inefficiencies in the company's operations indicate the need for additional process definition, tools engineers, or quality assurance personnel?

All too often, we try to make hiring decisions after we've spent a mere 30 to 60 minutes with a candidate. We encourage you to spend as much time as possible with the candidate and try to make a good hire the first time. Seek help in interviewing by adding people whom you trust and who have great interviewing skills. Call previous

managers and peers of the candidates, and be mindful to ask and prod for weaknesses of individuals in your background checks. Pay attention to more than just the skills—determine whether you and your team will like spending a lot of time with the individual. Interview the person to make certain that he or she will be a good fit with your culture and that the individual's behaviors are consistent with the behavioral expectations of the company.

The Cultural Interview

One of the most commonly overlooked components of any interview is ensuring that the candidate is a cultural and behavioral fit for the company. We recommend picking up a book or taking a class on behavioral interviewing to improve your understanding of this issue. In the meantime, here are some things that you can do in your next interview to find the right cultural and behavior fit for your company:

- Make a list of your company's beliefs regarding people. They may be printed on the back of your identification badge or posted on your intranet. Identify questions around these beliefs and distribute them to interview team members.
- Identify interviewers who are both high performers within your team and a good match with the culture, beliefs, and behaviors of your company (or the behaviors to which your company aspires).
- Gather all interviewers after the interview and discuss the responses to the questions and the feelings of the team.

It is as important to make the right cultural hire as it is to hire the right talent and experience. Can you spend 9 to 12 hours each day with this person? Can the other team members do the same? Can you learn from this person? Will the candidate allow the team to teach him or her? For more advice on hiring and interviewing, pick up one of Geoff Smart's books, such as *Who: The A Method for Hiring*.

Feeding your garden means spending time growing your team. Of all the practices involved in tending to your team, this is the one that is most often overlooked for lack of time. We might spend time picking new flowers (though not enough on a per-flower basis), but we often forget about the existing flowers needing nourishment within our garden.

The intent of feeding is to grow the members of your team who are producing to the expectations of your shareholders. Feeding consists of coaching, praising, correcting technique or approach, adjusting compensation and equity, and anything else that creates a stronger and more productive employee.

Feeding your garden also means taking individuals who might not be performing well in a certain position and putting them into positions where they can thrive. However, if you find yourself moving an employee more than once, it is likely that you are avoiding the appropriate action of weeding.

Finally, feeding your garden means raising the bar on the team overall and helping employees achieve greater levels of success. Great teams enjoy aggressive but achievable challenges, and it is your job as a manager to challenge them to be the best they can be.

Although you should invest as much as possible in seeding and feeding, we all know that underperforming and nonperforming individuals choke team productivity, just as surely as weeds steal vital nutrients from the flowers within your garden. The nutrients in this case are the time that you spend attempting to coach underperforming individuals to an acceptable performance level and the time your team spends compensating for an underperforming individual's poor results. Weeding our gardens is often the most painful activity for most managers and executives, and as a result it is often the one to which we tend last.

Although you must abide by your company's practices regarding the removal of people who are not performing (legal requirements vary by country and state), it is vital that you find ways to quickly remove personnel who are keeping you and the rest of your team from achieving your objectives. The sooner you remove the poor performers, the sooner you can find appropriate replacements and get your team moving forward.

When considering performance as a reason for termination, you should always include an evaluation of the person's behaviors. Sometimes one individual within an organization may create more and get more done than any other team member, yet exhibit actions and behaviors that bring down total team output. This is typically fairly obvious when an employee is creating a hostile work environment, but less so when an employee simply does not work well with others. For instance, an employee might get a lot done, but in such a manner that absolutely no one wants to work with him. The result might be that you spend a great deal of time soothing hurt feelings or finding out how to assign the employee tasks that do not require teamwork. If the employee's actions limit the output of the team, then ultimately they limit scalability, and you should act immediately to rectify the situation.

To assess these kinds of tradeoffs, it's often useful to use the concept of a two-dimensional axis with defined actions, such as that depicted in Figure 5.1. The *x*-axis here is the behavior of the employee and the *y*-axis is the employee's performance. Many employee reviews, when done properly, identify the actions on the *y*-axis, but they may not consider the impact of the behavioral *x*-axis. The idea here is that the employees you want to keep are in the upper-right portion of the graph. Those who should be immediately "weeded" are in the bottom-left portion of the graph. You should coach those individuals in the upper-left and lower-right portions

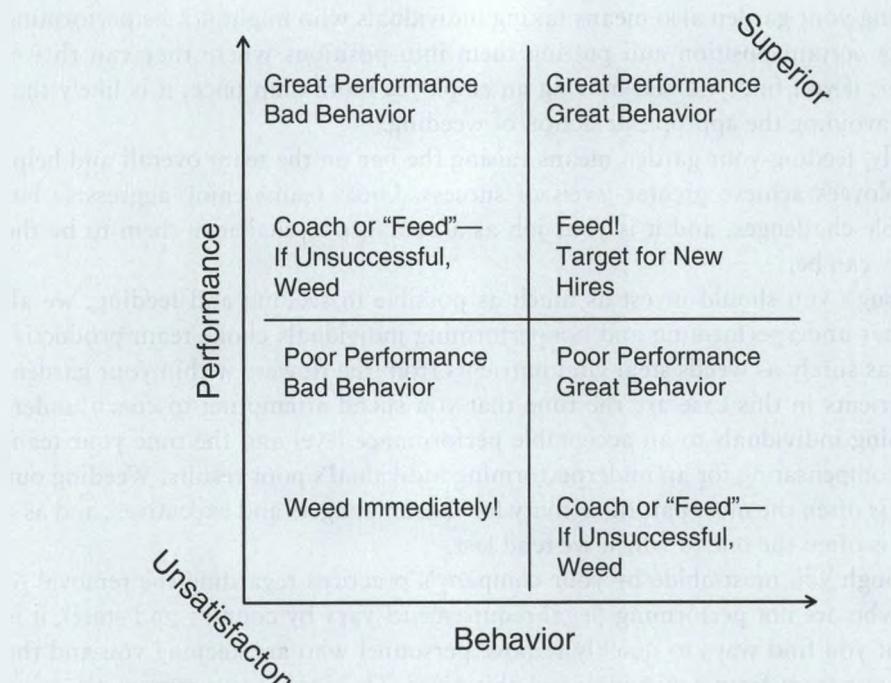


Figure 5.1 Evaluating Behaviors and Performance

of the graph, but be prepared to weed them if they do not respond to coaching. And of course, you want all of your seeds or new employees to be targeted in the upper-right portion of the graph.

One thing that we have learned over time is that you will always wish you had acted earlier in removing underperformers. There are a number of reasons why you can't act quickly enough, including company travel, competing requests, meetings, and so on. Don't waste time agonizing over whether you are acting too quickly—that never happens. You will always wish you had acted even sooner when you have completed the termination.

Seed, Feed, and Weed to Succeed

To continually upgrade or improve our team's performance, we need to perpetually perform three individual activities:

- *Seeding* is the addition of new and better talent to the organization.
- *Feeding* is the development of the people within the organization we want to retain.
- *Weeding* is the removal of underperforming individuals within the organization.

As managers, we often spend too little time interviewing and selecting our new employees, devote too little time to developing and coaching our high performers, and act too late to remove employees who do not display behaviors consistent with our culture or have the drive and motivation to create shareholder wealth.

Measurement, Metrics, and Goal Evaluation

We're not certain who first said it, but one of our favorite sayings is "You can't improve that which you do not measure." Amazingly, we've found ourselves in a number of arguments regarding this statement. These arguments range from "Measurement is too expensive" to "I know intuitively whether I've improved something." You can get away with both of these statements if you are the only shareholder of your company, although we would still argue that your results will be suboptimal. If you happen to be a manager in a company with external shareholders, however, you must be able to *prove* that you are creating shareholder value, and the only way to do so is through measurement.

We believe in creating cultures that support measurement of nearly everything related to the creation of shareholder value. With respect to scale, however, we believe in bundling our measurements thematically. The themes we most often recommend to track scale are cost, availability/response times, engineering productivity and efficiency, and quality.

As we've previously indicated, cost has a direct impact on the scalability of your platform. You undoubtedly will either be given or have helped develop a budget for the company's engineering initiatives. Ideally, a portion of that budget in a growth company will be dedicated to the scalability of the organization's platform or services. This percentage alone is an interesting value to monitor over time, as we would expect good managers to be able to reduce the cost of scaling their platforms over time—potentially as measured by the cost per transaction. Let's assume that you inherit a platform with scalability problems that manifest themselves as availability issues. You might decide that you need to spend 30% to 50% of your engineering time and a significant amount of capital to fix a majority of these issues in the first 2 to 24 months of your job. However, something is wrong if you can't slowly start giving more time back to the business for business initiatives (customer features) over time. Consequently, we recommend measuring the *cost of scale* as both a percentage of total engineering spending and a cost per transaction.

Cost of scale as a percentage of engineering time should decrease over time. Of course, it's easy to "game" this number. Suppose that in year 1 you have a team of 20 engineers and dedicate 10 to scalability initiatives; in this year, you are spending

50% of your engineering headcount-related budget on scalability. Next suppose that in year 2 you hire 10 more engineers but dedicate only the original 10 to scale issues; you are now spending only 33% of your budget on scalability. Although the percentages would suggest that you've reduced the cost of scale, you've really kept it constant, which could argue for measuring and reporting on the relative and absolute costs of scale.

Rather than reporting the absolute cost of scale (e.g., 10 engineers, \$1.2 million per annum), we often recommend normalizing this value based on the activities that create shareholder value. If your organization is a Software as a Service platform (SaaS) provider and makes money on a per-transaction basis, either through advertising or by charging transaction fees, this normalization might be accomplished by reporting the cost of scale on a per-transaction basis. For instance, if your company completes 1.2 million transactions each year and spends \$1.2 million in headcount on scale initiatives, its cost of scale would be \$1/transaction. Ouch! That's really painful if you don't make at least a dollar per transaction!

Availability is another obvious choice when figuring out what to measure. If a primary goal of your scalability initiatives is eliminating downtime, you must measure availability and report on how much of your organization's downtime is associated with scalability problems within its platforms or systems. The intent here is to eliminate lost opportunity associated with users not being able to complete their transactions. In the Internet world, this factor most often has a real impact on revenue; in contrast, in the back-office information technology world, it might result in a greater cost of operations because people are required to work overtime to complete jobs when systems become available again.

Closely related to availability is response time. In most systems, increasing response times often escalate to *brownouts*, which are then followed by *blackouts* or downtime for the system. Brownouts are typically caused by systems performing so slowly that most users abandon their efforts, whereas blackouts are a result of a system that completely fails under high demand. The measurement of response times should be compared to the absolute service level agreement (SLA), even if the agreement remains unpublished. Ideally, the measurement should be performed using actual end-user transactions rather than proxies for their interactions. In addition to the absolute measurement against internal or external service levels, relative measurement against past-month values should be tracked over time for critical transactions. This data can later be used to justify scalability projects if a slowing of any given critical transaction proves to be tightly correlated with revenues associated with that transaction, abandon rates, and so on.

Engineering productivity and efficiency is another important measurement. Consider an organization that measures and improves the productivity of its engineers over time versus one that has no such measurements. You would expect that the former will

start to produce more products and complete more initiatives at an equivalent cost to the latter, or that it will start to produce the same products at a lower cost. Both of these outcomes will help us in our scalability initiatives: If we produce more, by allocating an equivalent percentage of our engineering team, we can get more done more quickly and thereby reduce future scale demands on the engineering team. Also, if we can produce the same number of products at lower cost, we will increase shareholder value, because the net decrease in cost structure to produce a scalable platform means greater profitability for the company.

The real trick in figuring out how to measure engineering productivity and efficiency is to split it up into at least two components. The first part addresses whether your engineering teams are using as much of the available engineering days as possible for engineering-related tasks. To measure this component, assume that an engineer is available for 200 days per year minus your company's sick time, vacation time, training time, and so on. Perhaps your company allows 15 days of paid time off per year and expects engineers to be in 10 days of training per year, resulting in 175 engineering days/engineer. This becomes the denominator within our equation. Then, subtract from this denominator all of the hours and days spent "blocked" on issues related to unavailable build environments, nonworking test environments, broken tools or build environments, missing source code or documentation, and so on. If you haven't measured such value destroyers in the past, you might be astonished to discover that you are making use of only 60% to 70% of your engineering days.

The second part of engineering productivity and efficiency focuses on measuring how much you get out of each of your engineering days. This is a much harder exercise, as it requires you to choose among a set of unattractive options. These options range from measuring thousands of lines of code (KLOC) produced by an engineer, to stories, function points, or use cases produced. All of these options are unattractive because they all have "failures" within their implementation. For instance, you might produce 100 lines of code per engineer per day, but what if you really need to write only 10 lines to get the same job done? Function points, by comparison, are difficult and costly to calculate. Stories and use cases are largely subjective; as such, they all sound like bad options. Of course, an even worse option is to not measure this area at all. Training programs, after all, are intended to help increase individual output, and without some sort of measurement of their effectiveness, there is no way to prove to a shareholder that the money spent on training was well spent.

Quality rounds out our scalability management measurement suite. Quality can have a positive or negative impact on many other measurements. Poor product quality can cause scalability issues in the production environment and, as a result, can increase downtime and decrease availability. Poor product quality increases cost and reduces productivity and efficiency because rework is needed to meet the appropriate scalability needs. Although you obviously need to look at such typical metrics as

bugs, KLOC in production and KLOC per release, absolute bug numbers for your entire product, and the cost of your product quality initiatives, we also recommend further breaking these elements out into the issues that affect scale. For example, how many defects cause scalability (response time or availability) problems for your team? How many of these defects do you release per major or minor release of your code, and how are you getting this number to trend downward over time? How many of these defects are caught by quality assurance versus found in production? You can almost certainly think of even more possible subdivisions.

The Goal Tree

One easy way to map organizational goals to company goals is through a *goal tree*. A goal tree takes as its root one or more company or organizational goals and breaks it down into the subordinate goals to achieve that major goal. We'll use a modified goal tree that we created when we were at the advertising technology company Quigo (Figure 5.2). In 2005, the company had a desire to be profitable by the first quarter of 2006. In keeping with this aim, we entered the goal of "Achieve

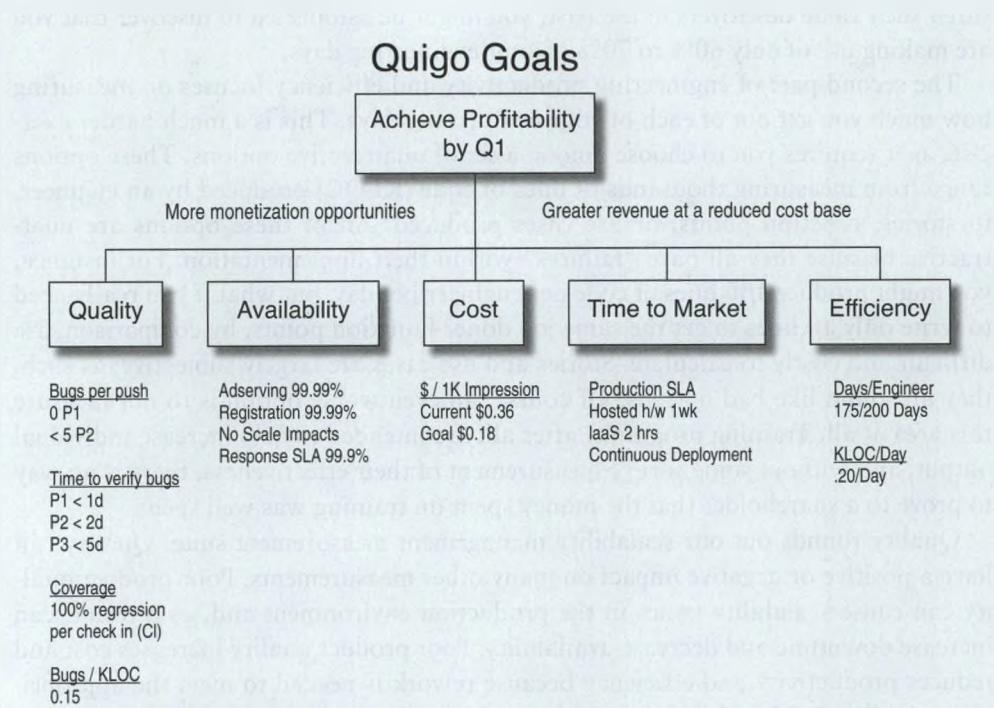


Figure 5.2 Example Goal Tree for Quigo

Profitability by Q1” as the root of our tree. We felt that the two ways the product team could impact profitability were by creating more monetization opportunities and by generating greater revenues at reduced costs.

Quality and availability both affected our opportunity to monetize the product. Unavailability means lost monetization opportunities. Many defects have a similar effect: We can’t monetize an ad placement if a defect forces us to serve the wrong ad or credit the wrong publisher, for example. From an availability perspective, we focused on achieving greater than 99.99% availability from the ad serving and registration systems, having no impacts on availability from scale-related issues, and achieving the desired response-time service levels (less than 1 second for each ad delivered) 99.9% of the time. Our quality metrics covered how quickly we would fix different priority levels of bugs, how many bugs we would release of each type, and how many defects per thousand lines of code we had in our production environment.

From a cost perspective, we focused on reducing the cost per thousand pages delivered by more than 50% (while the focus was real, the numbers given here are completely fictitious—if you have your own advertising solution, you won’t gain any insights here!). We also wanted to reduce time to market (TTM) and decrease our cost of delivery.

Paving the Path for Success

So far, we’ve painted a picture of a manager as being equal parts taskmaster, tactician, gardener, and measurement guru. But a manager’s job isn’t done there. Besides being responsible for ensuring the team is up to the job, deciding which path to take to a goal, and measuring progress, a manager must make sure that the path to that goal is bulldozed and paved. A manager who allows a team to struggle unnecessarily over rough terrain on the way to an objective when he or she can easily pave the way reduces the output of the team. This reduction in output means the team can’t scale efficiently, as less work is applied to the end goal. Less efficiency means lower shareholder return for an investment.

Bulldozed is a rather aggressive term, of course. In this context, we don’t mean to imply that a manager should act as a fullback attempting to lay out a linebacker so that a halfback can make a touchdown. Although that type of aggressive play might be required from time to time, employing it will likely damage your reputation as a desirable and cooperative colleague. Additionally, such behavior may be absolutely unacceptable in some cultures. Instead, we mean that managers are responsible for removing obstacles in the road to the success of an organization and its objectives.

It is very easy for people to confuse this idea with the notion that “anything that stands in my way is an obstacle to my success and should be removed.” Sometimes

the obstacles in your path serve to ensure that you are performing the correct functions. For instance, if you need to release something to your production environment, you might see the quality assurance organization as an obstacle. This observation is at odds with our definition of an obstacle, because the QA organization exists to help you ensure that you are meeting the shareholder's needs for a higher-quality product. The obstacle in this case is actually you and your flawed perception.

Real obstacles are issues that arise and that the team is not equipped to handle. Examples might be a failure of a partner to deliver software or hardware in a time frame consistent with your needs or difficulties in getting testing support or capital freed up for a project. When you encounter these kinds of obstacles, remember that the team isn't working *for* you but rather *with* you. You may be the captain of the team, but you are still a critical part of its success. Great managers actually get their hands dirty and "help" the team accomplish its goals.

Conclusion

Management is about execution and all of the activities necessary to realize the organization's goals, objectives, and vision, while adhering to its mission. It should be thought of as a "judicious and ethical use of means to accomplish an end." Succeeding at management, like thriving as a leader, requires a focus and commitment to learning and growth. It also necessitates respect for and caretaking of tasks, people, and measurements to accomplish the desired goals.

Project and task management is essential to successful management. This effort includes the disaggregation of goals into their associated projects and tasks, the assignment of individuals and organizations to those tasks, the measurement of progress, the communication of status, and the resolution of problematic issues. In larger projects, it will include the relationships between tasks, so as to determine which tasks should happen when and to establish timelines.

People management has to do with the composition and development of organizations and the hiring, firing, and development of individuals. We often spend too much time with our underperformers and wait too long to eliminate them from the team. As a consequence, we don't spend enough time growing the folks who are truly adding value. In general, we need to devote more time to giving timely feedback to the individuals on our team to ensure that they have ample opportunities to contribute to shareholder value. We also spend a great deal of time interviewing new candidates in total, but often not enough on a per-candidate basis. All too often, our new team members have spent only 30 minutes to an hour with six to seven people before a hiring decision is made. Set aside enough time with people to be sure that you will be comfortable with welcoming them into your family.

Measurement is critical to management success. Without measurements and metrics, we cannot hope to improve, and if there is no hope for improvement, why employ people as managers? The “Measurement, Metrics, and Goal Evaluation” section provides a number of measurement suggestions, and we highly recommend a review of these from time to time as you develop your scalability program.

Managers need to help their team complete its tasks. This means preventing issues from occurring when possible and ensuring that any obstacles that do appear are resolved in a timely fashion. Good managers work to immediately remove barriers, and great managers keep them from arising in the first place.

Key Points

- Management is the judicious and ethical use of means to accomplish an end.
- As with leadership, the pursuit of management excellence is a lifelong goal and as much a journey as it is a definition.
- Like leadership, management can be viewed as a function consisting of personal characteristics, skills, experiences, actions, and approaches. Increasing any aspect of these elements increases your management “quotient.”
- Forget about “management style” and instead learn to adapt to the needs of your team, your company, and your mission. Project and task management is critical to successful management. These activities require the ability to decompose a goal into component parts, determine the relationships among those parts, assign ownership with dates, and measure progress based on those dates.
- Spend 5% of your project management time creating detailed project plans and 95% of your time developing contingencies to those plans. Focus on achieving results in an appropriate time frame, rather than laboring to fit activities to the initial plan.
- People and organization management is broken into “seeding, feeding, and weeding.”
 - Seeding is the hiring of people into an organization with the goal of getting better and better people. Most managers spend too little time on the interview process and don’t aim high enough. Cultural and behavioral interviewing should be included when looking to seed new employees.
 - Feeding is the development of people within an organization. We can never spend enough time giving quality feedback to our employees.
 - Weeding is the elimination of underperforming people within an organization. It is rarely done “soon enough,” although we should always feel obligated to give someone performance-related feedback first.

- We can't improve what we don't measure. Scalability measurements should include measurements of availability, response time, engineering productivity and efficiency, cost, and quality.
- Goal trees are effective for mapping organizational goals to company goals and help create a "causal roadmap to success."
- Managers are responsible for paving the path to success. The most successful managers see themselves as critical parts of their teams working toward a common goal.

Chapter 6

Relationships, Mindset, and the Business Case

In war, the general receives his command from the sovereign.

—Sun Tzu

Thus far in Part I, “Staffing a Scalable Organization,” we have discussed the importance of getting the right people into the right roles, exercising great leadership and management, and establishing the right organization size and structure. Now we turn our attention to three additional pieces of the scalability puzzle:

- Building effective business leader/product team relationships
- Having the right “mindset” to build products
- Making the business case for product scalability improvements

Understanding the Experiential Chasm

We believe that a great many of the problems existing between many general managers (the people responsible for running a business) and their technical teams are a result of a vast and widening chasm in education and experience. From an education perspective, the technology executive may have a math-, science-, or engineering-focused degree, while the general manager likely has a less math-intensive background. Personality styles may also vary significantly, with the technical executive being somewhat introverted (“put him in a dark room alone and he can get anything done”) and the general manager being much more extroverted, friendly, and “salesperson like.” Their work experience likely varies as well, with the general manager having been promoted to this position through closing deals, selling proposals, and making connections. In contrast, the technical executive might have been promoted based on technical acumen or a demonstrated ability to get a product completed on time.

This mismatch in education and experience causes difficulty in communication for several reasons. First, with very little in common, there is often little reason outside of work-specific tasks for the two people to communicate or to have a relationship. They might not enjoy the same activities and might not know the same people. Without this common bond outside of work, the only way to build trust between the two individuals is through mutual success at work. Success may ultimately create a bond that kindles a relationship that can endure through hard times. But when mutual success has not yet been achieved, the spark that occurs kindles the opposite of trust—and without trust, the team is doomed.

Second, without some previous relationship, communication does not happen on mutual footing. Questions are often rightfully asked from a business perspective, and the answers are often given in technical terms. For instance, the general manager might ask, “When can we ship the Maxim 360 Gateway for revenue release?”, to which the technical executive might respond, “We are having problems with the RF modulation and power consumption, and we are not sure if it is a software potentiometer or a hardware rheostat. That said, I do not think we are more than two weeks off of the original delivery schedule to QA.” Although the technical executive here gave a full and complete response, it probably only frustrated the general manager. She quite likely has no idea what a soft-pot, rheostat, or even RF is. The information came so fast and was intermixed with so many important—but to her, meaningless—pieces of information that it just became confusing.

This resulting mismatch in communication quite often gives way to a more detrimental form of communication, which we call *destructive interference*. Questions begin to be asked in a finger-pointing fashion, such as “What are you doing to keep this on track?” or “How did you let it slip a week?”, rather than in a fashion meant to resolve issues early, such as “Let us see if we can work together to determine how we can get the project back on the timeline.” Of course, you should keep and hold high expectations of your management team, but doing so should *not* create a destructive team dynamic. It is possible to both have high standards and actually be a participative, supporting, and helpful leader and executive.

Why the Business Executive Might Be the Problem

Some questions can be asked to determine if the likely culprit of poor communication is the business executive. Do not fret; we will provide a similar set of questions to point the blame at the technology executive. The most likely case is that some amount of fault resides with both business and technology leaders. Acknowledging this fact represents a major step toward fixing the problems and improving communication.

- Has the head of technology been replaced more than once?
- Do different people in the technology team give the business leaders the same explanations but they are still not believed?

- Do business leaders spend as much time attempting to understand technology as they have invested in learning to read financial statements?
- Do the business leaders understand how to ask questions to know whether dates are both aggressive and achievable?
- Does the business executive spend time in the beginning of a product life cycle figuring out how to measure success?
- Do business leaders lead by example or do they point fingers?

Note how many of the preceding questions can be easily traced back to education and experience. For instance, if you are getting consistent answers from your team, perhaps you just do not understand what they are saying. There are two ways to resolve that: Either you can gain a better understanding of what they are telling you, or you can work with them so that they communicate in a language that you better understand. Better yet, do both!

These questions are not meant to imply that the business leaders are the only problem. However, failure to accept at least some culpability is often a warning sign. While perhaps a person (such as the CTO) needs to be “upgraded” to create a better team with better communication, constant churn within the CTO position is a clear indication that the issue is really the hiring manager.

Why the Technology Executive Might Be the Problem

Here is a similar list of questions to ask your technology leadership:

- Does the technology team provide early feedback on the likelihood of making key dates?
- Is that feedback consistently incorrect?
- Is the business repeatedly experiencing the same problems either in production or in product schedules?
- Do the technology team members measure their work against metrics that are meaningful to the business?
- Are the technology choices couched in terms of business benefit rather than technical merit?
- Does the technology team understand what drives the business, who the competitors are, and how their business will be successful?
- Does the technology team understand the business challenges, risks, obstacles, and strategy?
- Can the technology leadership read and understand the balance sheet, statement of cash flows, and income statement?

We often encounter technology executives who simply haven't learned the language of business. The best technology executives define metrics in terms of things that are important to the business: revenue, profit, cost, time to market, barriers to entry, customer retention, and so on. It is critical for the technology executive to understand the means by which the business makes money, the drivers of revenue, the current financial reality within the business, the competitive landscape, and the current year's financial goals for the business. Only with this knowledge can the CTO create the technology strategy necessary to achieve the business's objectives.

It is common for CTOs to be promoted to their positions based on technical acumen. To be successful in that role, however, they must find ways to significantly increase their business acumen. Many accelerated, online, and part-time MBA programs are available that can help the technical executive bridge the gap between the executive team and the technology team. CTOs needing greater business acumen can also seek out executive coaches and add business professional reading to their reading lists, especially in the areas of finance, strategy development, and capital planning/budgeting.

Defeating the IT Mindset

There are two basic models for technology organizations: the IT organization model and the product organization model.

IT organizations often provide consulting and implementation services to other organizations for the purposes of decreasing costs, increasing employee productivity, and so on. They tend to focus on implementing enterprise resource planning (ERP) solutions, communication systems (telecommunications, email, and the like), and customer relationship management (CRM) solutions. Often they work to elicit a set of requirements from internal customers (other employees) to implement packaged or homegrown solutions to meet organizational objectives. IT groups are often viewed as cost centers and are run accordingly. Their primary objectives are to reduce the cost basis of the company by employing technology solutions rather than by expanding headcount to similar activities. In turn, IT organizations may be evaluated based on their costs to the enterprise. This may be done by determining costs as a percentage of revenue, as a percentage of total operating (non-gross margin) costs, or on a cost-per-employee (per capita) basis within the company. Most companies need an IT group, and being in an IT organization is a great way to contribute to a company and to help generate shareholder value.

The second model that technology organizations can use to structure themselves is the product model. Product organizations define and create the products that drive growth and shareholder value creation. Because their mission is different from

that of IT organizations, product teams function, act, and think differently than their peers in IT. Rather than primarily being evaluated on a cost basis (e.g., cost per employee, cost relative to operating costs), product teams are viewed as profit centers. This is not to say that the cost of the organization is irrelevant, but rather that it is relevant only in evaluating the revenue and profit that the cost delivers. Whereas IT teams often see their customers as being other employees, product teams are focused on external market analysis, and they see their customers as being the users of the products created by the company.

The difference in mindset, organization, metrics, and approach between the IT and product models is vast. To better understand the differences between these two approaches, let's employ an analogy. Suppose you are the offensive coordinator of an NFL team. If you were to employ the IT mindset for your offensive squad, you would tell your offensive line that their customer is the quarterback and running backs. In this analogy, the offensive line provides "blocking services" for the running backs and the quarterback. The problem here is that customers and vendors, by definition, aren't part of the same team. They have different goals and objectives. Customers want value creation at a fair price, while vendors want to maximize shareholder value by selling their products and services at the highest price that attracts the number of customers needed to maximize revenue. While both can succeed, they don't share common objectives.

Football teams, however, understand that they must succeed as a unit. Members of the offensive squad understand that their job is to jointly throw up as many points as possible while committing as close to zero turnovers as possible. When they work well, they work as a team—not as one component of the group providing services to another component. While the offensive squad is subject to the salary cap set for the football team, their primary measures of success are points, turnovers, and ultimately wins. Their customer is the fan, not their teammates.

Successful companies such as eBay, PayPal, Amazon, and Google understand in their DNA that technology is part of their product and that their product is their business. These companies do not attempt to use IT metrics or IT approaches to develop their products. The product and executive teams understand that the customer is the outside person purchasing their solutions, not someone within their own company. They think in terms of revenue growth and the cost necessary to generate that revenue, not in terms of the cost per employee within the company. In such organizations, the business leaders use terms like "product architecture" rather than "enterprise architecture." They seek to isolate faults within the products they create rather than looking for how to share resources (and failures) for the purposes of reducing costs.

Real problems emerge when a company creates a product for real customers but does so using processes meant for internal IT organizations. The documentation

of requirements built into this approach tends to kill the notion of product discovery. Corporate technology governance tends to significantly slow time to market for critical projects. Top-down innovation from internal customers eliminates the advantages of broad-spectrum sourcing of innovation we described in Chapter 3, Designing Organizations (refer to Figure 3.5). Companies continue to seek internal solutions rather than sensing their market and reacting to how customers interact with their products. The bottom line is that IT mindsets are great for internal technology development, but disastrous for external product development.

Great products are built by companies that understand their customers. Great product teams innovate with their customers directly, rather than eliciting requirements from other employees. Great product companies understand the difference between running teams that create externally focused products and those that create internally focused efficiencies.

The Business Case for Scale

So far in this chapter, we have covered why a communication breakdown might occur between the technology leaders and business leaders as well as how to solve that problem. We also explored why the traditional IT mindset and approaches are inappropriate for the development of products. Armed with an understanding of these common problems and ideas about how to fix them, we now turn our attention to selling the need to invest in scale and availability.

We'll start by explaining how *not* to sell any initiative. Business is about revenues and costs, profits and losses. A surefire way to guarantee that a project gets no funding is to sell the project on something other than revenue and cost. Put another way, no one cares that the project upon which you want to embark is the newest and greatest technology available. Your job is to explain how and why it will maximize shareholder value by maximizing revenue and minimizing costs. Your business is unique, so each business case (each "sales package") will, in turn, need to be tailored for your platform or product.

By way of example, let's take a pass at how we might frame the business case for investing in higher availability. Downtime (the opposite of being available) usually means lost revenue and/or unhappy customers. For sites with transaction-based revenue models, simply take the projected revenue for the quarter or month and calculate revenue generated per minute. While even the simplest sites are likely to experience wild fluctuations in the actual revenue per minute (the peak usage can be 100 or more times the overnight usage of an ecommerce site), this straight-line calculation is a good start. For products with recurring license fees, calculate (based on contractual remuneration) what each minute costs your organization in terms of customer credits or

remunerations. For products monetized through advertising (as in ad impressions), use the ad revenue per minute calculation or identify the number of lost impressions (page views) multiplied by the per-page value. These calculations result in the revenue lost associated with any downtime on a per-minute or per-page basis.

For a more accurate example of downtime cost calculation, you can use a graph of your site's normal daily and weekly traffic. Take last week's traffic graph (assuming last week was a normal week), and lay over top of it the traffic graph depicting the outage. The amount of area between the two lines should be considered the percentage of downtime. This method is particularly useful for partial outages, which you should have if you follow our advice in Chapter 21, Creating Fault-Isolative Architectural Structures.

In Figure 6.1, assume that the solid line is your revenue per 10-minute interval for the prior week on Monday. The dotted line is the revenue for the Monday on which you had an intermittent service problem from roughly 16:30 hours to 18:20 hours.

This graph serves multiple purposes. First, if the data is plotted in real time, the graph helps identify when things aren't working as we expect. Our experience in high-volume sites indicates that these graphs tend to follow fairly predictive patterns. While you may increase your traffic or revenue (as the dotted is slightly higher than the solid line in Figure 6.1), the first and second derivatives of the curve (the rise over run and change in rise over run) are likely to display weekly or daily patterns.

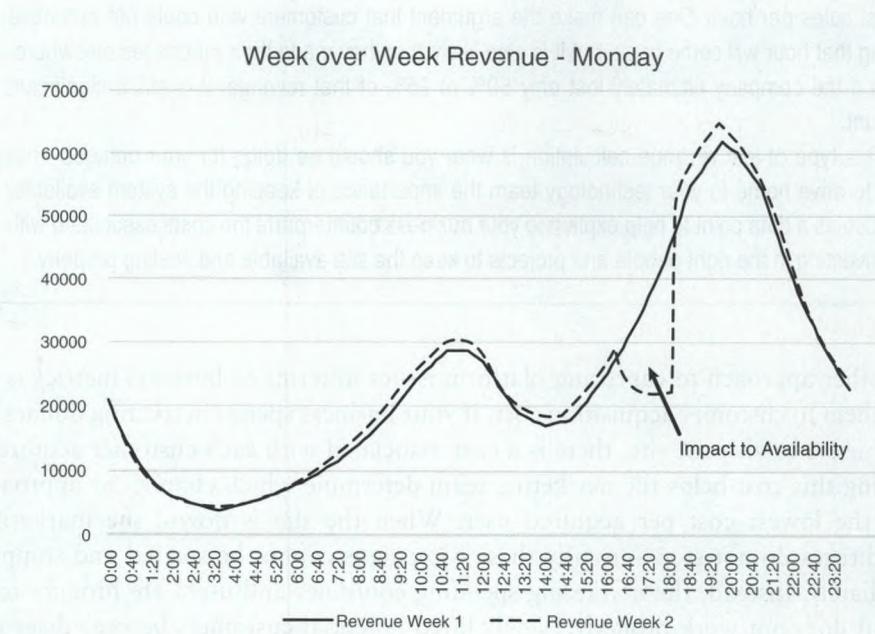


Figure 6.1 Availability Graph

Put another way, the majority of the dotted line displays the same wave pattern as the solid-line curve. That is, the two are consistent until an event occurs at roughly 16:30 hours. If a team were watching this curve, it might give them an early indication that something is happening—something to which they should respond.

The second purpose of this graph is to measure the impact of an outage in transactions or revenue. As previously indicated, the area between these curves is a great approximation of the impact of this particular outage. In this case, because we are calculating revenue in the graph, we can identify exactly how much revenue we lost by looking at the area between the two lines.

Amazon Outage

As an extreme example of how downtime translates into revenue dollars, let's consider a service such as Amazon and see what its downtime costs are. Now, we do not mean to single Amazon out in any negative way, because it typically has great uptime and almost any other large Internet service has seen equal or more downtime. But Amazon does make a great case study because it is so large and is a public company (NASD: AMZN).

According to the *New York Times* technology blog, "Bits," on June 6, 2008, Amazon experienced an outage of its site lasting more than one hour. Using the expected revenue from the second quarter of 2008 of \$4 billion, this calculates as a straight-line projection of \$1.8 million in lost sales per hour. One can make the argument that customers who could not purchase during that hour will come back, but it is also likely that they made their purchases elsewhere. Even if the company ultimately lost only 50% or 25% of that revenue, it is still a significant amount.

This type of lost revenue calculation is what you should be doing for your outages—not only to drive home to your technology team the importance of keeping the system available, but also as a data point to help explain to your business counterparts the costs associated with not investing in the right people and projects to keep the site available and scaling properly.

Another approach to capturing platform issues in terms of business metrics is to relate them to customer acquisition cost. If your business spends marketing dollars to attract users to visit the site, there is a cost associated with each customer acquired. Knowing this cost helps the marketing team determine which channel or approach offers the lowest cost per acquired user. When the site is down, the marketing expenditures do not stop—usually these campaigns cannot be started and stopped immediately. Instead, the marketing spending continues and users are brought to a site that does not work properly. Newly lured potential customers become disgruntled with the experience, and you lose the opportunity to convert them to paying

customers. Each downtime period then becomes a future revenue loss equivalent to the lifetime value of all the users who did not convert during the period of unavailability. Extending beyond this understanding, if you keep track of returning users, another metric to look at is how many users stop returning after an outage. If 35% of your users return to the site once per month, watch this metric after an outage. If their numbers drop, you may have just lost those users permanently.

Another idea for describing the outages or potential for outages in business terms is to translate it into costs within the organization. These costs can be cast in terms of operations staff, engineers, or customer support staff, with the last being the most immediately noticeable by the business. When downtime occurs and engineers and operations staff must attend to remedying this situation, they are not working on other projects such as customer features. A dollar amount can be calculated by determining the total engineering budget (e.g., salaries, support), then associating the number of engineers and time spent on the outage as a percentage of the total budget. As noted previously, the factor closest to the business would be customer support staff who are not able to work due to support tools being unavailable during the outage or who must handle extra customer complaints during the outage and for hours afterward. For companies with large support staffs, this amount of work can add up to significant amounts of money.

Although determining the actual cost of an outage can be a painful exercise for the technology staff, it serves several purposes. First, it quantifies the cost of downtime in real dollar values. This data should be helpful when you are arguing for support and staffing for scalability projects. Second, this calculation helps educate the technology staff about what an unavailable platform really costs the business. Understanding how profitability, bonuses, budgets, hiring plans, and so on are all correlated with platform performance can be a huge motivator to engineers.

Conclusion

In this chapter, we wrapped up Part I of this book by addressing the final link in the scalable organization chain: how to make the business case for hiring resources, allocating resources, and staying focused on scalability as a business initiative. An experiential chasm tends to exist between most technologists and their business counterparts, most likely including the CTO's boss, but a business will benefit when all parties have a corporate mindset. Ideas about how to cross the chasm and undo the corporate mindset so the business can respond to the need to focus on scalability, especially from a people and organizational perspective, include hiring the right people, putting them in the right roles, demonstrating the necessary leadership and management, and building the proper organizational structure around the teams.

Key Points

- An experiential chasm separates technologists and other business leaders due to education and experiences that are missing from most nontechnology executives' careers.
- Technologists must take responsibility for crossing over into the business to bridge this chasm.
- To garner support and to understand scaling, initiatives must be put in terms that the business leaders can understand.
- Calculating the cost of outages and downtime can be an effective method of demonstrating the need for a business culture focused on scalability.

Part II

Building Processes for Scale

Chapter 7: Why Processes Are Critical to Scale	131
Chapter 8: Managing Incidents and Problems	143
Chapter 9: Managing Crises and Escalations	159
Chapter 10: Controlling Change in Production Environments	177
Chapter 11: Determining Headroom for Applications	197
Chapter 12: Establishing Architectural Principles	209
Chapter 13: Joint Architecture Design and Architecture Review Board	225
Chapter 14: Agile Architecture Design	239
Chapter 15: Focus on Core Competencies: Build Versus Buy	249
Chapter 16: Determining Risk	259
Chapter 17: Performance and Stress Testing	273
Chapter 18: Barrier Conditions and Rollback	291
Chapter 19: Fast or Right?	303

Part II

Surging Incense to

Saints

Surging Incense to

Chapters 11-12

Surging Incense to

Chapters 13-14

Surging Incense to

Chapters 15-16

Surging Incense to

Chapters 17-18

Surging Incense to

Chapters 19-20

Chapter 7

Why Processes Are Critical to Scale

After that, comes tactical maneuvering, than which there is nothing more difficult. The difficulty of tactical maneuvering consists in turning the devious into the direct, and misfortune into gain.

—Sun Tzu

Perhaps no one knows the value of the right process at the right time better than NCAA football coach Nick Saban, who currently coaches the Alabama Crimson Tide. He is the first college football coach to win national championships at two different Division 1 schools (Louisiana State University and Alabama). As of the time we were writing this book, he had four national championships. To what does Saban credit his success? Listen to his interviews and you will hear him answer this question the same way over and over: “process focus.” “Good process produces good results” and “process guarantees success.”¹ Saban’s process is to focus not on winning or losing, but rather on the repetitive activities and fundamentals that, if done well, will result in a win. In the weight room, process is the adherence to the strict form of a complex functional lift. On the field, it is focusing on the repetitive mechanics specific to each player’s position. Saban’s process is “the standard that we want everybody to work toward, adhere to and do it on a consistent basis.”²

To scale the output of our engineering resources, we need individuals to work in teams. For teams to be effective in the delivery of products, we need processes to help them coordinate, govern, and guide their efforts. We also need processes to help teams learn from and avoid repeating past failures and to help them repeat their

1. Jason Selk. “What Nick Saban Knows About Success.” *Forbes*, September 12, 2012. <http://www.forbes.com/sites/jasonselk/2012/09/12/what-nick-saban-knows-about-success/>.
2. Greg Bishop. “Saban Is Keen to Explain Process.” *New York Times*, January 1, 2013. http://thequad.blogs.nytimes.com/2013/01/05/saban-is-keen-to-explain-process/?_php=true&_type=blogs&_r=0.

former successes. This section of the book characterizes various essential processes and the roles they should play in your organization. Processes that we will cover in Part II include the following:

- How to properly control and identify change in a production environment
- What to do when things go wrong or when a crisis occurs
- How to design scalability into your products from the beginning
- How to understand and manage risk
- When to build and when to buy
- How to determine the amount of scale in your systems
- When to go forward with a release and when to wait
- When to roll back and how to prepare for that eventuality

Before launching into these topics, we think it is important to discuss in general how processes affect scalability. First we look at the purpose of processes. We then discuss the importance of implementing processes appropriate to the size and maturity of the organization.

The Purpose of Process

As defined by *Wikipedia*, a business process is a “collection of related, structured activities or tasks that produce a specific service or product (serve a particular goal) for a particular customer or customers.” These processes can be directly related to providing a customer with a product or service, such as manufacturing, or they can be supportive processes, such as accounting. The Software Engineering Institute defines process as what holds together three critical dimensions of organizations: people, methods, and tools. In its published Capability Maturity Model for Development version 1.2, the Software Engineering Institute states that processes “allow you to address scalability and provide a way to incorporate knowledge of how to do things better.” Processes allow your teams to react quickly to crises, determine the root cause of failures, determine the capacity of systems, analyze scalability needs, implement scalability projects, and address many more fundamental needs for a scalable system and organization. These activities are vital if you want your system to scale with your organization’s growth. As an example, if you rely on an ad hoc response to restore your service when an outage occurs, you will inevitably experience much more downtime than if you have a clear set of steps that your team should follow to respond, communicate, debug, and restore services.

While part of managers’ job is to help resolve team problems, it isn’t their only responsibility; that is, managers cannot stand around all day waiting for problems

to solve. As an example, suppose an engineer is checking in code to a source repository and is unsure of which branch to use. This situation will likely occur multiple times, so it doesn't seem cost-effective to repeatedly involve a manager to resolve the dilemma. Instead, perhaps the engineering team can decide that bug fixes go into the maintenance branch and new features go into the main branch. To make sure everyone on the team knows this, someone might write up this policy and send it around to the team, post it on the team's wiki, or tell everyone about it at the next all-hands meeting. Congratulations: The team just developed a process! This case illustrates one of the principal purposes of processes—to manage work without needing a manager and to reduce the cost and increase the value of outcomes of repetitive tasks. Good processes supplement management, augment its reach, and ensure consistency of quality outcomes.

Back to our example of an engineer checking in code to the source code repository: What would happen if the engineer did not have a process to reference, could not find her manager, and did not have a process established for dealing with procedural uncertainties? Today, she decides that she should check her bug fix into the maintenance branch. This seems logical because the branch is called "maintenance" and the bug fix is maintaining the application. A few days later (long enough for the engineer to forget about her decision regarding where to check in the bug fix), she has been assigned another bug to fix. She quickly identifies the problem and makes the correction. Now ready to check in the new fix, she has the same question: Which branch? She again looks for her manager and cannot find him; he must have a very clever hiding spot in which to watch his favorite game show. The engineer also cannot seem to remember what she did last time in this situation. She does remember hearing that code is being promoted from the main branch tonight, and it seems logical that the product team and her boss would want this fix in as soon as possible. Consequently, she checks in her bug fix to the main branch and proceeds with her new feature development.

See the problem? Yes, without a clear process, there is room for everyone to make their own decisions about how to accomplish certain tasks. When everyone makes their own decisions, we get variability in results and outcomes. Variability in results and outcomes means, by definition, that we aren't avoiding past mistakes and repeating past successes. Thus, key reasons that we create and maintain processes are to standardize how we perform tasks and to standardize what we do in the event of procedural uncertainty.

In our consulting practice, we are often faced with teams that mistakenly believe that the establishment of processes will stifle creativity. Similarly, process change can be one of the more difficult for organizations to embrace. For example, a change control process initially might be seen as a way to slow down and control when things get released. The reality is quite different: Well-placed processes foster creativity among team members. There is only so much time in each workday and only so many tasks

that your engineers can concentrate on. Using the change control example, the results of the process should help the engineer release things safely more often and spend less time involved with an incident caused by an uncontrolled change. Equally important, people tend to have only a limited amount of creativity within them before they must “recharge their batteries.” If an engineer must spend precious time and some amount of creative thought on menial tasks, we lose that time and creative power, which could have otherwise been spent on the really important tasks, like designing a new user interface. A well-structured environment of processes can eliminate distractions and leave the engineer with more time and energy to focus on being creative.

Having discussed the purpose of processes, let’s turn our attention to determining how much process we should implement. Although processes do help improve organizational throughput and standardize repetitive or unclear tasks, too many processes can stall organizations and increase costs.

CMMI

The origin of the Capability Maturity Model (CMM) in software engineering can be traced back to a military-funded research project at Carnegie Mellon Software Engineering Institute that was seeking a method of evaluating software subcontractors. Founded as a pure software engineering model, many CMMs were later developed as a general assessment of the process capability maturity of many different technology arenas such as systems engineering, information technology, and acquisitions. The propagation of CMMs gave birth to the Capability Maturity Model Integration (CMMI) project, which was intended to create a general CMM framework. This framework supports “constellations,” which are collections of CMMI components. There are currently three constellations: CMMI for Development, CMMI for Services, and CMMI for Acquisitions.

CMMI uses levels to describe an evolutionary path of process improvement described as either a “capability level” for organizations utilizing continuous improvement or a “maturity level” for those using a staged representation. These levels are shown in Figure 7.1 and described in Tables 7.1 and 7.2.

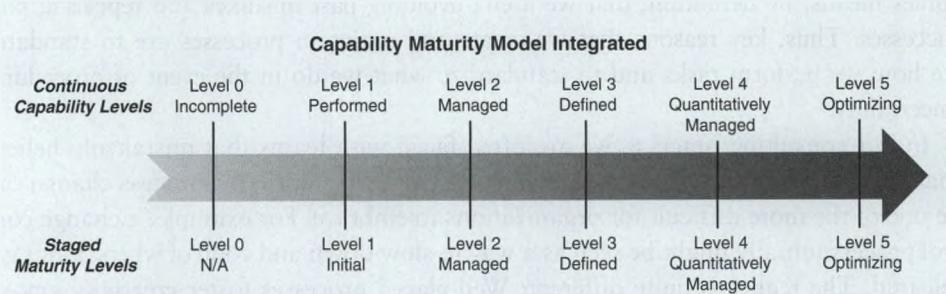


Figure 7.1 CMMI Levels

Table 7.1 Capability Levels

Level Number	Capability Level	Description
0.	Incomplete	When a process is not completed or is only partially completed
1.	Performed	When a process is performed that satisfies the stated goal of the process
2.	Managed	When processes have basic infrastructure to support them and are retained during times of stress
3.	Defined	When the standards and procedures are consistent and described rigorously
4.	Quantitatively Managed	When a process is controlled using statistical or other quantitative techniques
5.	Optimizing	When processes are improved through incremental and innovative improvements

Table 7.2 Maturity Levels

Level Number	Maturity Level	Description
1.	Initial	Processes are characterized as reactive and chaotic.
2.	Managed	Project management has been established and some amount of process discipline exists.
3.	Defined	Processes have been documented and institutionalized.
4.	Quantitatively Managed	Quantitative measurements of processes are used for improvements.
5.	Optimization	Processes are continuously improved based on incremental and innovative advances.

Right Time, Right Process

Organizations are composed of people with varying experiences, backgrounds, and relationships. Just as no two people are exactly the same, so, too, no two organizations are identical. Organizations are also in a constant state of flux. Just as time changes people, so, too, does time change an organization. An organization may

learn (or fail to learn) from past mistakes or learn to repeat certain successes. Members of the organization may increase their skills in a certain area and change their relationships outside of the organization. People join and leave the organization.

If all organizations are different and all organizations are in a constant state of flux, what does this mean for an organization's processes? Each and every process must be evaluated first for general fit within the organization in terms of its rigor or repeatability. Small organizations may need fewer processes, because communication is easier between team members. As organizations grow, however, they may need processes with a larger number of steps and greater rigor to ensure higher levels of repeatability between teams. As an example, when you first founded your company and the staff consisted of you and one other engineer, with very few customers for your products and services, the crisis management process would have simply been that you get out of bed in the middle of the night and reboot the server. If you missed the alert, you would reboot it the morning because there were likely no customers wanting to use your service in the middle of the night. Using that same process when your team consists of 50 engineers and you have thousands of customers would result in customer angst and lost revenue. In such a scenario, you need a process that spells out to everyone the necessary steps to take when a significant incident arises, and that process needs to be consistently repeatable.

A Process Maturity Framework

As a guideline for discussing the rigor or repeatability of a process, we like to refer to the capability and maturity levels from the Capability Maturity Model Integration (CMMI) framework. This section is in no way a full or complete explanation of the CMMI framework, and we are in no way suggesting that you must implement or adopt the CMMI. Instead, we introduce this framework as a way to standardize terminology for process improvement and repeatability. The CMMI levels are an excellent way to express how processes can exist in a number of states, ranging from ill defined to using quantitative information to make improvements. These extreme states are marked in Figure 7.2 with the O and the X points along the gradient

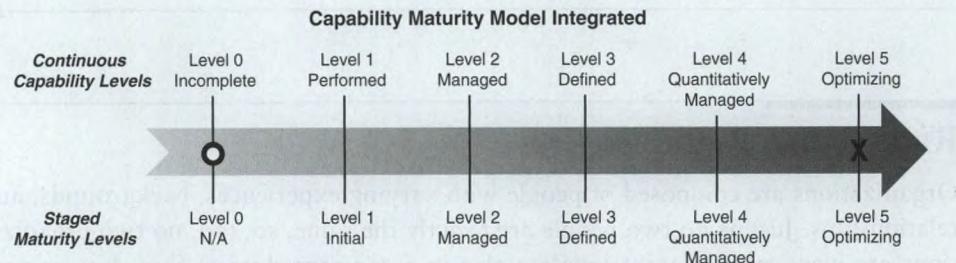


Figure 7.2 Extremes of Possible Process Levels

depicting the capability and maturity levels. For more information on the CMMI, visit the Software Engineering Institute's site at <http://www.sei.cmu.edu/>.

As introduced in the CMMI sidebar, the levels are used to describe an evolutionary path of process improvement described as either a “capability level” for organizations utilizing continuous improvement or a “maturity level” for those using a staged representation. Although it may be ideal to have all level 5 processes in your business, it is unlikely, especially in a startup, that you will have enough resources to focus on establishing, managing, documenting, measuring, and improving processes to accomplish this goal. In fact, such a focus is likely to be at odds with the real value creation in a startup—getting products to market quickly and “discovering” the right product to build. Next, we offer some guidelines to help determine if your processes are meeting your needs.

When to Implement Processes

Recall that processes augment the management of teams and employees; therefore, a sure sign that a process could be effective is if you or your managers are constantly managing people through similar tasks time and time again. Constantly managing people through repetitive tasks is a sign that you can gain from defining processes around these tasks.

Another sign of the need for process improvement is if a common task is performed differently (with different outcomes) by each person on the team. Perhaps one engineer checks bug fixes into the main branch, while another checks them into the maintenance branch. Other engineers may not bother checking in at all but instead build packages on their local machines. This kind of variance in approach inevitably means variance in outcomes and results. In these cases, developing processes can help us standardize approaches for the purposes of controlling, measuring, and improving on our desired outcomes.

Employees who are overly burdened by mundane tasks represent a third indication that increased process rigor is required. These distractions take away from employees' time, energy, and creativity. Early signs of issues here might include an increased number of engineers complaining about where they spend their time. Engineers generally are not the type of people to keep quiet if they feel hindered in performing their jobs or have a better idea of how to accomplish something.

Process Complexity

Choosing the right level of process complexity is not a matter of determining this level once and for all and forever, but rather choosing the right level of complexity for today.

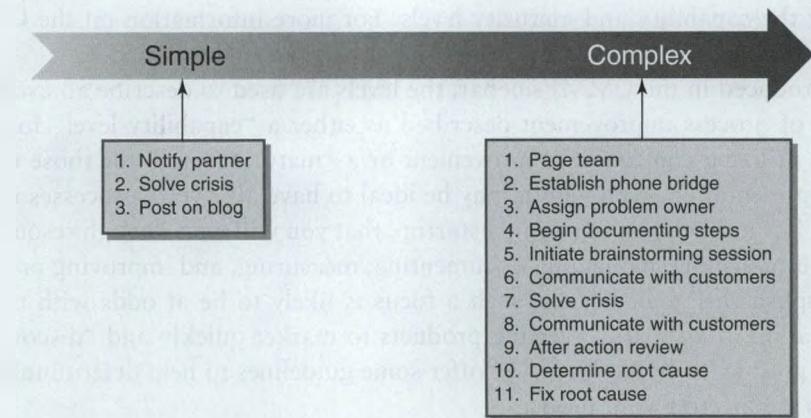


Figure 7.3 *Process Complexity*

We have two suggestions for ways to identify the right amount of process complexity. Before we explore these two methods, let's consider an example of the differences in complexity of processes. Figure 7.3 depicts another gradient, illustrating complexity from simple to complex, with two examples of a process for incident management. The very simple three-step process on the left is most applicable to a small startup with just a couple of engineers. The process on the right is much more complex and is more applicable to a larger organization that has a staffed operations team. As indicated by the gradient, a large variety of levels of complexity are possible for any given process.

Armed with the understanding that there can be many different variations on the same process, we now explore some methods for determining which of these options are right for our organization. We have two suggested methods of determining the process level, which have worked well for us in the past and can be used in combination or separately:

- The first method is to start with the smallest amount of process complexity and iteratively move to the more complex and sophisticated processes periodically. The advantage of this approach is that there is very little chance of overwhelming the team with the new process because it is likely to be much simpler than what is required or what they can tolerate (recall that culture influences how much process should exist on a team). The disadvantages of this approach are that it takes time to narrow in on the optimal amount of process, you must remember to revisit the process periodically, and you must change the process that people are used to on a frequent basis. If these disadvantages are too much, you might consider using our second method.
- The second method of narrowing in on the optimal process for your organization is to let the team decide for itself. This approach can either be democratic,

where everyone gets a voice, or representative, where a chosen few speak for the group. Either way, this approach will get you closer to the optimal amount of process much more quickly than the preceding small-to-large approach. It also has the advantage that it makes the team members feel a sense of ownership, which makes the adoption of the process much easier.

You can choose to implement one method or the other to find your optimal process, or you can mix them together. In a mixed method, the team might decide on the process, but then you step it down slightly to ensure the adoption occurs even more quickly. If the team members feel they need a very strict branching process, you could suggest that initially they ease up a bit and allow for some flexibility on the naming convention and timing of pulling branches until everyone is familiar with the process. After the process is fully established, with a release or two under your belt, you can modify the process and adopt the original suggestions for naming conventions and timing.

When Good Processes Go Bad

So far, we have discussed only the noble attributes of processes. As much as we would like to believe that there is no downside to developing and implementing processes, the reality is that processes can cause issues themselves. Similar to how a poorly designed monitoring system can cause downtime on a production site by overloading servers and services, processes can create problems within the organization when their complexity and level of rigor are not carefully considered. These challenges are generally not the fault of the processes themselves, or even due to having a process; rather, they are due to the fit between the process and the team. You often see this sort of mismatch with technology, designs, and architectures. Technologies and approaches are rarely “always right” or “always wrong”: flat network (sure, we can find a use for that), stateful apps (yes, even that can have a purpose), singletons (yes, they have a place in our world). Use these in the wrong place, however, and you’re sure to have problems.

One of the biggest problems with a badly fitted process is culture clash. When a Wild West culture meets a very complex 100-step process, sparks are sure to fly. The result is that the teams will either ignore the process, in which case it actually causes more problems than it mitigates, or will spend a lot of time complaining about the process. Both of these results are probably worse than not having the process at all. If you witness this kind of behavior on your teams, you must act quickly: The process is not only hurting the team in the short term, but is likely exacerbating the resistance to process or change, which will make implementing any process in the future more difficult.

Another big problem associated with poor fit between the organization and the process is the dreaded “b” word: bureaucracy. This term is defined by the *Merriam-Webster Online Dictionary* as “a system of administration marked by officialism, red tape, and proliferation.” The last thing we want to do with any process is create red tape or officialism. The result of bureaucracy, as you might expect, is lowered productivity and poorer morale. As we mentioned earlier, engineers love challenges and thrive on being asked to do difficult things. When they are so hindered that they are unable to succeed, it is easy for engineers to become demoralized. This is why engineers are typically so ready to speak out about things that diminish their ability to perform their jobs effectively. The challenge for you as a manager and leader is to decide when the complaining is just a matter of not liking change or an engineer being a curmudgeon versus when a real problem needs to be addressed. The best way to distinguish these cases is to know the team and how it generally reacts.

To prevent culture clash and expansion of bureaucracy, or in the event that you already have them, there are three key areas to focus on. First, listen to your teams. When you have learned the nuances of each team member’s personality, including those of your managers if you have a multilayered organization, then you will be able to tell when something is really bothering them versus when something is just a mild disturbance.

Second, implement processes using one of the two methods described earlier in this chapter. Either move from small to large on the process continuum, or let the team decide the right amount of process to establish. One or both of these methods, if you elect to use them in conjunction with each other, should result in a good fit between the team and their processes.

Third, perform periodic maintenance on your processes. This should happen regardless of where you fall on a process maturity scale such as CMMI. As we have repeatedly stated, there is no right or wrong process, just right processes for the right organization at the right time. Also, all organizations change over time. As an organization changes, its processes must be reevaluated to ensure they are still the optimal choices. To keep processes up-to-date, it often helps to assign owners to a process. When owners are established and well known, there is no lack of clarity in determining who can help make improvements. Finally, when examining the efficient use of process, you should always consider ways that you can automate a process. Performing periodic maintenance on the process is critical to ensure it does not turn into a source of culture clash or start to become a bureaucratic nightmare.

Conclusion

Processes serve three general purposes: They augment the management of teams and employees, they standardize employees’ actions while performing repetitive tasks,

and they free employees up from daily mundane decisions to concentrate on grander, more creative ideas. Without processes such as crisis management or capacity planning, and without processes that fit our teams well in terms of complexity and repeatability, we cannot scale our systems effectively.

Many variations in terms of complexity and process maturity exist. Likewise, organizations differ from one another, and they differ from themselves over time because they change as people are hired or leave or mature or learn. The real challenge is fitting the right amount of the right process to the organization at the right time. To achieve this goal, you might start off with a very low amount of process and then slowly increase the granularity and establish stricter definitions around the process. This manner of wading into the process can be effective for easing one's way into the world of processes. Alternatively, you might let the team decide what the right process is for a given task. Either assign one person to figure this out or ask the entire team to sit in a room for a couple hours to make the decision.

Among the problems that can arise with ill-fitting processes are culture clashes and bureaucracy. In this chapter, we pointed out some warning signs of these problems and some corrective actions to take to resolve them. Periodic maintenance of your processes will also go a long way toward circumventing these problems. Reviewing the fit for each process on an annual basis or as the organization undergoes a significant change such as a large number of new hires will help ensure you have the right process for the organization at the right time.

The rest of Part II deals with the details of specific processes that we feel are very important for scalability. For each process, you should remember the lessons learned in this chapter and think about how they would affect the way to introduce and implement the process.

Key Points

- Processes, such as application design or problem resolution, are a critical part of scaling an application.
- Processes assist in management tasks and standardization, and free employees up to focus on more creative endeavors.
- A multitude of process variations exist to choose from for almost any given process.
- Determining to implement any process at all is the first step. After that decision has been made, the next step is to identify the optimal amount of process to implement.
- Two methods for determining the optimal amount of process are (1) migrating from small to large through periodic changes or (2) letting the team decide on the right amount.

- A bad fit between a process and an organization can result in culture clashes or bureaucracy.
- To avoid problems between processes and organizations, you might let the team determine the right amount of process or, alternatively, start slowly and ramp up over time.
- Whenever a process is established, you should assign a person or team as owner of that process.
- Maintenance of processes is critical to ensure organizations do not outgrow their processes.

Chapter 8

Managing Incidents and Problems

Again, if the campaign is protracted, the resources of the State will not be equal to the strain.

—Sun Tzu

Recurring incidents are the enemy of scalability. Recurring incidents steal time away from our teams—time that could be used to create new functionality and greater shareholder value. This chapter considers how to keep incidents from recurring and how to create a learning organization.

Our past performance is the best indicator we have of our future performance, and our past performance is best described by the incidents we have experienced and the underlying problems that caused those incidents. To the extent that we currently have problems scaling our systems to meet end-user demand, or concerns about our ability to scale these systems in the future, our recent incidents and problems are very likely great indications of our current and future limitations. By defining appropriate processes to capture and resolve incidents and processes, we can significantly improve our ability to scale. Failing to recognize and resolve our past problems means failing to learn from our past mistakes in architecture, engineering, and operations. Failing to recognize past mistakes and learn from them with the intent of ensuring that we do not repeat them is disastrous in any field or discipline. For that reason, we've dedicated a chapter to incident and problem management.

Throughout this chapter, we will rely upon the United Kingdom's Office of Government Commerce (OGC) Information Technology Infrastructure Library (ITIL) for definitions of certain words and processes. The ITIL and the Control Objectives for Information and Related Technology (COBIT) created by the Information Systems Audit and Control Association are the two most commonly used frameworks for developing and maturing processes related to managing the software, systems, and organizations within information technology. This chapter is not meant to be a comprehensive review or endorsement of either ITIL or COBIT. Rather, we will

summarize some of the most important aspects of the parts of these systems as they relate to managing incidents and their associated problems and identify the portions that you absolutely must have regardless of the size or complexity of your organization or company.

Whether you work at a large company that expects to complete a full implementation of either ITIL or COBIT or a small company that is looking for a fast and lean process to help identify and eliminate recurring scalability-related issues, the following activities are absolutely necessary:

- Recognize the difference between incidents and problems and track them accordingly.
- Follow an incident management life cycle (such as DRIER [identified shortly]) to properly catalog, close, report on, and track incidents.
- Develop a problem management tracking system and life cycle to ensure that you are appropriately closing and reacting to scalability-related problems.
- Implement a daily incident and problem review to support your incident and problem management processes.
- Implement a quarterly incident review to learn from past mistakes and help identify issues repeatedly impacting your ability to scale.
- Implement a robust postmortem process to get to the heart of all problems.

What Is an Incident?

The ITIL definition of an *incident* is “Any event which is not part of the standard operation of a service and which causes, or may cause, an interruption to, or a reduction in, the quality of that service.” That definition has a bit of “government-speak” in it. Let’s give this term a more easily understood definition: “any event that reduces the quality of our service.”¹ An incident here could be a downtime-related event, an event that causes slowness in response time to end users, or an event that causes incorrect or unexpected results to be returned to end users.

Issue management, as defined by the ITIL, is “to restore normal operations as quickly as possible with the least possible impact on either the business or the user, at a cost-effective price.” Thus, management of an issue really becomes the management of the impact of the issue. We love this definition and love the approach, because it separates cause from impact. We want to resolve an issue as quickly as possible, but that does not necessarily mean understanding its root cause. Therefore,

1. “ITIL Open Guide: Incident Management.” http://www.itilibrary.org/index.php?page=incident_management.

rapidly resolving an incident is critical to the perception of scale. Once a scalability-related incident occurs, it starts to cause the perception (and, of course, the reality) of a lack of scalability.

Now that we understand that an incident is an unwanted event in our system that affects our availability or service levels and that incident management has to do with the timely and cost-effective resolution of incidents to force the system into the perceived normal behavior, let's discuss problems and problem management.

What Is a Problem?

The ITIL defines a *problem* as “the unknown cause of one or more incidents, often identified as a result of multiple similar incidents.” It further defines a “known error” as an identified root cause of a problem. Finally, “The objective of Problem Management is to minimize the impact of problems on the organization.”²

Again, we can see the purposeful separation of events (incidents) and their causes (problems). This simple separation of definition for incident and problem helps us in our everyday lives by forcing us to think about their resolution differently. If for every incident we attempt to find root cause before restoring service, we will very likely have lower availability than if we separate the restoration of service from the identification of cause. Furthermore, the skills necessary to restore service and manage a system back to proper operation may very well be different from those necessary to identify the root cause of any given incident. If that is the case, serializing the two processes not only wastes engineering time, but also further destroys shareholder value.

For example, assume that a Web site makes use of a monolithic database structure and is unavailable in the event that the database fails. This Web site has a database failure where the database simply crashes and all processes running the database die during its peak traffic period from 11 a.m. to 1 p.m. One very conservative approach to this problem might be to say that you never restart your database until you know why it failed. Making this determination could take hours and maybe even days while you go through log and core files and bring in your database vendor to help you analyze everything. The intent behind this approach is obvious—you don't want to cause any data corruption when you restart the database.

In reality, most databases these days can recover from nearly any crash without significant data hazards. A quick examination could tell you that no processes are running, that you have several core and log files, and that a restart of the database might actually help you understand which type of problem you are experiencing. Perhaps you can start up the database and run a few quick “health checks,” like the insertion

2. “ITIL Open Library.” http://www.itilibrary.org/index.php?page=problem_management.

and updating of some dummy data to verify that things are likely to work well, and then put the database back into service. Obviously, this approach, assuming the database will restart, is likely to result in less downtime associated with scalability-related events than serializing the management of the problem (identifying the root cause) and the management of the incident (restoration of service).

We've just highlighted a very real conflict between these two processes that we'll address later in this chapter. Specifically, incident management (the restoration of service) and problem management (the identification and resolution of the root cause) are often in conflict with each other. The rapid restoration of service often conflicts with the forensic data gathering necessary for problem management. Perhaps the restart of servers or services really will cause the destruction of critical data. We'll discuss how to handle this situation later. For now, recognize that there is a benefit in thinking about the differences in actions necessary for the restoration of service and the resolution of problems.

The Components of Incident Management

The ITIL defines the activities essential to the incident management process as follows:

- Incident detection and recording
- Classification and initial support
- Investigation and diagnosis
- Resolution and recovery
- Incident closure
- Incident ownership, monitoring, tracking, and communication

Implicit in this list is an ordering such that nothing can happen before incident detection, classification comes before investigation and diagnosis, resolution and recovery must happen only after initial investigation, and so on. We completely agree with this list of necessary actions. However, if your organization is not strictly governed by the OGC and you do not require any OGC-related certification, there are some simple changes you can make to this order that will accelerate issue recovery. First, we wish to create our own simplified definitions of the preceding activities.

Incident detection and recording is the activity of identifying that an incident is affecting users or the operation of the system and then recording it. Both elements are very important, and many companies have quite a bit they can do to make both actions better and faster. Incident detection is all about monitoring your systems. Do you have customer experience monitors in place to identify problems before the first

customer complaint arrives? Do they measure the same things customers do? In our experience, it is very important to perform actual customer transactions within your system and measure them over time both for the expected results (are they returning the right data?) and for the expected response times (are they operating as quickly as you would expect?).

A Framework for Maturing Monitoring

Far too often, we see clients attempting to implement monitoring solutions that are intended to tell them the root cause of any potential problem they might be facing. This sounds great, but this monitoring panacea rarely works. Its failures are largely attributed to two issues:

- The systems the organization is attempting to monitor aren't designed to be monitored.
- The company does not approach monitoring in a planned, methodical evolutionary (or iterative) fashion.

You should not expect a monitoring system (or incident identification system) to correctly identify the faults within your platform if you did not design your platform to be monitored. The best-designed systems build the monitoring and notification of incidents into their code and systems. As an example, world-class real-time monitoring solutions have the capability to log the times and errors for each internal call to a service. In this case, the service may be a call to a data store or another Web service that exposes account information, and so on. The resulting times, rates, and types of errors might be plotted in real time in a statistical process control chart (SPC) with out-of-bound conditions highlighted as an alert on some sort of monitoring panel.

Designing a platform to be monitored is necessary but not sufficient to identify and resolve incidents quickly. You also need a system that identifies issues from the perspective of your customer and helps to identify the underlying platform component causing that problem.

Far too many companies bypass the step of monitoring their systems from a customer perspective. Ideally, you should build or incorporate a real-time monitoring and alerting system that simulates how customers use the platform and performs the most critical transactions. Throw an alert when the system performance falls outside internally defined service levels for response time and availability.

Next, implement processes and supporting tools to help identify which system is causing the incident. In the ideal world, you will have developed a fault-isolative architecture to create *failure domains* that will isolate failures and help you determine where the fault is occurring (we discuss failure domains and fault-isolative architectures in Chapter 21, Creating Fault-Isolative Architectural Structures). At the very least, you need monitoring that can help provide visibility into your platform's vital signs and critical components. These are typically aggregated system statistics such as load, CPU, memory utilization, or network activity.

Note that our first step here is not just issue identification but also the recording of the issues. Many companies that correctly identify issues don't immediately record them before taking other actions or don't have systems implemented that will record the problems. The best approach is to have an automated system that will immediately record the issue and its timestamp, leaving operators free to handle the rest of the process.

The ITIL identifies classification and initial support as the next step, but we believe that in many companies this can really just be the step of "getting the right people involved." Classification is an activity that can happen in hindsight in our estimation—after the issue is resolved.

Investigation and diagnosis is followed by resolution and recovery. Put simply, these activities identify what has failed and then take the appropriate steps to put that service back into proper working order. As an example, they might determine that application server #5 is not responding (investigation and diagnosis), at which point we immediately attempt a reboot (a resolution step) and the system recovers (recovery).

Incident closure is the logging of all information associated with the incident. The final steps include assigning an owner for follow-up, communication, tracking, and monitoring.

We often recommend an easily remembered acronym when implementing incident management (see Figure 8.1). Our acronym supports ITIL implementations and for smaller companies can be adopted with or without an ITIL implementation. This acronym, DRIER, stands for

- Detect an incident through monitoring or customer contact.
- Report the incident, or log it into the system responsible for tracking all incidents, failures, or other events.
- Investigate the incident to determine what should be done.
- Escalate the incident if it is not resolved in a timely fashion.
- Resolve the incident by restoring end-user functionality and log all information for follow-up.

In developing DRIER, we've attempted to make it easier for our clients to understand how issue management can be effectively implemented. Note that although we've removed the classification of issues from our acronym, we still expect that these activities will be performed to develop data from the system and help inform other processes. We recommend that the classification of issues happen within the daily incident management meeting (discussed later in this chapter).

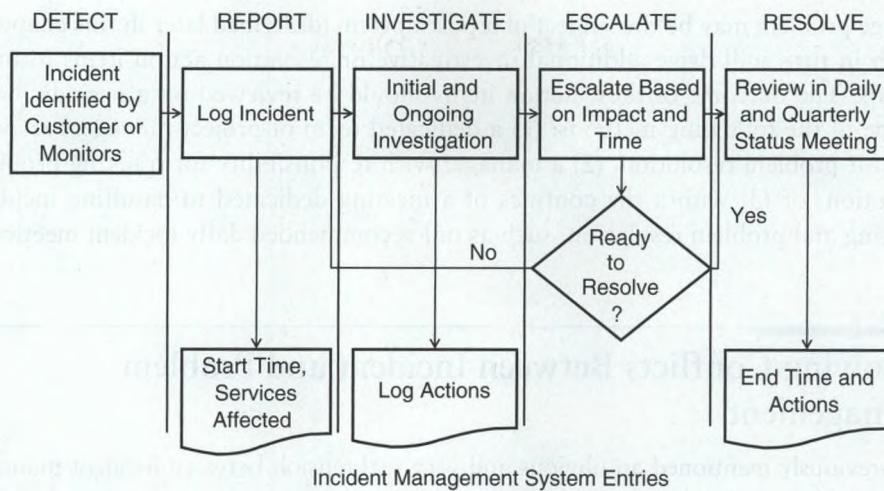


Figure 8.1 DRIER Process

The Components of Problem Management

The ITIL-defined components of problem management are a little more difficult to navigate than those for incident management. The ITIL definitions define a number of processes that control other processes. For anything but a large organization, this can be a bit cumbersome. This section attempts to boil these processes down to something more manageable. Remember that problems are the causes of incidents and are likely culprits for a large number of scalability problems.

Problems in our model start concurrently with an issue and last until the root cause of an incident is identified. As such, most problems last longer than most incidents. A problem can be the cause of many incidents.

Just as with incidents, we need a type of workflow that supports our problem resolutions. We need a system or place to keep all of the open problems and to ensure that they can be associated with the incidents they cause. We also need to track these problems to closure. Our reasoning underlying this definition of “closure” is that a problem exists until it no longer causes incidents.

Problems vary in size. Small problems can be handed to a single person. When they are ready for closure, validation is done in QA and by meeting the required testing criteria. A problem is then confirmed as closed by the appropriate management or owner of the system experiencing the incidents and problems. Larger problems are more complex and need specialized processes to help ensure their rapid resolution.

A large problem may be the subject of a postmortem (described later in this chapter), which in turn will drive additional investigative or resolution action items to individuals. The outcome of these action items should be reviewed on a periodic basis by one of the following methods: (1) a dedicated team of project managers responsible for problem resolution, (2) a manager with responsibility for tracking problem resolution, or (3) within the confines of a meeting dedicated to handling incident tracking and problem resolution, such as our recommended daily incident meeting.

Resolving Conflicts Between Incident and Problem Management

We previously mentioned an obvious and very real tension between incident management and problem management. Very often, the actions necessary to restore a system to service will potentially destroy evidence necessary to determine the root cause (problem resolution). Our experience is that incident resolution (the restoration of service) should always trump root cause identification unless an incident will continue to have a high frequency of recurrence without root cause and problem resolution.

That said, we also believe it is important to think through your approach and protocol *before* you face the unenviable position of needing to make calls on when to restore service and when to continue root cause analysis. We have some suggestions:

- Determine what needs to be collected by the system before system restoration.
- Determine how long you are willing to collect diagnostic information before restoration.
- Determine how many times you will allow a system to fail before you require root cause analysis to be more important than system restoration.
- Determine who should make the decision as to when systems should be restored if there is a conflict (who is the R and who is the A).

In some instances, due to a lack of forensic data, the problem management process may not yield a clear set of root causes. In these cases, it is wise to make the preceding determinations for that incident and ensure you clearly identify all of the people who should be involved, if the incident reoccurs, to capture better diagnostics more quickly.

Incident and Problem Life Cycles

There is an implied life cycle and relationship between incidents and problems. An incident is open or ongoing until the system is restored. This restoration of the

system may cause the incident to be closed in some life cycles, or it may move the incident to “resolved” status in other life cycles. Problems are related to incidents and are likely opened at the time that an incident happens, potentially “resolved” after the root cause is determined, and “closed” after the problem is corrected and verified within the production environment. Depending on your particular approach, incidents might be closed after service is restored, or several incidents associated with a single problem might not be finally closed until their associated problems are fixed.

Regardless of which words you associate with stages in the life cycles, we often recommend the following simple phases be tracked to collect good data about incidents, problems, and their production costs:

Incident Life Cycle

Open	Upon an incident or when an event happens in production
Resolved	When service is restored
Closed	When the associated problems have been closed in production

Problem Life Cycle

Open	When associated to an incident
Identified	When a root cause of the problem is known
Closed	When the problem has been “fixed” in production and verified

Our approach here is intended to ensure that incidents remain open until the problems that cause them have their root causes identified and fixed in the production environment. Note that these life cycles don’t address the other data we would like to see associated with incidents and problems, such as the classifications we suggest adding in the daily incident meeting.

We recommend against reopening incidents, as doing so makes it more difficult to query your incident tracking system to identify how often an incident reoccurs. That said, having a way to “reopen” a problem is useful as long as you can determine how often you take this step. Having a problem reoccur after it was thought to be closed is an indication that you are not truly finding its root cause and is an important data point to any organization. Consistent failure to correctly identify the root cause results in continued incidents and is disastrous to your scalability initiatives, because it steals valuable time away from your organization. Ongoing fruitless searches for root causes lead to repeated failures for your customers, dilute shareholder wealth, and stymie all other initiatives having to do with high availability and quality of service for your end users.

Implementing the Daily Incident Meeting

The daily incident meeting (also called the daily incident management meeting) is a meeting and process we encourage all of our clients to adopt as quickly as possible. This meeting occurs daily in most high-transaction, rapid-growth companies and serves to tie together the incident management process and the problem management process.

During this meeting, all incidents from the previous day are reviewed to assign ownership of problem management to an individual, or if necessary to a group. The frequency with which a problem occurs and its resulting impact serve to prioritize the problems to be subjected to root cause analysis and fixed. We recommend that incidents be given classifications meaningful to the company within this meeting. Such classifications may take into account severity, systems affected, customers affected, and so on. Ultimately, the classification system employed should be meaningful in future reviews of incidents to determine impact and areas of the system causing the company the greatest pain. This last point is especially important to identify scalability-related issues throughout the system.

Additionally, the open problems are reviewed. Open problems are problems associated with incidents that may be in the open or identified state but not completely closed (i.e., the problem has not been put through root cause analysis and fixed in the production environment). The problems are reviewed to ensure that they are prioritized appropriately, that progress is being made in identifying their causes, and that no help is required by the owners assigned the problems. It may not be possible to review all problems in a single day; if that is the case, a rotating review of problems should start with the highest-priority problems (those with the greatest impact) being reviewed most frequently. Problems should also be classified in a manner consistent with business need and indicative of the type of problem (e.g., internal versus vendor related), subsystem (e.g., storage, server, database, login application, buying application), and type of impact (e.g., scalability, availability, response time). This last classification is especially important if the organization is to be able to pull out meaningful data to help inform its scale efforts in processes. Problems should inherit the impact determined by their incidents, including the aggregate downtime, response time issues, and so on.

Let's pause to review the workflow we've discussed thus far in this section. We've identified the need to associate incidents with systems and other classifications, the need to associate problems with incidents and still more classifications, and the need to review data over time. Furthermore, owners should be assigned to problems and potentially to incidents, and status needs to be maintained for everything until closed. Most readers have probably figured out that a system to aid in this collection of information would be really useful. Most open source and third-party "problem ticketing" solutions offer a majority of this functionality enabled with some small configuration right out of the box. We don't think you should wait to implement an

incident management process, a problem management process, and a daily meeting until you have a tracking system. However, it will certainly help if you put a tracking system in place shortly after the implementation of these processes.

Implementing the Quarterly Incident Review

No set of incident and problem management processes would be complete without a process for reviewing their effectiveness and ensuring that they are successful in eliminating recurring incidents and problems.

In the “Incident and Problem Life Cycles” section, we mentioned that you may find yourself incorrectly identifying root causes for some problems. This is almost guaranteed to happen to you at some point, and you need to have a way for determining when it is happening. Is the same person incorrectly identifying the root causes? This failure may require coaching of the individual, a change in the person’s responsibilities, or removal of the person from the organization. Is the same subsystem consistently being misdiagnosed? If so, perhaps you have insufficient training or documentation on how the system really behaves. Are you consistently having problems with a single partner or vendor? If so, you might need to implement a vendor scorecard process or give the vendor other performance-related feedback.

Additionally, to ensure that your scalability efforts are applied to the right systems, you need to review past system performance and evaluate the frequency and impact of past events on a per-system or per-subsystem basis. This evaluation helps to inform the prioritization for future architectural work and becomes an input to processes such as the headroom process or 10x process that we describe in Chapter 11, Determining Headroom for Applications.

The output of the quarterly incident review also provides the data that you need to define the business case for scalability investments, as described in Chapter 6, Relationships, Mindset, and the Business Case. Showing the business leaders where you will put your effort and why, prioritized by impact, is a powerful persuasive tool when you are seeking to secure the resources necessary to run your systems and maximize shareholder wealth. Furthermore, using that data to paint the story of how your efforts are resulting in fewer scalability-associated outages and response time issues will make the case that past investments are paying dividends and helps give you the credibility you need to continue doing a good job.

The Postmortem Process

Earlier in this chapter, we noted that some large problems require a special approach to help resolve them. Most often, these large problems will require a cross-functional

brainstorming meeting, often referred to as a *postmortem* or after-action review meeting. Although the postmortem meeting is valuable in helping to identify root causes for a problem, if run properly, it can also assist in identifying issues related to process and training. It should not be used as a forum for finger pointing.

The first step in developing a postmortem process is to determine the size of incident or group of incidents for which a postmortem should be required. Although postmortems are very useful, they take multiple people away from their assigned tasks and put them on the special duty of helping to determine what failed and what can work better within a system, process, or organization. To get the most out of these meetings, you must make sure you are maximizing the participants' time and learning everything possible from the incident. In terms of the metrics and measurements discussed in Chapter 5, Management 101, the assignment of people to postmortem duty reduces the engineering efficiency metric, as these individuals would be spending hours away from their responsibilities of creating product and scaling systems.

The attendees of the postmortem should consist of a cross-functional team from software engineering, systems administration, database administration, network engineering, operations, and all other technical organizations that could have valuable input, such as capacity planning. A manager who is trained in facilitating meetings and also has some technical background should be assigned to run the meeting.

The input to the postmortem process comprises a timeline that includes data and timestamps leading up to the end-user incident, the time of the actual customer incident, all times and actions taken during the incident, and everything that happened up until the time of the postmortem. Ideally, all actions and their associated timestamps will have been logged in a system during the restoration of the service, and all post-incident actions will have been logged either in the same system or in other places to cover what has been done to collect diagnostics and fix the root cause. Logs should be parsed to grab all meaningful data leading up to the incidents with timestamps associated to the collected data. Figure 8.2 introduces the process that the team should cover during the postmortem meeting.

In the first step in the postmortem process, which we call the Timeline Phase, the participants review the initial timeline and ensure that it is complete. Attendees of the postmortem might identify that critical dates, times, and actions are missing. For instance, the team might notice that an alert from an application was thrown and not acted upon two hours before the first item identified in the initial incident timeline. Note that during this phase of the process, only times, actions, and events should be recorded; no problems or issues should be identified or debated.

The next step in the postmortem meeting—the Issue Phase—examines the timeline and identifies issues, mistakes, problems, or areas where additional data would be useful. Each of these areas is logged as an issue, but no discussion over what should happen to fix the issue happens until the entire timeline is discussed and all

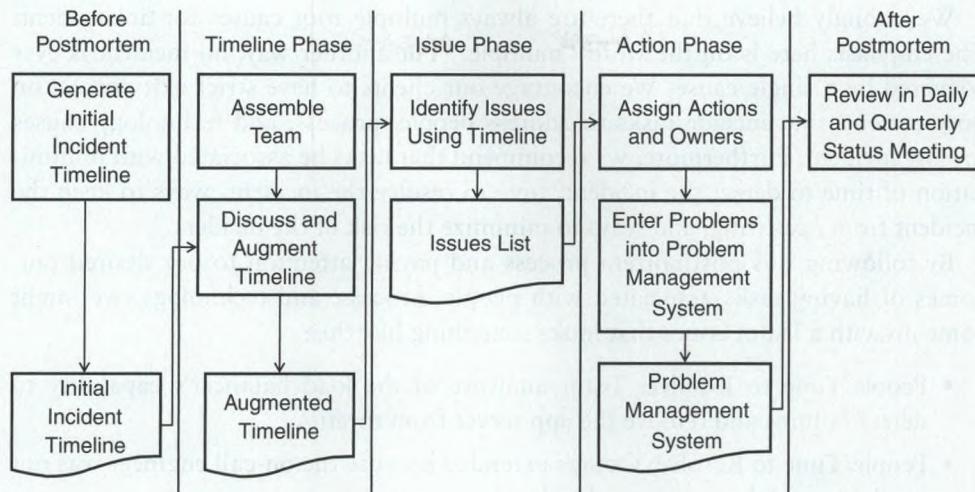


Figure 8.2 Example Postmortem Process

of the issues are identified. During this phase, the facilitator of the meeting needs to create an environment in which participants are encouraged to voice all ideas and concerns over what might be issues without concern for retribution or retaliation. The facilitator should also ensure that no reference to ownership is made. For instance, it is inappropriate in a postmortem to say, “John ran the wrong command there.” Instead, the reference should be “At 10:12 a.m., command A was incorrectly issued.” Management can address individual accountability later if someone is found to have violated company policy, repeatedly makes the same mistake, or simply needs some coaching. The postmortem is not meant to be a public flogging of an individual. In fact, if it is used as such a forum, the efficacy of the process will be destroyed.

After a first run through the timeline is made and an issues list generated, a second pass through the timeline should be made with an eye toward determining whether actions were taken in a timely manner. For instance, suppose a system starts to exhibit high CPU utilization at 10 a.m. and no action is taken. At noon, customers start to complain about slow response times. A second pass through the timeline might result in someone noting that the early indication of CPU might be correlated with slow response times later and an issue generated in regard to this relationship.

After a complete list of issues is generated from at least one pass (and preferably two passes) through the timeline, we are ready to begin the Action Phase—that is, the creation of the task list. The task list is generated from the issues list, with at least one task identified for each issue. If the team cannot agree on a specific action to fix some issue, an analysis task can be created to identify additional tasks to solve the issue.

We strongly believe that there are always multiple root causes for an incident. The emphasis here is on the word “multiple.” Put another way, no incident is ever triggered by a single cause. We encourage our clients to have strict exit criteria on postmortems that include tasks to address people, process, and technology causes for any incident. Furthermore, we recommend that tasks be associated with minimization of time to detect the incident, time to resolve the incident, ways to keep the incident from occurring, and ways to minimize the risk of the incident.

By following this postmortem process and paying attention to our desired outcomes of having tasks associated with people, process, and technology, we might come up with a list of issues that looks something like this:

- People/Time to Resolve: Team unaware of the load balancer’s capability to detect failures and remove the app server from rotation.
- People/Time to Resolve: Outage extended because the on-call engineer was not available or did not answer the alert.
- Process/Time to Resolve: Network Operations Center personnel did not have access or training to remove the failed app server from the load balancer upon failure of the app server.
- Technology/Ability to Avoid: Load balancer did not have a test to determine the health of the app server and remove it from rotation.
- Technology/Time to Detect: Alerts for a failed ping test on the app server did not immediately show up in Network Operations Center.
- Technology/Ability to Avoid: Single drive failure in the app server should not result in app server failure.

After the task list is created, owners should be assigned for each task. Where necessary, use the RASCI methodology to clearly identify who is responsible for completing each task, who is the approver of each task, and so on. Also use the SMART criteria when defining the tasks, making them specific, measurable, aggressive/attainable, realistic, and timely. Although initially intended for goals, the SMART acronym can also help ensure that we put time limits on our tasks. Ideally, these items will be logged into a problem management system or database for future follow-up.

Putting It All Together

Collectively, the components of issue management, process management, the daily incident management meeting, and the quarterly incident review, plus a well-defined postmortem process and a system to track and report on all systems and problems,

will give us a good foundation for identifying, reporting, prioritizing, and taking action to address past scalability issues.

Any given incident will follow the DRIER process of detecting the issue, reporting on the issue, investigating the issue, escalating the issue, and resolving the issue. The issue is immediately entered into a system developed to track incidents and problems. Investigation leads to a set of immediate actions, and if help is needed, we escalate the situation according to our escalation processes. Resolving the issue changes the issue status to “resolved” but does not close the incident until its root cause is identified and fixed within our production environment.

The problem is assigned to an individual or organization during the daily status review unless it is of such severity that it needs immediate assignment. During that daily review, we also review incidents and their status from the previous day and the high-priority problems that remain open in the system. In addition, we validate the closure of problems and assign categories for both incidents and problems in the daily meeting.

Problems, when assigned, are worked by the team or individual assigned to them in priority order. After root causes are determined, the problem moves to “identified” status; when fixed and validated in production, it is designated as “closed.” Large problems go through a well-defined postmortem process that focuses on identifying all possible issues within the process and technology stacks. Problems are tracked within the same system and reviewed in the daily meeting.

On a quarterly basis, we review incidents and problems to determine whether our processes are correctly closing problems and to determine the most common occurrences of incidents. This data is collected and used to prioritize architectural, organizational, and process changes to aid in our mission of increasing scalability. Additional data is collected to determine where we are doing well in reducing scale-related problems and to help create the business case for scale initiatives.

Conclusion

One of the most important processes within any technology organization is the process of resolving, tracking, and reporting on incidents and problems. Incident resolution and problem management should be envisioned as two separate and sometimes competing processes. In addition, some sort of system is needed to help manage the relationships and the data associated with these processes.

A few simple meetings can help meld the incident and problem management processes. The daily incident management meeting helps the organization manage incident and problem resolution and status, whereas the quarterly incident review helps team members create a continual process improvement cycle. Finally, supportive processes such as the postmortem process can help drive major problem resolution.

Key Points

- Incidents are issues in the production environment. Incident management is the process focused on timely and cost-effective restoration of service in the production environment.
- Problems are the cause of incidents. Problem management is the process focused on determining root cause of and correcting problems.
- Incidents can be managed using the acronym DRIER, which stands for detect, report, investigate, escalate, and resolve.
- There is a natural tension between incident and problem management. Rapid restoration of service may cause some forensic information to be lost that would otherwise be useful in problem management. Thinking through how much time should be allowed to collect data and which data should be collected will help ease this tension for any given incident.
- Incidents and problems should have defined life cycles. For example, an incident may be opened, resolved, and closed, whereas a problem may be open, identified, and closed.
- A daily incident meeting should be organized to review incidents and problem status, assign owners, and assign meaningful business categorizations.
- A quarterly incident review is an opportunity to look back at past incidents and problems so as to validate proper first-time closure of problems and thematically analyze both problems and incidents to help prioritize scalability-related architecture, process, and organization work.
- The postmortem process is a brainstorming process used for large incidents and problems to help drive their closure and identify supporting tasks.
- Postmortems should always focus on the identification of multiple causes (problems) for any incident. These causes may address all dimensions of the product development process: people, process, and technology. Ideally, the output of the postmortem will identify how to avoid such an incident in the future, how to detect the incident more quickly, and how to drive to incident resolution faster.

Chapter 9

Managing Crises and Escalations

There is no instance of a country having benefitted from prolonged warfare.

—Sun Tzu

You've probably never seen the 404 (page not found) error on Etsy's site (www.etsy.com). The error page is a cartoon that depicts a woman knitting a three-armed sweater (Figure 9.1). Etsy is the de facto marketplace for handmade or vintage items, and as such it seems only fitting that the company commissioned an actual three-armed sweater that it uses to reward the engineer who most spectacularly brings down the site. While this fun fact is indicative of the amazing culture that thrives in Etsy's technology teams, what's more interesting about this 404 page is the site incident that it caused a few years ago. John Allspaw, Senior VP of Technical Operations at Etsy and author of *The Art of Capacity Planning* (O'Reilly Media, 2008) and *Web Operations: Keeping the Data on Time* (O'Reilly, 2010), tells the story. It all started



Figure 9.1 *Etsy's 404 Error Page*

with the removal of a CSS file. This particular CSS file supported outdated versions of Internet Explorer browser (IE versions less than 9.0). Everything appeared normal for a few minutes after the removal of the file—but then the CPU cycles on all 80 Web servers spiked to 100%.

Allspaw and his team have been incredibly focused on monitoring; they monitor more than 1 million time-based metrics such as registrations and payment types. All of this data gives the operations and software developers incredible insight into the systems. In Allspaw's story, the monitors almost immediately began alerting that something was amiss. A combination of tech ops and software developers gathered in a virtual IRC chat room as well as a physical conference room in Etsy's Brooklyn office. The team formed hypotheses and quickly began vetting them. Anthony, the engineer who would eventually get awarded the three-armed sweater for this incident, determined that during the space of time between the rsync of the static assets and the execution of the code (i.e., the Smarty tag that checked asset version), an error was being thrown. The code in question was executed on every request, not just IE browsers, which consumed all the available CPU cycles on all the servers.

Now that Allspaw's team had identified the problem, the fix should have been easy—put back the missing CSS file. But not so fast: With all the CPU being consumed by the infinite loop, there wasn't even enough CPU to allow an SSH connect onto the servers. The administrators connected to the iLO (integrated-Lights Out) cards to reboot the servers, but as soon as they came back online, the CPU spiked. The team started pulling servers out of the load balancer and then rebooting them, which gave them time to SSH into the servers and replace the CSS file. The administrators and network engineers were able to do this so quickly because Allspaw's team had practiced synchronization and sequencing exercises many times before this incident. Because they already had a communication plan in place, they were able to communicate with their customers during the entirety of this incident through both the site [EtsyStatus](http://etsystatus.com/) (<http://etsystatus.com/>) and the Twitter account @etsystatus. These communications reassured customers that Etsy was working on the problem.

Etsy is one of the best-prepared and best-executing teams in the industry. But even this execution prowess and preparedness doesn't keep the company from having issues and needing to award the three-armed sweater to an engineer. Etsy understands how critical it is to respond quickly and appropriately to incidents when they happen—and take our word for it, they *will* happen. In this chapter, we discuss how to handle major crises both during the incident and afterward.

What Is a Crisis?

The *Merriam-Webster Dictionary* defines a *crisis* as “the turning point for better or worse in an acute disease or fever” and “a paroxysmal attack of pain, distress or

disordered function." A crisis can be both cathartic and galvanizing. It can be your Nietzsche event, allowing you to rise from the ashes like the mythical Phoenix. It can in one fell swoop fix many of the things we described in Chapter 6, Relationships, Mind-set, and the Business Case, and force the company to focus on scale. If you've arrived here and take the right actions, your organization can become significantly better.

How do you know if an incident has risen to the level sufficient to qualify it as a crisis? To answer this question, you need to know the business impact of an incident. Perhaps a 30- to 60-minute failure between 1 a.m. and 1:30 a.m. is not really a crisis situation for your company, whereas a 3-minute failure at noon is a major crisis. Your business may be such that you make 30% of your annual revenue during the three weeks surrounding Christmas. As such, downtime during this three-week period may be an order of magnitude more costly than downtime during the remainder of the year. In this case, a crisis situation for you may be any downtime between the first and third weeks of December, whereas at any other point during the year you are willing to tolerate 30-minute outages. Your business may rely upon a data warehouse that supports hundreds of analysts between the hours of 8 a.m. and 7 p.m., but has nearly no usage after 7 p.m. or anytime during weekends. A crisis for you in this case may be any outage during working hours that would idle your analysts.

Of course, not all crises are equal, and obviously not everything should be treated as a crisis. Certainly, a brownout of activity on your Web site for 3 minutes Monday through Friday during "prime time" (peak utilization) is more of a crisis than a single 30-minute event during relatively low user activity levels. Our point here is that you must determine the unique *crisis threshold* for your company. Everything that exceeds this threshold should be treated as a crisis. Losing a leg is absolutely worse than losing a finger, but both require immediate medical attention. The same is true with crises; after the predefined crisis threshold is passed, each crisis should be approached the same way.

Recall from Chapter 8, Managing Incidents and Problems, that recurring problems (those problems that occur more than once) rob you of time and destroy your ability to scale your services and scale your organization. Crises also ruin scale, because they steal inordinately large amounts of resources. Allowing the root cause of a crisis to surface more than once will not only waste vast resources and keep you from scaling your organization and services, but also carries the risk of destroying your business.

Why Differentiate a Crisis from Any Other Incident?

You can't treat a crisis like any normal incident, because it won't treat you the way any normal incident would treat you. This is the time to restore service faster than ever before, and then continue working to get to the real root causes (problems) of

said incident. With the clock ticking, customer satisfaction, future revenue, and even business viability are on the line.

Although we believe that there is a point at which adding resources to a project has diminishing and even negative returns, in a crisis you are looking for the shortest possible time to resolution rather than the efficiency or return on those resources. A crisis is not the time to think about future product delivery, as such thoughts and their resulting actions will merely increase the duration of the crisis. Instead, you need to lead by example and be at the scene of the crisis for as long as it is humanly possible and eliminate all other distractions from your schedule. Every minute that the crisis continues is another minute that destroys shareholder value.

Your job is to stop the crisis from causing a negative trend within your business. If you can't fix it quickly by getting enough people on the problem to ensure that you have appropriate coverage, three things will happen. First, the crisis will perpetuate itself. Events will happen again and again, and you will lose customers, revenue, and maybe your business. Second, in allowing the crisis to siphon precious time out of your organization over a prolonged period, you will eventually lose traction on other projects. The very thing you were trying to avoid by not putting "all hands on deck" will happen anyway *and* you will have allowed the problem to go on longer than necessary. Third, you will lose credibility.

How Crises Can Change a Company

While crises are bad, if properly handled they can have significant long-term benefits to a company. The benefits are realized if, as a result of a crisis or series of crises, the company enacts long-term changes in its processes, culture, organizational structure, and architecture. In these cases, crisis serves as a catalyst for change.

Our job, of course, is to avert crises whenever possible given their cost to shareholders and employees. But when crises happen (and they will), we must adhere to the adage that "A crisis is a terrible thing to waste." This adage indicates a strong desire to maximize the learning from a crisis through diligent postmortem assessments, and as such strengthen our people, processes, technologies, and architectures.

The eBay Scalability Crisis

As proof that a crisis can change a company, consider eBay in 1999. In its early days, eBay was the darling of the Internet. Until the summer of 1999, few, if any, companies had experienced such exponential growth in users, revenue, and profits. Throughout the summer of 1999, however, eBay experienced many outages, including a 20-plus hour outage in June.

These outages were at least partially responsible for the reduction in stock price from a high in the mid-\$20s in the week of April 26, 1999, to a low of \$9.47 in the week of August 2, 1999.¹

The cause of the outages isn't really as important as what happened within eBay *after* the outages occurred. Additional executives were brought in to ensure that the engineering organization, the engineering processes, and the technology they produced could scale to the demand placed on them by the eBay community. Initially, additional capital was deployed to purchase systems and equipment (although eBay succeeded in lowering both its technology expense and capital on an absolute basis well into 2001). Processes were put in place to help the company design systems that were more scalable, and the engineering team was augmented with engineers experienced in high availability and scalable designs and architectures. Most importantly, the company created a culture of scalability. The lessons from the summer of pain are still discussed at eBay, and scalability has since become part of eBay's DNA.

eBay continued to experience crises from time to time, but these crises were smaller in terms of their impact and shorter in terms of their duration as compared to the summer of 1999. Its implementation of a culture of scalability netted architectural changes, people changes, and process changes. One such change was eBay's focus on managing each and every crisis in the fashion described in this chapter.

Order Out of Chaos

Bringing together and managing people from several different organizations during a crisis situation is difficult. Most organizations have their own unique subculture, and oftentimes those subcultures don't speak the same technical language. It is entirely possible that an application developer will use terms with which a systems engineer is not familiar, and vice versa.

If left unmanaged, the collision of a large number of people from multiple organizations within a crisis situation will create chaos. This chaos will feed on itself, creating a vicious cycle that can actually prolong the crisis or—worse yet—aggravate the damage done in the crisis through someone taking an ill-advised action. Indeed, if you cannot effectively manage the force you throw at a crisis, you are better off using fewer people.

Your company may have a crisis management process that consists of both phone and chat (instant messaging or IRC) communications. If you listen on the phone or follow the chat session, you are very likely to see an unguided set of discussions and statements as different people and organizations go about troubleshooting. You

1. August 4, 1999. <http://finance.yahoo.com/q/hp?s=EBAY&a=07&b=2&c=1999&d=08&e=6&f=1999&g=d>.

may have questions asked that go unanswered or requests to try something that go without authorization. You might as well be witnessing a grade school recess, with different groups of children running around doing different things with absolutely no coordination of effort. But a crisis situation isn't a recess—it's a war, and in war such a lack of coordination results in an increased casualty rate through "friendly fire." In a technology crisis, these friendly casualties are manifested as prolonged outages, lost data, and increased customer impact.

You must control the chaos. Rather than a grade school recess, you hope to see a high school football game. Ideally, you will witness a group of professionals being led with confidence to identify a path to restoration and subsequently a path to identification of root causes.

Different groups should have specific objectives and guidelines unique to their expertise. For all groups, there should be an expectation that they will report their progress clearly and succinctly in regular time intervals. Hypotheses should be generated, quickly debated, and either prioritized for analysis or eliminated as good initial candidates. These hypotheses should then be quickly restated as the tasks necessary to determine validity and handed out to the appropriate groups to work them, with times for results clearly communicated.

Someone on the call or in the crisis resolution meeting should be in charge, and that someone should be able to paint an accurate picture of the impact, the steps that have been tried, the best hypotheses being considered, and the timeline for completion of the current set of actions. Other members should be managers of the technical teams assembled to help solve the crisis and one of the experienced (described in organizations as senior, principal, or lead) technical people from each manager's teams. We will now describe these roles and positions in greater detail. Other engineers should be gathered in organizational or cross-functional groups to deeply investigate domain areas or services within the platform experiencing a crisis.

The Role of the Problem Manager

The preceding paragraphs have been leading up to a position definition. We can think of lots of names for such a position: outage commander, problem manager, incident manager, crisis commando, crisis manager, issue manager, and (from the military) battle captain. Whatever you call this person, you need someone who is capable of taking charge on the phone. Unfortunately, not everyone can fill this kind of a role. We aren't arguing that you need to hire someone just to manage your major production incidents to resolution (although if you have enough incidents, you might consider doing just that); rather, you should ensure that at least one person on your staff has the skills to manage such a chaotic environment.

The characteristics of the person who is capable of successfully managing chaotic environments are rather unique. As with leadership, some people are born with the

ability to make order out of chaos, whereas others build this skill over time. An even larger group of folks have neither the desire nor the skills to lead in a time of crisis. Putting these people into crisis leadership positions can be disastrous. The person leading the crisis team absolutely needs to be technically literate, but does not necessarily need to be the most technical person in the room. This individual should be able to use his technical base to formulate questions and evaluate answers relevant to the crisis at hand. The problem manager does not need to be the chief problem solver, but rather needs to effectively manage the process followed by the chief problem solvers gathered within the crisis. This person also needs to be incredibly calm “inside,” yet persuasive “outside.” This might mean that the problem manager has the type of presence to which people are naturally attracted, or it might mean that he isn’t afraid to yell to get people’s attention within the room or on the conference call.

The problem manager needs to be able to speak and think in business terms. He needs to be sufficiently conversant with the business model to make decisions in the absence of higher guidance on when to force incident resolution over attempting to collect data that might be destroyed and would be useful in problem resolution (remember the differences in definitions from Chapter 8). The problem manager also needs to be able to create succinct, business-relevant summaries from the technical chaos that’s occurring to keep the remainder of the business informed about the crisis team’s progress.

In the absence of administrative help to document the event, the problem manager is responsible for ensuring that the actions and discussions are represented in a written form for future analysis. Thus, the problem manager will need to keep a history of the crisis and ensure that others are keeping their own histories, to be merged later. A shared chat room with timestamps enabled is an excellent choice for this type of documentation.

In terms of *Star Trek* characters and financial gurus, the ideal problem manager is one-third Scotty, one-third Captain Kirk, and one-third Warren Buffet. He is one-third engineer, one-third manager, and one-third business manager. He has a combat arms military background, an MBA, and an MS in some engineering discipline. Clearly, it will be difficult to find someone with the right blend of experience, charisma, and business acumen to perform such a function. To make the task even harder, when you find the person, he probably will not want the job, because it is a bottomless pool of stress. However, just as some people enjoy the thrill of being an emergency room physician, some operators enjoy leading teams through technical crises.

Although we flippantly suggested the MBA, MS, and military combat arms background, we were only half kidding. Such people actually do exist! As we mentioned earlier, the military has a role for people who manage its battles or what most of us would view as crises. The military combat arms branches attract many leaders and managers who thrive on chaos and are trained and have the personalities to handle

such environments. Although not all former military officers have the right personalities, the percentage who do is significantly higher than in the rest of the general population. As a group, these leaders also tend to be highly educated, with many of them having at least one and sometimes multiple graduate degrees. Ideally, you would want someone who has proven himself both as an engineering leader and as a combat arms leader.

The Role of Team Managers

Within a crisis situation, a team manager is responsible for passing along action items to her teams and reporting progress, ideas, hypotheses, and summaries back to the crisis manager. Depending on the type of organization, the team manager may also be the “senior” or “lead” engineer on the call for her discipline or domain.

A team manager functioning solely in a management capacity is expected to manage her team through the crisis resolution process. The majority of this team will be located somewhere other than the crisis resolution (or “war”) room or on a call other than the crisis resolution call if a phone is being used. This means that the team manager must communicate and monitor the progress of the team as well as interact with the crisis manager. Although this may sound odd, the hierarchical structure with multiple communication channels is exactly what gives this process so much scale. This structured hierarchy affects scale in the following way: If every manager can communicate and control 10 or more subordinate managers or individual contributors, the capability in terms of human resources grows by orders of magnitude. The alternative is to have everyone communicate in a single room or through a single channel, which obviously doesn’t scale well. In such a scenario, communication becomes difficult and coordination of people becomes near impossible. People and teams quickly drown each other out in their debates, discussions, and chatter. Very little gets done in such a crowded environment.

Furthermore, this approach of having managers listen and communicate on two channels has proved very effective for many years in the military. Company commanders listen to and interact with their battalion commanders on one channel, and issue orders and respond to multiple platoon leaders on another channel (the company commander is in the upper-left portion of Figure 9.2). The platoon leaders then do the same with their platoons; each platoon leader speaks to multiple squads on a frequency dedicated to the platoon in question (see the center of Figure 9.2). Thus, although it might seem a bit awkward to have someone listen to two different calls or be in a room while issuing directions over the phone or in a chat room, this concept has worked well in the military since the advent of radio. It is not uncommon for military pilots to listen to four different radios at one time while flying the aircraft: two tactical channels and two air traffic control channels. In our consulting work, we have employed this approach successfully in several companies.

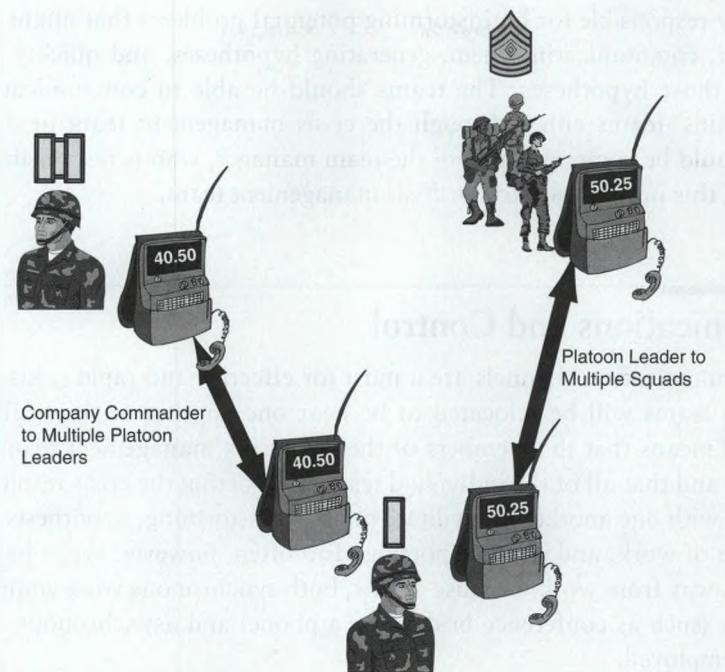


Figure 9.2 Military Communication

The Role of Engineering Leads

Each engineering discipline or engineering team necessary to resolve the crisis should have someone capable of both managing that team and answering technical questions placed within the higher-level crisis management team. This person serves as the lead individual investigator for her domain experience on the crisis management call and is responsible for helping the higher-level team vet information, clear ideas, and prioritize hypotheses. This person can also be on both the calls of the organization she represents and the crisis management call or conference, but her primary responsibility is to interact with the other senior engineers and the crisis manager to help formulate appropriate actions to end the crisis.

The Role of Individual Contributors

Individual contributors within the teams assigned to the crisis management call or conference communicate on separate chat and phone conferences or reside in separate conference rooms. They are responsible for generating and running down leads within their teams and work with the lead or senior engineer and their manager on the crisis management team. The individual contributor and his teams are

additionally responsible for brainstorming potential problems that might be causing the incident, communicating them, generating hypotheses, and quickly proving or disproving those hypotheses. The teams should be able to communicate with the other domains' teams either through the crisis management team or directly. All statuses should be communicated to the team manager, who is responsible for communicating this information to the crisis management team.

Communications and Control

Shared communication channels are a must for effective and rapid crisis resolution. Ideally, the teams will be relocated to be near one another at the beginning of a crisis. That means that the members of the lead crisis management team are in the same room and that all of the individual teams supporting the crisis resolution effort are located with one another to facilitate rapid brainstorming, hypothesis resolution, distribution of work, and status reporting. Too often, however, crises happen when people are away from work; because of this, both synchronous voice communication conferences (such as conference bridges on a phone) and asynchronous chat rooms should be employed.

The voice channel should be used to issue commands, stop harmful activity, and gain the attention of the appropriate team. It is absolutely essential that someone from each of the teams listens on the crisis resolution voice channel and simultaneously controls his or her team. In many cases, two representatives—the manager and the senior (or lead) engineer—should be present from each team on such a call. They serve as the command and control channel in the absence of everyone being in the same room. All shots are called from here, and this channel serves as the temporary change control authority and system for the company. The authority to do anything other than perform nondestructive “read” activities like investigating logs is first approved within this voice channel or conference room to ensure that two activities do not compete with each other and either cause system damage or result in an inability to determine which action “fixed” the system.

The chat or IRC channel is used to document all conversations and easily pass around commands to be executed so that time isn't wasted in communication. Commands that are passed around can be cut and pasted for accuracy. Additionally, the timestamps within the IRC or chat can be used in follow-up postmortem sessions. The crisis manager is responsible for ensuring not only that he puts his own notes in the chat room and writes his decisions in the chat room for clarification, but also that status updates, summaries, hypotheses, and associated actions are put into the chat room.

In our experience, it is absolutely essential that both the synchronous voice and asynchronous chat channels remain open and available during any crisis. The

asynchronous nature of chat allows activities to go on without interruption and allows individuals to monitor overall group activities between the tasks within their own assigned duties. Through this asynchronous method, scale is achieved while the voice channel allows for immediate command and control of different groups for immediate activities. Should everyone be in one room, there is no need for a phone call or conference call other than to facilitate experts who might not be on site and to give updates to the business managers. But even when everyone is present in the same room, a chat room should be opened and shared by all parties. In the case where a command is misunderstood, it can be buddy checked by all other crisis participants and even “cut and pasted” into the shared chat room for validation. The chat room allows actual system or application results to be shared in real time with the remainder of the group, and an immediate log with timestamps is generated when such results are cut and pasted into the chat.

The War Room

Phone conferences are a poor but sometimes necessary substitute for the “war room” or crisis conference room. So much more can be communicated when people are in a room together, as body language and facial expressions can be highly meaningful in a discussion. How many times have you heard someone say something, but when you read or looked at the person’s face you realized he was not convinced of the validity of his statement? Perhaps the person was not actually lying, but rather was passing along some information that he did not wholly believe. For instance, someone says, “The team believes that the problem could be with the login code,” but the scowl on her face shows that something is wrong. A phone conversation would not pick up on that discrepancy, but you have the presence of mind in person to question the team member further. She might answer that she doesn’t believe this scenario is possible given that the login code hasn’t changed in months, which might lower the priority for this hypothesis’s investigation. Alternatively, she might respond, “We just changed that damn thing yesterday,” which would increase the prioritization for investigation.

In the ideal case, the war room is equipped with phones, a large table space, terminals capable of accessing systems that might be involved in the crisis, plenty of work space, projectors capable of displaying key operating metrics or any person’s terminal, and lots of whiteboard space. Although the inclusion of a whiteboard might initially appear to be at odds with the need to log everything in a chat room, it actually supports chat activities by allowing graphics, symbols, and ideas best expressed in pictures to be drawn quickly and shared. These concepts can then be reduced to words and placed in chat, or a picture of the whiteboard can be taken and sent to the chat

members. Many new whiteboards even have systems capable of reducing their contents to pictures immediately. Should you have an operations center, the war room should be close to that to allow easy access from one area to the next.

You might think that creating such a war room would be a very expensive proposition. “We can’t possibly afford to dedicate space to a crisis,” you might say. Our answer is that the war room need not be expensive or dedicated to crisis situations. It simply needs to be given a priority in any crisis. As such, any conference room equipped with at least one and preferably two lines or more will do. Moreover, the war room is useful for the “ride along” situation described in Chapter 6. If you want to make a good case for why you should invest in creating a scalable organization, scalable processes, and a scalable technology platform, invite some business executives into a well-run war room to witness the work necessary to fix scale problems that result in a crisis. One word of caution here: If you can’t run a crisis well and make order out of its chaos, do not invite people into the fray. Instead, focus your time on finding a leader and manager who can run such a crisis and then invite other executives when you feel more confident about your crisis management system.

Tips for a Successful War Room

A good war room has the following:

- Plenty of whiteboard space
- Computers and monitors with access to the production systems and real-time data
- A projector for sharing information
- Phones for communication to teams outside the war room
- Access to IRC or chat
- Workspace for the number of people who will occupy the room

War rooms tend to get loud, and the crisis manager must maintain control within the room to ensure that communication is concise and effective. Brainstorming can and should be used, but limit communication during discussion to one individual at a time.

Escalations

Escalations during crisis events are critical for several reasons. The first and most obvious is that the company’s job in maximizing shareholder value is to ensure that it isn’t destroyed in these events. As such, the CTO, the CEO, and other executives

need to hear quickly of issues that are likely to take significant time to resolve or have a significant negative customer impact. In a public company, it's even more important that the senior execs know what is going on, because shareholders demand that they know about such things; indeed, public-facing statements may need to be made. Moreover, well-informed executives are more likely to be able to marshal *all* of the resources necessary to bring a crisis to resolution, including customer communications, vendors, partner relationships, and so on.

The natural tendency for engineering teams is to believe that they can solve the problem without outside help or without help from their management teams. That may be true, but solving the problem isn't enough—it needs to be resolved the quickest and most cost-effective way possible. Often, that will require more than the engineering team can muster on their own, especially if third-party providers are to blame for the incident. Moreover, communication throughout the company is important, because your systems are either supporting critical portions of the company or—in the case of Web companies—they *are* the company. Someone needs to communicate with shareholders, partners, customers, and maybe even the press. People who aren't involved in fighting are the best options to handle that communication.

Think through your escalation policies and get buy-in from senior executives *before* you have a major crisis. It is the crisis manager's job to adhere to those escalation policies and get the right people involved at the time defined in the policies, regardless of how quickly the problem is likely to be solved after the escalation.

Status Communications

Status communications should happen at predefined intervals throughout the crisis and should be posted or communicated in a somewhat secure fashion, so that the organizations needing information on resolution time can get the information they need to take the appropriate actions. Status communications differ from escalation. Escalation is undertaken to bring in additional help as time drags on during a crisis, whereas status communications are made to keep people informed. Using the RASCI framework, you escalate to R, A, S, and C personnel, and you post status communication to I personnel.

A status message should include the start time, a general update of actions since the start time, and the expected resolution time if known. This resolution time is important for several reasons. Perhaps you are supporting a manufacturing center, and the manufacturing manager needs to know if she should send her hourly employees home. Or perhaps you provide sales or customer support software in SaaS fashion, and those companies need to figure out what to do with their sales and customer support staff.

To: Crisis Manager Escalation List
Subject: September 22 Login Failures

Issue: 100% of Internet logins from our customers started failing at 9:00 AM on Thursday, 22 September. Customers who were already logged in could continue to work unless they signed out or closed their browsers.

Cause: Unknown at this time, but likely related to the 8:59 AM code push.

Impact: User activity metrics are off by 20% as compared to last week, and 100% of all logins from 9 AM have failed.

Update: We have isolated potential causes to one of three candidates within the code and we expect to find the culprit within the next 30 minutes.

Time to Restoration: We expect to isolate root cause in the code, build the new code and roll out to the site within 60 minutes.

Fallback Plan: If we are not live with a fix within 90 minutes we will roll the code back to the previous version within 75 minutes.

Johnny Onthespot
Crisis Manager
AllScale Networks

Figure 9.3 Status Communication

Your crisis process should clearly define who is responsible for communicating to whom, but it is the crisis manager's job to ensure that the timeline for communications is followed and that the appropriate communicators are properly informed. A sample status email is shown in Figure 9.3.

Crisis Postmortem and Communication

Just as a crisis is an incident on steroids, so a crisis postmortem is a juiced-up postmortem. Treat this postmortem with extra-special care. The systems that you helped create and manage have just caused a *huge* problem for a lot of people. This isn't the time to get defensive; it is the time to be reborn. Recognize that this meeting will fulfill or destroy the process of turning around your team, setting up the right culture, and fixing your processes.

Absolutely everything should be evaluated. The very first crisis postmortem is referred to as the "master postmortem," and its primary task is to identify subordinate postmortems. It is not intended to resolve or identify all of the issues leading to the incident, but rather to identify the areas that subordinate postmortems should address. For example, you might have postmortems focused on technology, process, and organization failures. You might have several postmortems on technology

covering different aspects—one on your communication process, one on your crisis management process, and one on why certain organizations didn't contribute appropriately early on in the postmortem.

Just as you had a communication plan during your crisis, so you must have a communication plan that remains in effect until all postmortems are complete and all problems are identified and solved. Keep all members of the RASCI chart updated, and allow them to update their organizations and constituents. This is a time to be completely transparent. Explain, in business terms, everything that went wrong and provide aggressive but achievable dates in your action plan to resolve all problems. Follow up with communication in your staff meeting, your boss's staff meeting, and/or the company board meeting. Communicate with everyone else via email or whatever communication channel is appropriate for your company. For very large events where morale might be impacted, consider conducting a company all-hands meeting, to be followed by weekly updates via email or on a blog.

A Note on Customer Apologies

When you communicate to your customers, buck the recent trend of apologizing without actually apologizing and try sincerity instead. Actually *mean* that you are sorry for disrupting their businesses, their work, and their lives! Too many companies use the passive voice, point the fingers in other directions, or otherwise misdirect customers as to true root cause. If you find yourself writing something like "Our company experienced a brief 6-hour downtime last week and we apologize for any inconvenience that this may have caused you," stop right there and try again. Try the first person "I" instead of the third person "we," drop the "may" and "brief," acknowledge that you messed up what your customers were planning on doing with your application, and get your message posted immediately.

It is very likely that your crisis will have significantly affected your customers. Moreover, this negative customer impact is not likely to have been the fault of the customer. Acknowledge your mistakes and be clear about what you plan to do to ensure that they do not happen again. Your customers will appreciate your forthrightness, and assuming that you can make good on your promises, you are more likely to have happy and satisfied customers over the long term.

Conclusion

Not every incident is created equally; some incidents require significantly more time to truly identify and solve all of the underlying problems. You should have a plan to handle such crises from inception to end. The end of the crisis management process is the point at which all problems identified through postmortems have been resolved.

The technology team is charged with responding to, resolving, and handling the problem management aspects of a crisis. The roles on this team include the problem manager/crisis manager, engineering managers, senior engineers/lead engineers, and individual contributor engineers from each of the technology organizations.

Four types of communication are necessary in crisis resolution and closure: internal communications, escalations, and status reports both during and after the crisis. Handy tools for crisis resolution may also be employed, including conference bridges, chat rooms, and the war room concept.

Teams who either don't see crises a lot or are new to the process should consider drilling in crisis management. Practice what each person would do so that when an actual crisis occurs—and it inevitably will—the team is amply prepared to manage it effectively.

Key Points

- Crises are incidents on steroids that can either make your company stronger or kill your business. Crises, if not managed aggressively, will destroy your company's ability to scale its customers, its organizational structure, and its technology platform and services.
- To resolve crises as quickly and cost-effectively as possible, you must contain the chaos with some measure of order.
- The leaders who are most effective in crises are calm on the inside but capable of forcing and maintaining order throughout the crisis management process. They must have business acumen and technical experience and be calm leaders under pressure.
- The crisis resolution team consists of the crisis manager, engineering managers, and senior engineers. In addition, teams of engineers reporting to the engineering managers are employed.
- The role of the crisis manager is to maintain order and follow the crisis resolution, escalation, and communication processes.
- The role of the engineering manager is to manage her team and provide status to the crisis resolution team.
- The role of the senior engineer from each engineering team is to help the crisis resolution team create and vet hypotheses regarding cause and to help determine the most appropriate rapid resolution approaches.
- The role of the individual contributor engineer is to participate in his team and identify rapid resolution approaches, create and evaluate hypotheses about the cause of the crisis, and provide status information to his manager on the crisis resolution team.

- Communication between crisis resolution team members should happen face to face in a crisis resolution or war room; when face-to-face communication isn't possible, the team should use a conference bridge on a phone. A chat room should also be employed.
- War rooms, which are ideally sited adjacent to operations centers, should be developed to help resolve crisis situations.
- Escalations and status communications should be defined during a crisis. After a crisis, the crisis process should provide status updates at periodic intervals until all root causes are identified and fixed.
- Crisis postmortems should be vigorous examinations that identify and manage a series of follow-ups in the form of subordinate postmortems that thematically attack all issues identified in the master postmortem.

Chapter 10

Controlling Change in Production Environments

If you know neither the enemy nor yourself, you will succumb in every battle.

—Sun Tzu

In engineering and chemistry circles, the word *stability* is defined as a resistance to deterioration, or alternatively as a constancy in makeup and composition. Something is “highly instable” if its composition changes regardless of the actual rate of activity within the system, and it is “stable” if its composition remains constant and it does not disintegrate or deteriorate. In the hosted services world, as well as with enterprise systems, one way to create a stable service is simply to not allow activity on it and to limit the number of changes made to the system. The term “changes,” in the previous sentence, is an indication of activities that an engineering team might take in reference to a system, such as modifying configuration files or updating a revision of code on the system. Unfortunately for many of us, the elimination of changes within a system, while potentially improving stability, will limit the ability of our businesses to grow. Therefore, we must not simply allow changes, but both enable and embrace changes while simultaneously implementing processes that manage the risk to availability that these changes present.

Our experience has taught us that a large portion of incidents in a production environment are caused or triggered by software and hardware changes. Without active change management with the intent of reducing risk, services will deteriorate or disintegrate (that is, become unavailable). It follows that you must manage change to ensure that you have a scalable service and happy customers.

Having processes that help you manage the effect of your changes is critical to your scalability success. These processes become even more important in a world of continuous delivery, where changes happen throughout the day and where short spans of time separate the ideation of an action and the implementation of that action. The

absence of any process to help manage the risk associated with change is a surefire way to cause both you and your customers a great deal of heartache. Thinking back to our “shareholder” test, can you really see yourself walking up to one of your largest shareholders and saying, “We will never log our changes or attempt to manage them, because doing so is a complete waste of time”? You’re highly unlikely to make such a statement, at least if you want to keep your job. With that in mind, let’s explore the world of change management and change identification.

What Is a Change?

Sometimes, teams define a change as any action that has the possibility of breaking a service, system, or application. The primary problem with this “classic” definition is its failure to acknowledge that change is necessary in any system that we hope to evolve and grow. The tone of the definition is pejorative in nature—it implies that a change is a risk and that risk is always bad. While change absolutely represents a risk, this is not inherently bad. Without risk, we cannot have a return on our investment. Put another way, there are no risk-free returns. Products that are lucky enough to experience rapid growth in acquired users or transactions require new servers and capacity to fulfill additional user and transaction demand. Product teams that are seeking to add functionality to increase user engagement need to add new services or modify existing services. Successful product teams may want to increase what they charge for their product or service. All of these actions are changes, and while they all carry some level of risk, they also represent some potential future value.

Our preferred definition of a change is agnostic to both the risk and the value creation elements inherent to said change. A *change* is any action you take to modify a product or service (including system data). Modifying the prices of products is a change, as is modifying the schema that supports those products in a relational database. Changes are actions you make in regard to a product or system to modify its behavior, enhance its value, or extend its capacity.

Changes include modifications in configuration, such as altering values used during startup or run time of your operating systems, databases, proprietary applications, firewalls, network devices, and so on. Changes also include any modifications to code, additions of hardware, removal of hardware, connection of network cables to network devices, and powering systems on and off. As a general rule, whenever one of your employees needs to touch, twiddle, prod, or poke any piece of hardware, software, or firmware, it is a change.

Change Identification

The very first thing you should do to limit the impact of changes is to ensure that each and every change that goes into your production environment gets logged with the following data:

- Exact time and date of the change
- System undergoing change
- Actual change
- Expected results of the change
- Contact information of the person making the change

An example of the minimum necessary information for a change log is included in Table 10.1. To understand why you should include all of the information from these five items, let's examine a hypothetical event with the AKF Partners Web site (www.akfpartners.com). While it is not a revenue-generating site, AKF Partners relies on this Web site to promote its services and bring in new clients. We also host a blog rich in content focused on helping current and future clients solve many pressing needs within their product teams. The AKF Partners Web site has login and client authentication functionality that allows clients access to white papers, proprietary research, and data collected on usage statistics for technology. Our internal definition of a crisis is anything greater than a 10% failure rate for any critical component (login is considered to be critical). When that threshold is exceeded, Mike Fisher, the AKF Partners crisis manager, is paged, and he starts to assemble the

Table 10.1 Example Excerpt from the AKF Partners Change Log

Date	Time	System	Change	Expected Results	Performed By
1/31/15	00:52	search02	Add watchdog.sh to init.d	Watchdog daemon starts on startup	mabbott
1/31/15	02:55	login01	Restart login01	Hung system restored to service	mfisher
1/31/15	03:10	search01	Install key_word.java	Install and start new service to count and report search terms to key_terms.out	mpaylor
1/31/15	12:10	db01	Add @autoextend to config.db	Tables automatically extend when out of space	tkeeven
1/31/15	14:20	lb02	Run syncmaster	Sync state from master load balancer	habbott

crisis management team with the composition discussed in Chapter 9. Once Mike has everyone assembled, what do you think should be the first question that he asks?

We often get answers to this query ranging from “What is going on right now?” to “How many customers are impacted?” to “What are the customers experiencing?” All of these are good questions and absolutely should be asked, but they are not the question that is most likely to reduce the duration and impact of an outage. The question Mike should ask first is “What changed most recently?” In our experience, changes are the leading cause of incidents in production environments.

Asking this question gets people thinking about what they did that might have caused the problem at hand. It focuses the team on attempting to quickly undo anything that is correlated in time to the beginning of the incident. It is the best opening question for any incident, from one with a small customer impact to a major crisis. It is a question focused on restoration of service rather than problem resolution.

One of the most humorous answers we encounter time and again after asking, “What changed most recently?” goes like this: “We just changed the configuration of the *blah* service but *that can't possibly be the cause of this problem!*” Collectively, we’ve heard this phrase hundreds, if not thousands, of times in our careers. From this experience, we can almost guarantee that when you hear this answer, you have found the cause of the incident. Focus on this change as the most likely cause! In our experience, the person might as well have said, “I caused this—sorry!” Okay, back to more serious matters.

Rarely will everyone who has made a change be present when you are troubleshooting an incident. As such, you really need a place to easily collect the information identified earlier. The system that stores this information does not need to be an expensive third-party change management and logging tool. It can just as easily be a shared email folder, with all changes identified in the subject line and sent to the folder at the time of the actual change by the person making the change. We have seen companies a few hundred people in size effectively use email folders to communicate and identify changes. Larger companies probably need more functionality, including a way to query the system by the subsystem being affected, the type of change, and so on. But *all* companies need a place to log changes so that they can quickly recover from those that have an adverse customer or stakeholder impact. This change identification log, wherever it may be, should provide the definitive answer to the “What changed most recently?” question.

Change Management

Change identification is a component of a much larger and more complex process called *change management*. The intent of change identification is to limit the impact

of any change by being able to determine its correlation in time to the start of an event and thereby its probability of causing that event. This limitation of impact increases your ability to scale, because it ensures that less time is spent working on value-destroying incidents. The intent of change management is to limit the probability of changes causing production incidents by controlling them through their release into the production environment and by logging them as they are introduced to production. Great companies implement change management to increase the rate of change, thereby benefiting from the value inherent in change while minimizing the associated risks.

Change Management and Air Traffic Control

One useful metaphor for change management is that of the Federal Aviation Administration (FAA)'s Air Traffic Control (ATC) system. ATC exists to reduce the frequency and impact of aircraft accidents during takeoff, landing, and taxiing at airports, just as change management exists to reduce the frequency and impact of change-related incidents within your platform, product, or service.

ATC works to order aircraft landings and takeoffs based on the availability of the aircraft, its personal needs (e.g., does the aircraft have a declared emergency, is it low on fuel), and its order in the queue for takeoffs and landings. Queue order may be changed for a number of reasons, including the aforementioned declaration of emergencies.

Just as ATC orders aircraft for safety, so the change management process orders changes for safety. Change management considers the expected delivery date of a change, its business benefit, the risk associated with the change, and its relationship with other changes, all in an attempt to maximize value creation while minimizing change-related incidents and conflicts.

Change identification is a point-in-time action, where someone indicates a change has been made and moves on to other activities. By comparison, change management is a life-cycle process whereby changes are

- Proposed
- Approved
- Scheduled
- Implemented and logged
- Validated as successful
- Reviewed and reported on over time

The change management process may start as early as when a project is going through its business validation (or return on investment analysis), or it may start as late as when a project is ready to be moved into the production environment. Change management also includes a process of continual process improvement whereby metrics regarding incidents and resulting impact are collected to improve the change management process.

Note that none of these activities, from proposal to reporting, needs to be a manual process. In the world of continuous integration and continuous delivery, each can be an automated task. For instance, once a product change (e.g., a code commit) has passed all appropriate tests, the continuous delivery framework can auto-approve and schedule the change so that it does not conflict with other changes. The system can then implement the change, perform automatic validation, and (of course) log the exact time of the change. Some of our clients extend this concept to implement “standard changes” that may include such activities as configuration changes and capacity adds. These changes are common and repeatable tasks, but still bear some risk and, therefore, must be logged for identification. The use of the word “process” here does not indicate human intervention; in fact, each step can be automated.

Continuous Delivery

Continuous delivery (CD) is an increasingly popular approach within many online product teams. In an ideal scenario, granular changes in functionality, bug fixes, or a few lines of code are checked in by developers to a version control system and shepherded through a build phase, assessed in an automated acceptance test phase, and, upon successful completion of all tests, released in an automated fashion to the site or service.

The theory behind the adoption of CD is that smaller, more frequent releases represent a lower risk to a production environment than larger, more complex releases (see Chapter 16, Determining Risk). Because the probability of a failure increases with size, complexity, and effort, smaller releases are less likely in isolation to cause failures. Because the solution is automated, manual failures in process—such as forgetting to log the time of a release—are almost completely avoided.

Proper practice of CD has other benefits for many organizations. To practice CD, organizations must maintain a high percentage of code coverage with their automated test suites. Tracking of release times and correlation to incidents become easier to automate. Developers often find themselves being more productive, because they don't waste time in waiting for feedback from change management teams. Finally, developers are incented to properly consider the overall risk in their changes and take more accountability for the benefits and negative impacts of their individual work effort.

The one exception to this is the need to review the efficacy and impact of changes over time. While our process may be completely automated, as in the case of a well-designed continuous delivery process, it still must be evaluated to ensure it is meeting our organization's needs. Which improvements can we make? Where is the continuous delivery process not meeting our end objectives? Where can we make tweaks to more effectively manage our risk and maximize value creation?

Change Management and ITIL

The Information Technology Infrastructure Library (ITIL) defines the goal of change management as follows:

The goal of the Change Management Process is to ensure that standardized methods and procedures are used for efficient and prompt handling of all changes, in order to minimize the impact of change-related incidents upon service quality, and consequently improve the day-to-day operations of the organization.

Change management is responsible for managing the change process, involving the following elements:

- Hardware
- Communications equipment and software
- System software
- All documentation and procedures associated with the running, support, and maintenance of live systems

The ITIL is a great source of information to consult should you decide to implement a robust change management process as defined by a recognized industry standard. For our purposes, we will describe a lightweight change management process that should be considered for any small- or medium-sized enterprise.

Change Proposal

As described earlier, the proposal for a change can occur anywhere in your change management cycle. The IT Service Management (ITSM) and ITIL frameworks hint at identification occurring as early in the cycle as the business analysis for a change. Within these frameworks, the change proposal is called a *request for change*. Opponents to ITSM actually cite the inclusion of business/benefit analysis within the change process as one of the reasons that the ITSM and ITIL are not good frameworks. These critics insist that the business benefit analysis and feature/product

selection steps have nothing to do with managing change. Although we agree that these are two separate processes, we also believe that a business benefit analysis should be performed at some point. If business benefit analysis isn't conducted as part of another process, including it within the change management process is a good first step. That said, this book deals with scalability and not with product and feature selection, so we will simply state that such a benefit analysis should occur somewhere.

The most important point to remember regarding a change proposal is that it kicks off all other activities. Ideally, it will occur early enough to allow some evaluation as to the impact of the change and its relationship with other changes. For the change to actually be “managed,” we need to know certain things about the proposed change:

- The system, subsystem, and component being changed
- The expected result of the change
- How the change is to be performed
- Known risks associated with the change
- The relationship of the change to other systems and recent or planned changes

You might decide to track significantly more information than this, but we consider the preceding items to be the minimum information necessary to properly plan change schedules. As we indicated earlier, none of these needs to be manual in nature and none of the data elements fly in the face of a continuous delivery processes. It is easy, upon submitting a product for integration, testing, and deployment, to identify each of these areas within the continuous delivery framework.

The system undergoing change is important because we hope to limit the number of changes to a given system during a single time interval. Consider that a system is the equivalent of a runway at an airport. We don't want two changes colliding on the same system because if a problem arises during the change, we won't immediately know which change caused it. Similarly, we want the ability to monitor and measure the results of each change independently of the others. As such, the change management process or continuous delivery system needs to know the item being changed, down to the granularity of what is actually being modified. For instance, if we are making a software change and a single large executable or script contains 100% of the code for that subsystem, we need only identify that we are changing out that executable or script. In contrast, if we are modifying one of several hundred configuration files, we should identify exactly which file is being modified. If we are changing a file, configuration, or software on an entire pool of servers with similar functionality, the pool is the most granular thing being changed and should be so identified.

Architecture here plays a huge role in helping us increase change velocity. If our technology platform comprises a number of discrete services, we have increased the number of venues in which changes have been made—equivalent to increasing the number of airports or runways on which we can land planes in our aircraft analogy. The result is a greater throughput of total changes: Aircraft now have more landing slots. If the services communicate asynchronously, we would have a few more concerns, but we are also likely more willing to take risks. Conversely, if the services communicate synchronously, there isn't much more fault tolerance than is available with a monolithic system (see Chapter 21, Creating Fault-Isolative Architectural Structures). If Service A and Service B communicate synchronously to produce a result, we can't change them at exactly the same time because then we would have no clue which change was proximate to a potential incident. Returning to the aircraft analogy, landing slots are not increased for change management. The expected result of the change is important because we want to be able to verify later that the change was successful. For instance, if a change is being made to a Web server and is intended to allow more threads of execution in the Web server, we should state that as the expected result. If we are making a modification to our proprietary code to correct an error where the capital letter "Q" shows up as its hex value 51, we should indicate that outcome is intended.

Information regarding how the change is to be performed will vary with your organization and system. You may need to indicate precise steps if the change will take some time or requires a lot of work. For instance, if a server needs to be stopped and rebooted, that might affect which other changes can be happening at the same time. Providing the steps for the change also allows you to reproduce and improve the process when you need to repeat the change or roll out a similar change in the future, which will both reduce the time it takes to prepare the change the next time and reduce the risk of missing a step. The larger and more complex the steps for the change in production, the more you should consider requiring those steps to be clearly outlined.

Identifying the known risks of the change is an often-overlooked step. Very often, requesters of a change will quickly type in a commonly used risk to speed through the change request process. A little time spent in this area could pay huge dividends in avoiding a crisis. For example, if there is the risk that data corruption may occur if a certain database table is not "clean" or truncated prior to the change, this possibility should be pointed out during the change. The more risks that are identified, the more likely it is that the change will receive the proper management oversight and risk mitigation, and the higher the probability that the change will be successful. We cover risk identification and management in Chapter 16, Determining Risk, in much greater detail. With automated continuous delivery processes and systems, identification of a risk level or risk value may help the system determine how many

changes can be applied in any given interval. We describe this process and consider how it applies to automated continuous delivery systems in the “Change Scheduling” section later in this chapter.

Complacency often sets in quickly with these processes, and teams are quick to assume that identifying risks is simply a “check the box” exercise. A great way to incent the appropriate behaviors and to get your team to analyze risks is to reward those team members who identify and avoid risks and to counsel those who have incidents that occur outside of the risk identification limits. This isn’t a new technique, but rather the application of tried-and-true management techniques. Another great tactic may be to show the team data from your environment indicating how changes have resulted in incidents. Remind the team that a little time spent managing risks can save a lot of time wasted on managing incidents.

Depending on the process that you ultimately develop, you may or may not decide to include a required or suggested date for your change to take place. We highly recommend developing a process that allows individuals to suggest a date and time; however, the approving and scheduling authorities should be responsible for deciding on the final date based on all other changes, business priorities, and risks. In the case of continuous delivery, suggested dates may be considered by the solution in identifying the optimal risk level for any given change interval.

Change Approval

Change approval is a simple portion of the change management process. Your approval process may simply be a validation that all of the required information necessary to “request” the change is indeed present (or that the change proposal has all required fields filled out appropriately). To the extent that you’ve implemented some form of the RASCI model, you may also decide to require that the appropriate A, or owner of the system in question, has signed off on the change and is aware of it. Another lightweight step that may expedite the approval process is to include a peer review or peer approval. In that case, the peer may be asked to review the steps for completeness and to help identify missing dependencies. The primary reason for the inclusion of this step in the change control process is to validate that everything that should happen prior to the change occurring has, in fact, happened. This is also the place at which changes may be questioned with respect to their priority relative to other changes.

An approval here is not a validation that the change will have the expected results; it simply means that everything has been discussed and that the change has met with the appropriate approvals in all other processes prior to rolling out to the system, product, or platform. Bug fixes, for instance, may have an abbreviated approval process compared to a complete reimplementation of the entire product, platform, or system. The former is addressing a current issue and might not require the approval

of any organization other than QA, whereas the latter might require a final sign-off from the CEO.

Continuous delivery systems may auto-approve most types of changes but place others (e.g., data definition [DDL] changes, schema modifications to complex databases) on hold based on a predefined category or the risk level limit. Because QA is usually completely automated in such systems, the QA approval is achieved upon successful completion of the automated regression suite, including any new tests checked into the test suite for the purposes of testing the new functionality. Other major risk factors are evaluated by the algorithm codified within the continuous delivery solution, with major concerns kicked out for manual review, and all others scheduled for introduction into the production environment.

Change Scheduling

The process of scheduling changes is where most of the additional benefit of change management occurs relative to the benefit obtained when you implement change identification. This is the point where the real work of the “air traffic controllers” comes in. That is, a group tasked with ensuring that changes do not collide or conflict applies a set of rules identified by its management team to maximize change benefit while minimizing change risk.

The business rules will likely limit changes during peak utilization of your platform or system. If you experience the heaviest utilization rates between 10 a.m. and 2 p.m., it probably doesn’t make sense to roll out your largest and most disrupting changes during this time frame. In fact, you might limit or eliminate altogether changes during this time frame if your risk tolerance is low. The same might hold true for specific times of the year. Sometimes, though, as in very high-volume change environments, we simply don’t have the luxury of disallowing changes during certain portions of the day and we need to find ways to manage our change risks elsewhere.

The Business Change Calendar

Many businesses, from large to small, put the next three to six months’ and maybe even the next year’s worth of proposed changes into a shared calendar for internal viewing. This concept helps communicate changes to various organizations and often reduces the risks of changes as teams start requesting dates that are not full of changes already. Consider implementing the change calendar concept as part of your change management system. In very small companies, a change calendar may be the only thing you need to implement (along with change identification).

This set of business rules might also include an analysis of the type of risk discussed in Chapter 16, Determining Risk. We are not arguing for an intensive analysis of risk or even indicating that your process absolutely needs to have risk analysis. Rather, we are stating that if you can develop a high-level, easy-to-use mechanism for risk analysis of the change, your change management process will be more robust and likely yield better results. Each change might include a risk profile of, say, high, medium, and low during the change proposal portion of the process. The company then might decide that it wants to implement no more than 3 high-risk changes, 6 medium-risk changes, and 20 low-risk changes during the same week. Assigning risk levels as simple as high, medium, and low also makes it easy to correlate incidents to risk levels and, therefore, to evaluate and improve your risk assessment process. Obviously, as the number of change requests increases over time, the company's willingness to accept more risk on any given day within any given category will need to increase; otherwise, changes will back up in the queue and the time to market to implement any change will increase. One way to help both limit risk associated with change and increase change velocity is to implement fault-isolative architectures described in Chapter 21.

Another consideration during the change scheduling portion of the process might be the beneficial business impact of the change. This analysis ideally will be done in some other process, rather than being performed first for the benefit of change. That is, someone, somewhere will have decided that the initiative requiring the change will be of benefit to the company. If you can then represent that analysis in a light-weight way within the change process, it is very likely to inform better decisions around scheduling. If the risk analysis measures the product of the probability of failure multiplied by the effect of failure, the benefit assessment would then analyze the probability of success along with the impact of such success. The company would be incented to move as many high-value activities to the front of the queue as possible, while remaining wary not to starve lower-value changes.

An even better process would be to implement both processes, with each recognizing the other in the form of a cost-benefit analysis. Risk and reward might offset each other to create some value that the company determines, with guidelines being established to implement changes in any given day based on a risk-reward tradeoff between two values. We'll cover the concepts of risk and benefit analysis in Chapter 16.

Key Aspects of Change Scheduling

Change scheduling is intended to minimize conflicts and reduce change-related incidents. Key aspects of most scheduling processes are as follows:

- Change blackout times/dates during peak utilization or revenue generation
- Analysis of risk versus reward to determine priority of changes

- Analysis of relationships of changes for dependencies and conflicts
- Determination and management of maximum risk per time period or number of changes per time period to minimize probability of incidents

Change scheduling need not be burdensome. Indeed, it can be contained within another meeting and in small companies can be quick and easy to implement without additional headcount.

Change Implementation and Logging

Change implementation and logging is basically the function of implementing the change in a production environment in accordance with the steps identified within the change proposal and consistent with the limitations, restrictions, or requests identified within the change scheduling phase. This phase consists of two steps: logging the start time of the change and logging the completion time of the change. This is slightly more robust than the change identification process identified earlier in the chapter, but also will yield greater results in a high-change environment. If the change proposal does not include the name of the individual performing the change, the change implementation and logging steps should explicitly identify the individuals associated with the change. Clearly automated CD systems can do this easily, creating a great change management log to help aid in incident and problem management.

Change Validation

No process is complete without verification that you accomplished what you expected to accomplish. While this should seem intuitively obvious to the casual observer, how often have you asked yourself, “Why the heck didn’t Sue check that before she said she was done?” That question follows us outside the technology world and into everything in our life: The electrical contractor completes the work on your new home, but you find several circuits that don’t work; your significant other says that his portion of the grocery shopping is done, but you find five items missing; the systems administrator claims that he is done with rebooting and repairing a faulty system, but your application still doesn’t work.

Our point here is that you shouldn’t perform a change unless you know what you expect to get from that change. In turn, if you do not get that expected result, you should consider undoing the change and rolling back or at least pausing and discussing the alternatives. Maybe you made it halfway to where you want to be if it was a tuning change to help with scalability, and that’s good enough for now.

Validation becomes especially important in high-scalability environments. If your organization is a hyper-growth company, we highly recommend adding a *scalability validation* to every significant change. Did you change the load, CPU utilization, or memory utilization for the worse on any critical systems as a result of your change?

If so, does that put you in a dangerous position during peak utilization/demand periods? The result of validation should either be an entry indicating when validation was complete by the person making the change, a rollback of the change if it did not meet the validation criteria, or an escalation to resolve the question of whether to roll back the change.

In CD systems, the payload delivery (also known as rollout or automated release) component may be augmented to include test scripts that validate the new system similar to how it was validated in the unit test suite. Additionally, DevOps or operations personnel should see key performance indicators identifying when new changes were delivered to the runtime environment. Such indicators may be plotted along the x -axis of the metrics and monitors that these personnel watch in real time so that they can correlate system behavior changes with new changes in the environment.

The Case for Rollback Plans

Oftentimes we find that companies with change management processes do not have rollback plans as part of their change proposals. This is common whether the company is small and uses a very lightweight change management process or the company is large and has established very mature processes. All too often, we find that change implementers have either not thought through or not documented the steps to rollback—or even worse, taken the approach of “It’s very difficult, if we run into a problem, we will fix it going forward.”

The ability to roll back a change quickly is a critical component when the goal is to have a highly available and scalable product or service. If a change does not produce the desired outcome, or if an incident occurs as a result of a change, once the change has been isolated as the cause of the incident, the first reaction should be to roll it back and restore the service. You can always fix the problem and try again. Too many companies spend hours and even days trying to fix changes in a production environment—often because they had not planned how they would roll back a change.

Advantages of having rollback plans:

- Allows operations and incident teams to focus on restoring service.
- Forces engineers to think through and document all the steps required to roll back. This is particularly useful when an incident manifests itself well after the change—for example, in the middle of the night or a few days later—when the engineer who implemented the change may not be available.
- Creates opportunities to automate rollout and rollback procedures, especially for similar changes. Both will speed up the time to restore your service in the event of an incident.
- Gives you more flexibility to roll out changes gradually to see the results, such as cases where configurations or software changes are released to a pool of servers.

Change Review

The change management process should include a periodic review of its effectiveness. As explained in Chapter 5, Management 101, you simply cannot improve that which you do not measure. Key metrics to analyze during the change review include the following items:

- Number of change proposals submitted
- Number of successful change proposals (without incidents)
- Number of failed change proposals (without incidents but changes were unsuccessful and didn't make it to the validation phase)
- Number of incidents resulting from change proposals
- Number of aborted changes or changes rolled back due to failure to validate
- Average time to implement a proposal from submission

Obviously, we are looking for data indicating the effectiveness of our process. If we have a high rate of change but also a high percentage of failures and incidents, something is definitely wrong with our change management process; in addition, something is likely wrong with other processes, our organization, and maybe our architecture. Aborted changes, on the one hand, should be a source of pride for the organization, confirming that the validation step is finding issues and keeping incidents from happening. On the other hand, they also serve as a source of future corrections to process or architecture, when the primary goal should be to have a successful change.

The Change Control Meeting

We've referred several times to a meeting wherein changes are approved and scheduled. ITIL and ITSM refer to such meetings and gatherings of people as the change control board or change approval board. Whatever you decide to call it, we recommend a regularly scheduled meeting with a consistent set of people. It is absolutely okay for this to be an additional responsibility for several individual contributors and/or managers within your organization; oftentimes, having a diverse group of folks from each of your technical teams and even some of the business teams leads to the most effective reviewing authority possible.

Depending on your rate of change, you should consider holding such a meeting once a day, once a week, or once a month. If the change implementers use diligence when filling out their change proposals with all required fields, the meetings can

be short and efficient. Attendees ideally will include representatives from each of your technical organizations and at least one team outside of technology that can represent the business or customer needs. Typically, the head of the infrastructure or operations teams “chairs” the meeting, as he or she most often has the tools needed to review change proposals and completed or failed changes.

The team should have access to the database wherein the change proposals and completed changes are stored. It should also have a set of guidelines by which it analyzes changes and attempts to schedule them for production. Some of these guidelines were discussed previously in this chapter.

With automated delivery systems, such as exist within CD environments, meetings can occur less frequently and focus on changes that were rejected by the automated system. In such environments, meetings may also address the efficacy of the CD solution with an eye toward tweaking the system to allow for more risk, added control parameters, and other modifications.

Part of the change control meetings, on a somewhat periodic basis, should be a review of the change control process using the metrics we’ve identified. It is absolutely acceptable to augment these metrics. Where necessary, postmortems should be scheduled to analyze failures of the change control process. These postmortems should be run consistently with the postmortem process identified in Chapter 8, Managing Incidents and Problems. The output of these postmortem sessions should be tasks to correct issues associated with the change control process, or information that feeds into requests for architecture changes or changes to other processes.

Continuous Process Improvement

Besides the periodic internal review of the change control process identified in the “Change Control Meeting” section, you should implement a quarterly or annual review of the change control process. Are changes taking too long to implement as a result of the process? Are change-related incidents increasing or decreasing as a percentage of total incidents? Are risks being properly identified? Are validations consistently performed and consistently correct? As with any other process, the change control process should not be assumed to be correct, or to stay correct forever. Although it might work well for a year or two given some rate of change within your environment, as your organization grows in complexity, rate of change, and rate of transactions, this process very likely will need tweaking to continue to meet your needs. As discussed in Chapter 7, Why Processes Are Critical to Scale, no process is right for every stage of your company’s life span.

Change Management Checklist

Your change management process should, at a minimum, include the following phases:

- Change proposal (the ITIL request for change [RFC])
- Change approval
- Change scheduling
- Change implementation and logging
- Change validation
- Change review

Your change management meeting should include representatives from all teams within technology and members of the business responsible for working with your customers or stakeholders.

Your change management process should have a continual process improvement loop that helps drive changes to the change management process as your company and its needs mature. This process also drives changes to other processes, organizations, and architectures as they are identified with change metrics.

In completely automated environments, the CD solution controls most of the changes, and the change management meeting is more focused on improving the efficacy of the CD solution as well as approving or modifying changes rejected by CD.

Conclusion

Change identification is a very lightweight process for very young and small companies. It is powerful in that it can help limit the negative impact on customers when changes go badly. However, as companies grow and their rate of change grows, they often need a much more robust process that more closely approximates an air traffic control system.

Change management is a process whereby a company attempts to take control of its changes. This process serves to safely manage changes, not to slow the change process down. Change management processes can vary from lightweight processes that simply attempt to schedule changes and avoid change-related conflicts to very mature processes that attempt to manage the total risk–reward tradeoff on any given day or hour within a system. As your company grows and as its need to manage change-associated risks grows, you will likely move from a simple change

identification process to a very mature change management process that takes into consideration risk, reward, timing, and system dependencies.

Key Points

- A change happens whenever any employee needs to touch, twiddle, prod, or poke any piece of hardware, software, or firmware.
- Change identification is an easy process for young or small companies focused on being able to find recent changes and roll them back in the event of an incident.
- At a minimum, an effective change identification process should include the exact time and date of the change, the system undergoing change, the expected results of the change, and the contact information of the person making the change.
- The intent of change management is to limit the impact of changes by controlling them through their release into the production environment and by logging them as they are introduced to production.
- Change management consists of the following phases or components: change proposal, change approval, change scheduling, change implementation and logging, change validation, and change efficacy review.
- The change proposal kicks off the process and should contain, at a minimum, the following information: the system or subsystem being changed, the expected result of the change, information on how the change is to be performed, known risks, known dependencies, and relationships to other changes or subsystems.
- The change proposal in more advanced processes may also contain information regarding risk, reward, and suggested or proposed dates for the change.
- The change approval step validates that all information is correct and that the person requesting the change has the authorization to make the change.
- The change scheduling step is the process of limiting risk by analyzing dependencies, assessing rates of changes in subsystems and components, and minimizing the risk of an incident. Mature processes will include an analysis of risk and reward.
- The change implementation step is similar to the change identification lightweight process, but includes the logging of start and completion times within the changes database.
- The change validation step ensures that the change had the expected result. A failure here might trigger a rollback of the change, or an escalation if a partial benefit is achieved.

- The change review step is the change management team's internal review of the change process and the results. It looks at data related to rates of changes, failure rates, impact to time to market, and so on.
- At the change control meeting, changes are approved, scheduled, and reviewed after their implementation. This meeting is typically chaired by the head of operations and/or infrastructure and has as its members participants from each engineering team and customer-facing business teams.
- The change management process should be reviewed by teams outside the change management team to determine its efficacy. A quarterly or annual review is appropriate and should be performed by the CTO/CIO and members of the executive staff of the company.
- For companies that practice continuous delivery, the notion of change management is codified within the automated system responsible for automated releases. Risk analysis, change identification, change approval, and level loading of risk are inherent within the CD solution.

Chapter 11

Determining Headroom for Applications

Knowing the place and the time of the coming battle, we may concentrate from the greatest distances in order to fight.

—Sun Tzu

The last thing that anyone wants is for his or her company to be remembered for or associated with some type of failure. The authors of this book remember this feeling all too well from our early days at eBay, when the company was the poster child for technical glitches. More recently, at least into 2012, the best-known story of this type has been about Twitter and the infamous “Fail Whale.” In the early days of Twitter, when it would experience significant problems, the service would display the picture of a whale being lifted out of the water by tiny birds, along with the statement “Twitter is over capacity.” This event seemed to happen so often that the Fail Whale took on a following all its own, with people making money selling clothing with the image of the giant mammal and tiny birds.¹ Failing for reasons of under-capacity, as the Fail Whale’s statement would seem to imply, is completely avoidable with the right processes and appropriate funding. Unfortunately, besides giving Twitter a great deal of unwanted press, this type of event remains one of the most common failures we see in our consulting practice. And for the record, it’s happened to us as well.

This chapter walks you through the process of determining headroom for your product. We start with a brief discussion of the purpose of headroom and an exploration of where it is used. Next, we describe how to determine the headroom of some common components found in systems. Lastly, we discuss the ideal conditions that you want to look for in your components in terms of load or performance.

1. Sarah Perez. “How an Unknown Artist’s Work Became a Social Media Brand Thanks to the Power of Community.” July 17, 2008. http://readwrite.com/2008/07/17/the_story_of_the_fail_whale.

Purpose of the Process

The purpose of determining the headroom of your product is to understand where your product is from a capacity perspective relative to future expected demand. Headroom answers the question, “How much capacity do I have remaining before I start to have issues?” Headroom calculations come in handy during several phases of the product development cycle.

For instance, annual budget planning should be informed by an understanding of your product headroom. To understand how much equipment (e.g., network gear, compute nodes, database licenses, storage) you need to buy, you must first understand both the forecasted demand and your current capacity (or headroom). Without this understanding, you are just making an educated guess with equal probability of over-purchasing or (gasp) under-purchasing. Many organizations do a rough, back-of-the-envelope calculation by saying, for example, they grew $x\%$ this year and spent \$ y , so therefore if they expect to grow $x\%$ again next year, they should spend \$ y again. Although this passes as a planned budget in many organizations, it is guaranteed to be wrong. Not taking into account different types of growth, existing headroom capacity, and optimizations, there is no way your projections could be accurate other than by pure luck.

Hiring is another area in which headroom calculations are useful. Without understanding headroom and future product growth expectations, how do you know how many people with each skill set (e.g., software developers, database administrators) you will need in your organization? If you understand that you have plenty of headroom on your application servers and on your database but you are bumping up against the bandwidth capacity of your firewalls and load balancers, you might want to add another network engineer to the hiring plan instead of another systems administrator.

Headroom is also useful in product planning. As you design and plan for new features during your product development life cycle, you should be considering the capacity requirements for these new features. If you are building a brand-new service, you will likely want to run it on its own pool of servers. If this feature is an enhancement of another service, you should consider its ramifications for the headroom of the current servers. Will the new feature require the use of more memory, larger log files, intensive CPU operations, the storage of external files, or more SQL calls? Any of these factors can impact the headroom projections for your entire application, from network to database to application servers.

Understanding headroom is critical for scalability projects. You need a way to prioritize your scalability and technical debt projects. In the absence of such a prioritization mechanism, “pet projects” (those without clearly defined value creation) will bubble to the top. The best (and, in our mind, only) way of prioritizing projects

is to use a cost–benefit analysis. The cost is the estimated time in engineering and operations effort to complete the project. The benefit is the increase in headroom or scale that the projects will bring—sometimes it is measured in terms of the amount of business or transactions that will fail should you not make the investment. After reading through the chapter on risk management, you might want to add a third comparison—namely, risk. How risky is the project in terms of impact on customers, completion within the timeline, or impact on future feature development?

By incorporating headroom data into your budgeting, planning, and prioritization decisions, you will start making much more data-driven decisions and become much better at planning and predicting.

Structure of the Process

The process of determining your product’s headroom is straightforward but takes effort and tenacity. Getting it right requires research, insight, and calculations. The more attention to detail that you pay during each step of the process, the better and more accurate your headroom projections will be. You already have to account for unknown user behaviors, undetermined future features, and many more variables that are not easy to pin down. Do not add more variation by cutting corners or getting lazy.

The first step in the headroom process is to identify the major components of the product. Infrastructure such as network gear, compute nodes, and databases needs to be evaluated on a granular level. If you have a service-oriented architecture and different services reside on different servers, treat each pool separately. A sample list might look like this:

- Account management services application servers
- Reports and configuration services application servers
- Firewalls
- Load balancers
- Bandwidth (intra- and inter-data center or colocation connectivity as well as public transit/Internet links)
- Databases, NoSQL solutions, and persistence engines

After you have reduced your product to its constituent components, assign someone to determine the usage of each component over time, preferably the past year, and the maximum capacity in whatever is the appropriate measurement. For most components, multiple measurements will be necessary. The database measurements, for example, would include the number of SQL transactions (based on the current

query mix), the storage, and the server loads (e.g., CPU utilization, “load”). The individuals assigned to handle these calculation tasks should be the people who are responsible for the health and welfare of these components whenever possible. The database administrators are most likely the best candidates for the database analysis; the systems administrators would be the best choice to measure the capacity and usage of the application servers. Ideally, these personnel will use system utilization data that you’ve collected over time on each of the devices and services in question.

The next step is to quantify the growth of the business. This data can usually be gathered from the general manager of the appropriate business unit or a member of the finance organization. Business growth is typically made up of many parts. One factor is the natural or intrinsic growth—that is, how much growth would occur if nothing else was done to the product or by the business (no deals, no marketing, no advertising, and so on) except basic maintenance. Such measurements may include the rate at which new users are signing on and the increase or decrease in usage by existing users. Another factor is the expected increase in growth caused by business activities such as developing new or better features, marketing, or signing deals that bring more customers or activities.

The natural (organic) growth can be determined by analyzing periods of growth without any business activity explanations. For instance, if in June the application experienced a 5% increase in traffic, but there were no deals signed in the prior month or release of customer-facing features that might explain this increase, we could use this amount as our organic growth. Determining the business activity growth requires knowledge of the planned feature initiatives, business department growth goals, marketing campaigns, increases in advertising budgets, and any other similar metric or goal that might potentially influence how quickly the application usage will grow. In most businesses, the business profit and loss (P&L) statement, general manager, or business development team is assigned goals to meet in the upcoming year in terms of customer acquisition, revenue, or usage. To meet these goals, they put together plans that include signing deals with customers for distribution, developing products to entice more users or increase usage, or launching marketing campaigns to get the word out about their fabulous products. These plans should have some correlation with the company’s business goals and can serve as the baseline when determining how they will affect the application in terms of usage and growth.

After you have a very solid projection of natural and human-made growth, you can move on to understanding the seasonality effect. Some retailers receive 75% of their revenue in the last 45 days of the year due to the holiday season. Some experience the summer doldrums, as people spend more time on vacation and less time browsing sites or purchasing books. Whatever the case for your product, you should take this cycle into account to understand what point of the seasonality curve you are on and how much you can expect this curve to raise or lower demand for your products and services. If you have at least one year’s worth of data, you can begin

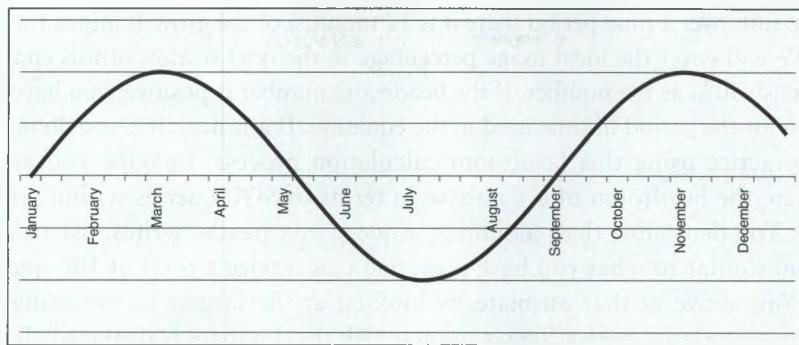


Figure 11.1 Seasonality Trend

projecting seasonal differences. The way to accomplish this is to strip out the average growth from the numbers and see how the traffic or usage changed from month to month. You are looking for a sine wave or something similar to Figure 11.1.

Now that you have seasonality data, growth data, and actual usage data, you need to determine how much headroom you are likely to retrieve through your scalability initiatives next year. Similar to the way we used the business growth for customer-facing features, you need to determine the amount of headroom that you will gain by completing *scalability projects*. These scalability projects may include splitting a database or adding a caching layer. For this purpose, you can use various approaches such as historical gains from similar projects or multiple estimations by several architects, just as you would when estimating effort for story points (similar to the practice of “pokerizing” in Agile methods). When organized into a timeline, these projects will indicate a projected increase in headroom throughout the year. Sometimes, projects have not been identified for the entire next 12 months; in that case, you would use an estimation process similar to that applied to business-driven growth. Use historical data to estimate the most likely outcome of future projects weighted with an amount of expert insight from your architects or chief engineers who best understand the system.

The last step is to bring all the data together to calculate the headroom. The formula for doing so is shown in Figure 11.2.

This equation states that the headroom of a particular component of your system is equal to the ideal usage percentage of the maximum capacity minus the current usage

$$\text{Headroom} = (\text{Ideal Usage Percentage} \times \text{Maximum Capacity}) - \text{Current Usage}$$

$$-\sum_{t=1}^{12} (\text{Growth}(t) - \text{Optimization Projects}(t))$$

Figure 11.2 Headroom Equation

minus the sum over a time period (here it is 12 months) of the growth minus the optimization. We will cover the ideal usage percentage in the next section of this chapter; for now, let's use 50% as the number. If the headroom number is positive, you have enough headroom for the period of time used in the equation. If it is negative, you do not.

Let's practice using this headroom calculation process. Imagine you are asked to calculate the headroom of a database in terms of SQL queries within your environment. You determine that, assuming a query mix (reads, writes, use of indexes, and so on) similar to what you have now, you can service a total of 100 queries per second. You arrive at that estimate by looking at the largest constraining factors within the server and storage arrays upon which the database is deployed. Today the database processes 25 queries per second during peak utilization. Based on information you receive from your business unit leader, you determine that organic and business-driven growth will likely result in an additional 10 queries per second over the course of the year during peak transaction processing time. As a result of examining past transaction rates over the course of the last year, you know that there is a great deal of seasonality in your business and that peak transaction times tend to happen near the holiday season. You and your team further decide that you can reduce the queries per second on the database by 5 through a combination of query elimination and query tuning.

To evaluate your headroom, you plug these numbers into the headroom equation shown in Figure 11.2, using 0.5 as the ideal usage percentage. This results in the following expression, where q/s stands for queries per second and tp is the time period—which in our case is 1 year or 12 months:

$$\text{Headroom q/s} = 0.5 \times 100 \text{ q/s} - 25 \text{ q/s} - (10 - 5) \text{ q/s/tp} \times 1 \text{ tp}$$

Reducing this equation results in the following:

$$\text{Headroom q/s} = 50 \text{ q/s} - 25 \text{ q/s} - 5 \text{ q/s} = 20 \text{ q/s}$$

Since the number is positive, you know that you have enough headroom to make it through the next 12 months.

But what does the headroom number 20 q/s mean? Strictly speaking, and relative to your ideal usage factor, you have 20 queries per second of spare capacity. Additionally, this number, when combined with the summation clause (growth, seasonality, and optimization over the time period), tells the team how much time it has before the application runs out of headroom. The summation clause is shown in Figure 11.3.

$$\text{Headroom Time} = \text{Headroom} / \sum_{t=1}^{12} (\text{Growth}(t) - \text{Optimization Projects})$$

Figure 11.3 Headroom Time Equation

Solving this equation, Headroom Time = $20 \text{ q/s} + 5 \text{ q/s/tp} = 4.0 \text{ tp}$. Because the time period is 12 months or 1 year, if your expected growth stays constant (10 q/s per year), then you have 4 years of headroom remaining on this database server.

There are a lot of moving parts within this equation, including hypotheses about future growth. As such, the calculation should be run on a periodic basis.

Ideal Usage Percentage

The ideal usage percentage describes the amount of capacity for a particular component that should be planned for usage. Why not 100% of capacity? There are several reasons for not planning to use every spare bit of capacity that you have in a component, whether that is a database server or load balancer. The first reason is that you might be wrong. Even if you base your expected maximum usage on testing, no matter how well you stress test a solution, the test itself is artificial and may not provide accurate results relative to real user usage. We'll cover the issues with stress testing in Chapter 17, Performance and Stress Testing. Equally possible is that your growth projections and improvement projections may be off. Recall that both of these are based on educated guesses and, therefore, will vary from actual results. Either way, you should leave some room in the plan for being off in your estimates.

The most important reason that you do not want to use 100% of your capacity is that as you approach maximum usage, unpredictable things start happening. For example, thrashing—the excessive swapping of data or program instructions in and out of memory—may occur in the case of compute nodes. Unpredictability in hardware and software, when discussed as a theoretical concept, is entertaining, but when it occurs in the real world, there is nothing particularly fun about it. Sporadic behavior is a factor that makes a problem incredibly difficult to diagnose.

Thrashing

As one example of unpredictable behavior, let's look at thrashing or excessive swapping. You are probably familiar with the concept, but here's a quick review.

Almost all operating systems have the ability to swap programs or data in and out of memory if the program or data that is being run is larger than the allocated physical memory. Some operating systems divide memory into pages, and these pages are swapped out and written to disk. This capability to swap is very important for two reasons. First, some items used by a program during startup are accessed very infrequently and should be removed from active memory. Second, when the program or data set is larger than the physical memory, swapping the needed parts into memory makes the execution much faster. This speed difference between disk and

memory is actually what causes problems. Memory is accessed in nanoseconds, whereas disk access is typically measured in milliseconds. The difference is thousands of times slower.

Thrashing occurs when a page is swapped out to disk but is soon needed and must be swapped back in. After it is in memory, the page gets swapped back out so as to let something else have the freed memory. The reading and writing of pages to disk is very slow compared to memory; therefore, the entire execution begins to slow down while processes wait for pages to land back in memory. Lots of factors influence thrashing, but closing in on the limits of capacity on a machine is a very likely cause.

What is the ideal percentage of capacity to use for a particular component? The answer depends on a number of variables, with one of the most important being the type of component. Certain components—most notably networking gear—have highly predictable performance as demand ramps up. Application servers are, in general, much less predictable in comparison. This isn't because the hardware is inferior, but rather reflects its nature. Application servers can and usually do run a wide variety of processes, even when dedicated to single services. Therefore, you may decide to use a higher percentage in the headroom equation for your load balancer than you do on your application server.

As a general rule of thumb, we like to start at 50% as the ideal usage percentage and work up from there as the arguments dictate. Your app servers are probably your most variable component, so someone could argue that the servers dedicated to application programming interface (API) traffic are less variable and, therefore, could run higher toward the true maximum usage, perhaps 60%. Then there is the networking gear, which you might feel comfortable running at a usage capacity as high as 75% of maximum. We're open to these changes, but as a guideline, we recommend starting at 50% and having your teams or yourself make the case for why you should feel comfortable running at a higher percentage. We would not recommend going above 75%, because you must account for error in your estimates of growth.

Another way that you can arrive at the ideal usage percentage is by using data and statistics. With this method, you figure out how much variability resides in your services running on the particular component and then use that as a guide to buffer away from the maximum capacity. If you plan to use this method, you should consider revisiting these numbers often, but especially after releases with major features, because the performance of services can change dramatically based on user behavior or new code. When applying this method, we look at weeks' or months' worth of performance data such as load on a server, and then calculate the standard deviation of that data. We then subtract 3 times the standard deviation from the maximum

Table 11.1 Load Averages

Mon	Tue	Wed	Thu	Fri	Sat	Sun
5.64	8.58	9.48	5.22	8.28	9.36	4.92
8.1	9.24	6.18	5.64	6.12	7.08	8.76
7.62	8.58	5.6	9.02	8.89	7.74	6.61

capacity and use that in the headroom equation as the substitute for the Ideal Usage Percentage \times Maximum Capacity.

In Table 11.1, we provide three weeks of maximum load values for our application servers. The standard deviation for this sample data set is 1.49. If we take 3 times that amount, 4.48, and subtract the maximum load capacity that we have established for this server class, we then have the amount of load capacity that we can plan to use up to but not exceed. In this case, our systems administrators believe that 15 is the maximum; therefore, $15 - 4.48 = 10.5$ is the maximum amount we can plan for. This is the number that we would use in the headroom equation to replace the Ideal Usage Percentage \times Maximum Capacity.

Headroom Calculation Checklist

Follow these steps when completing a headroom calculation:

1. Identify major components.
2. Assign responsibility for determining actual usage and maximum capacity.
3. Determine intrinsic or natural growth.
4. Determine business activity-based growth.
5. Determine peak seasonality effects.
6. Estimate headroom or capacity reclaimed by infrastructure projects.
7. Perform the headroom calculation:
 - If the result is positive, you have sufficient capacity for the time frame analyzed.
 - If the result is negative, you do *not* have sufficient capacity over the specified time frame.
8. Divide headroom by the (growth + seasonality – optimizations) value to get the amount of time remaining to use up the capacity.

A Quick Example Using Spreadsheets

Let's show a quick example, through tables and a diagram, of how our headroom calculations can be put into practice. For this example, we'll use a modified spreadsheet that one of our readers, Karl Arao, put together and distributed on Twitter. For the sake of brevity, we'll leave out the math associated with the headroom calculation and just show how the tables might be created to provide an elegant graphical depiction of headroom.

Karl was focused on defining the headroom in terms of CPU utilization for his organization. He first did all the legwork to determine business growth and ran some tests. The result was that he expected, based on the existing CPU utilization rate of 20% and the business unit expectations of growth, among other factors, that CPU utilization for the devices in question would grow at approximately 3.33% per month. This growth, as described within our equation, is an absolute growth in percentage, not compounded or relative growth to past months. Karl assigned an ideal usage of 80% to his devices and created a table like that shown in Figure 11.4. He calculated the values in this table based on three years of transactions, but we have eliminated several rows—you'll likely get the picture of the type of information you need from the truncated version.

With this table in hand, it's quite easy to plot the expected growth in transactions against the ideal utilization and determine our time to live visually. Figure 11.5 is taken from Karl Arao's public work and modified for the purposes of this book. We can readily see in this figure that we hit our ideal limit in June 2015.

The spreadsheet method of representation also makes it easy to use compounded rates of growth. For instance, if our general manager indicated that the number of transactions is likely to grow 3% month over month (MOM), and if we knew that

Growth on an absolute basis, so 3.33% means CPU goes up by exactly 3.33% per month

Remaining headroom to 80% ideal usage

Effect of baseline 20% plus summation of absolute growth per month

Abs CPU % Growth	Month	Initial Usage + ABS Growth	Headroom
3.33%	Jan-14	23%	57%
3.33%	Feb-14	27%	53%
3.33%	Mar-14	30%	50%
3.33%	Apr-14	33%	47%
3.33%	May-14	37%	43%
3.33%	Jun-14	40%	40%
...			
3.33%	Nov-14	137%	-57%
3.33%	Dec-14	140%	-60%

Figure 11.4 Headroom Calculation Table

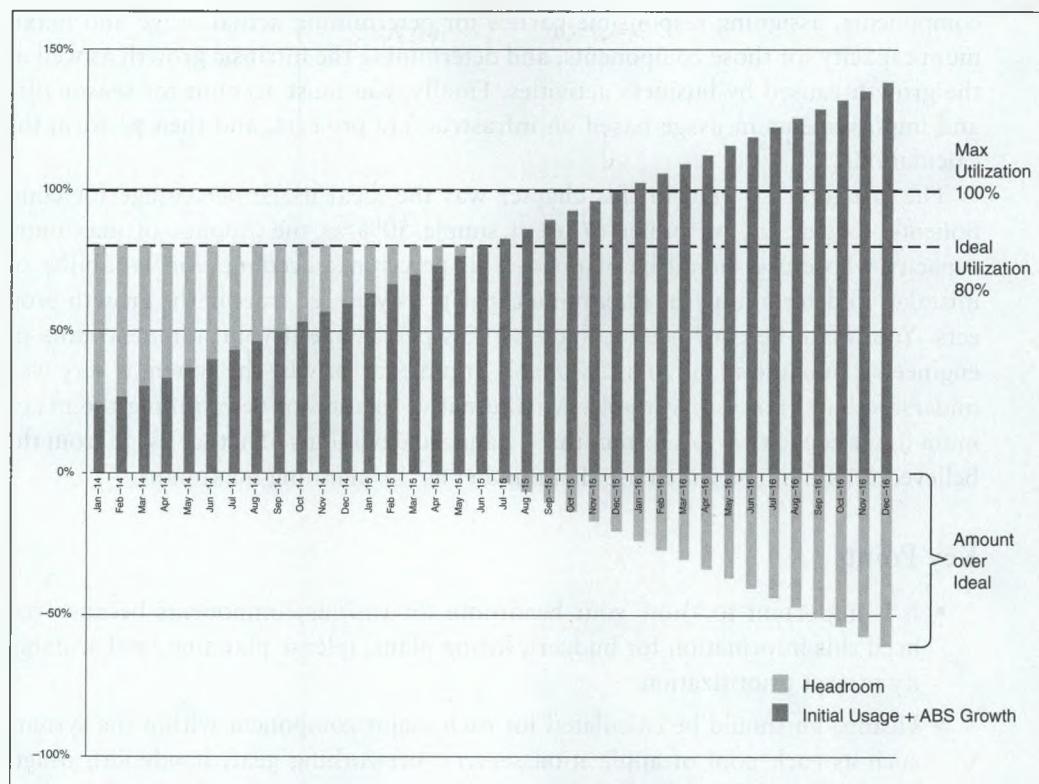


Figure 11.5 Headroom Graph

each transaction would have roughly the same impact on CPU utilization (or queries, or something else), we could modify the spreadsheet to compound the rate of growth. The resulting graph would have a curve in it, moving upward at increasing speeds as each month experiences significantly greater transactions than the prior month.

Conclusion

In this chapter, we started our discussion by investigating the purpose of the headroom process. There are four principal areas where you should consider using headroom projections when planning: budgets, headcount, feature development, and scalability projects.

The headroom process consists of many steps, which are highly detail oriented, but overall it is very straightforward. The steps include identifying major

components, assigning responsible parties for determining actual usage and maximum capacity for those components, and determining the intrinsic growth as well as the growth caused by business activities. Finally, you must account for seasonality and improvements in usage based on infrastructure projects, and then perform the calculations.

The last topic covered in this chapter was the ideal usage percentage for components. In general, we prefer to use a simple 50% as the amount of maximum capacity whose use should be planned. This percentage accounts for variability or mistakes in determining the maximum capacity as well as errors in the growth projects. You might be convinced to increase this percentage if your administrators or engineers can make sound and reasonable arguments for why the system is very well understood and not very variable. An alternative method of determining the maximum usage capacity is to subtract three standard deviations of actual usage from the believed maximum and then use that number as the planning maximum.

Key Points

- It is important to know your headroom for various components because you need this information for budgets, hiring plans, release planning, and scalability project prioritization.
- Headroom should be calculated for each major component within the system, such as each pool of application servers, networking gear, bandwidth usage, and database servers.
- Without a sound and factually based argument for deviating from this rule of thumb, we recommend not planning on using more than 50% of the maximum capacity on any one component.

Chapter 12

Establishing Architectural Principles

He wins his battles by making no mistakes. Making no mistakes is what establishes the certainty of victory, for it means conquering an enemy that is already defeated.

—Sun Tzu

Corporate “values” statements are often posted in workplaces and serve to remind employees of the desired behavior and culture within their companies. In this chapter, we present the scalability analog to these corporate values—*architectural principles*. These principles both guide the actions of technology teams and form the criteria against which technology architectures are evaluated.

Principles and Goals

Upon reviewing the high-level goal tree that was originally defined in Chapter 5, Management 101, and reproduced in Figure 12.1, you might recall that the tree indicates two general themes: the creation of more monetization opportunities and greater revenue at a reduced cost base. These themes are further broken out into topics such as quality, availability, cost, time to market (TTM), and efficiency. Furthermore, these topics are each assigned specific SMART goals.

Ideally, our architectural principles will be based on our high-level goal tree topics and *not* on specific goals. Our goals are likely to change over time, so our principles should broadly support both future goals and current ones.

Our clients often find it useful to begin a principle-setting session by brainstorming. For example, you might start by taking the goal tree from Figure 12.1 and writing each theme on a whiteboard. Ensure that the goal tree stays prominently displayed throughout the session for reference. For each topic (e.g., quality, availability, cost, TTM, and efficiency), focus on finding broad principles that can serve

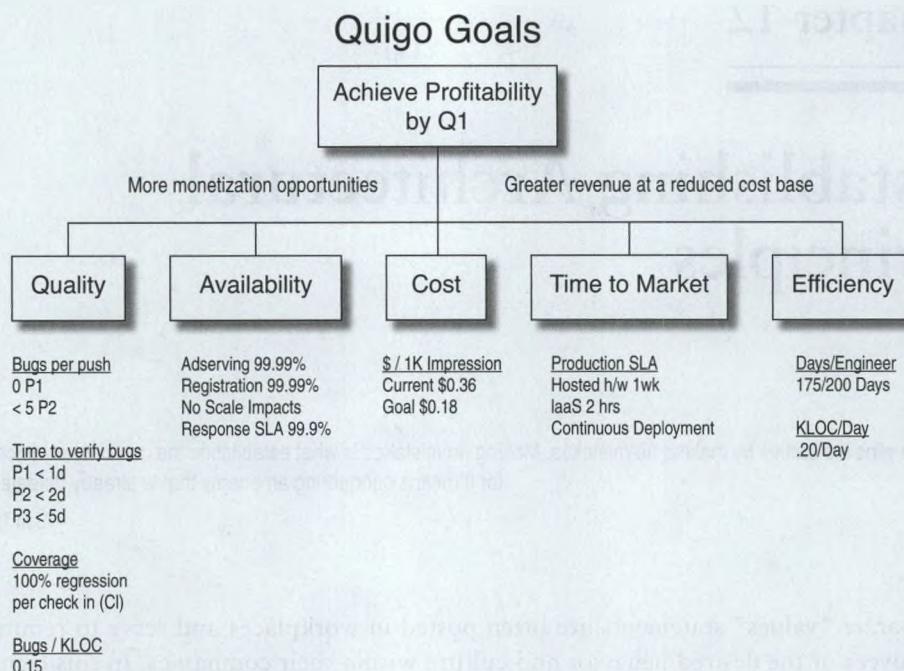


Figure 12.1 Quigo Goal Tree

as a “test” for any given initiative—that is, principles with which you can determine whether a suggested architectural design change meets the selected criteria. Principles that are so lofty and nebulous that they cannot be used practically in a design or architecture review (e.g., “design to be scalable”) won’t assist in driving cost-effective, scalable, and highly available architectures. Likewise, principles that are aspirational in nature but are often cast aside in practice in favor of expediency or for other reasons are not useful.

When we facilitate these sessions, we keep our clients focused on principle exploration—writing down as many principles as possible under each topic. Discussion is valuable, because each person should be able explain the meaning of a proposed principle. Oftentimes these discussions improve the phrasing of the principle itself. For instance, a principle that is presented as “horizontal scalability” may start a discussion about whether that principle is meant to keep the team from implementing “vertical scalability” (the ability to use a larger solution or server). The team may then decide to change the title “horizontal scalability” to “scale out, not up” to show the desire to limit vertical scalability wherever possible in favor of horizontal scalability.

When conducting an exercise similar to the one described here, an experientially diverse team is important. Recall from Chapter 3, Designing Organizations, that

experiential diversity increases the level of cognitive conflict, and hence the innovation and strategic value of the results. Brainstorming sessions, then, should include members with backgrounds in software development, general architecture, infrastructure, DevOps, quality assurance, project management, and product ownership. Questions that appear initially driven by a session participant's lack of experience in a particular area frequently uncover valuable insights.

It is important that each of the candidate principles embody as many of the SMART characteristics discussed in Chapter 4, Leadership 101, as possible. One notable exception is that principles are not really "time bounded." When speaking of architectural principles, the "T" in the SMART acronym means the principle can be used to "test" a design and validate that it meets the required intent. Lofty nebulous principles don't meet this criterion.

One often-used lofty—and therefore unfortunate—architectural principle is "infinitely scalable." Infinite scalability is simply not achievable and would require infinite cost. Furthermore, this principle does not allow for testing. (How would you test against a principle that by definition has no limit?) No deployment architecture can meet the defined goal. Finally, we must consider the role of principles in guiding organizational work habits. Principles should ideally help inform design and indicate a path toward success. Lofty and aspirational principles like "infinite design" don't offer a map to a destination, making them useless as a guide.

Good Principles Are . . .

Principles should influence both the behavior and the culture of the team. They should help guide designs and be used in testing to determine if those designs meet the goals and needs of the company. Effective principles are tightly aligned with the company's goals, vision, and mission. A good principle has the following characteristics:

- Specific. A principle should not be confusing in its wording.
- Measurable. Words like "infinite" should not be included in a principle.
- Achievable. Although principles should be inspirational, they should be capable of being achieved in both design and implementation.
- Realistic. The team should have the capabilities of meeting the objective. Some principles are achievable but not with the time or talent you have available.
- Testable. A principle should be amenable to use in "testing" a design and validating that it meets the intent of the principle.

Make sure your principles follow the SMART guidelines.

Principle Selection

We can all cite examples where we worked on a problem for an extended period of time but just couldn't get to the right solution, and then magically, with just a bit of sleep, the solution came to us. Brainstorming exercises often follow a similar pattern. For that reason when facilitating principle-setting sessions, we like to perform the principle exploration exercise (the whiteboard exercise described previously) on day 1 and the principle selection exercise on day 2. These exercises do not need to take a full day each. For small companies, a few hours each day may suffice. For larger and more complex companies, given the high value that the principles will return, it makes sense to allocate more time and effort to these activities.

At the beginning of the second day, review the principles you wrote on the whiteboard and have the team recap the discussions from the previous day. One way to do so is to have participants present each of the principles on the board and invite a brief discussion of its merits. Kicking off the second day in this fashion will stimulate the collective memory of the team and jumpstart the value creation effort for the remainder of the day.

When selecting principles (during the latter portion of the second day), we like to use a crowd sourcing technique. Our approach is to allow each of the participants in the meeting to vote for a constrained number of principles. Using this technique, each participant walks to the whiteboard and casts up to his or her quota of votes. Principles are best kept to an easily memorized and easy-to-use number (say, 8 to 15); we generally limit votes per person to roughly half the number of desired principles. Having fewer than 8 principles isn't likely to cover the entire spectrum necessary to achieve the desired effect. Conversely, having more than 15 principles tends to be too cumbersome for reviews and makes it too hard to remember the principles as means to drive daily efforts. Choosing half the number of principles as a vote quota usually makes it easy to narrow the number of candidate principles to a manageable size.

After the team voting is complete, you will likely end up with clear winners and a number of principles that are "ties" or are close in terms of total votes. It is common to have more than your desired number of principles, in which case you might decide to invite some discussion and debate (cognitive conflict) to narrow the list down to your desired number. Once the voting is finished, it's time to think of your "principle go to market" strategy.

We like to present the principles in a fashion that is easily distributed on a single page and that makes explicit the causal roadmap between the principle and what it is expected to achieve. One way to do so is to bundle the principles into the areas they are expected to support in a Venn diagram. We are big fans of Venn diagrams for presenting principles because so many of the principles are likely to affect multiple areas.

Returning to the earlier example, Quigo's goal tree topics include efficiency, time to market, availability, cost, and quality. When developing the principles at Quigo,

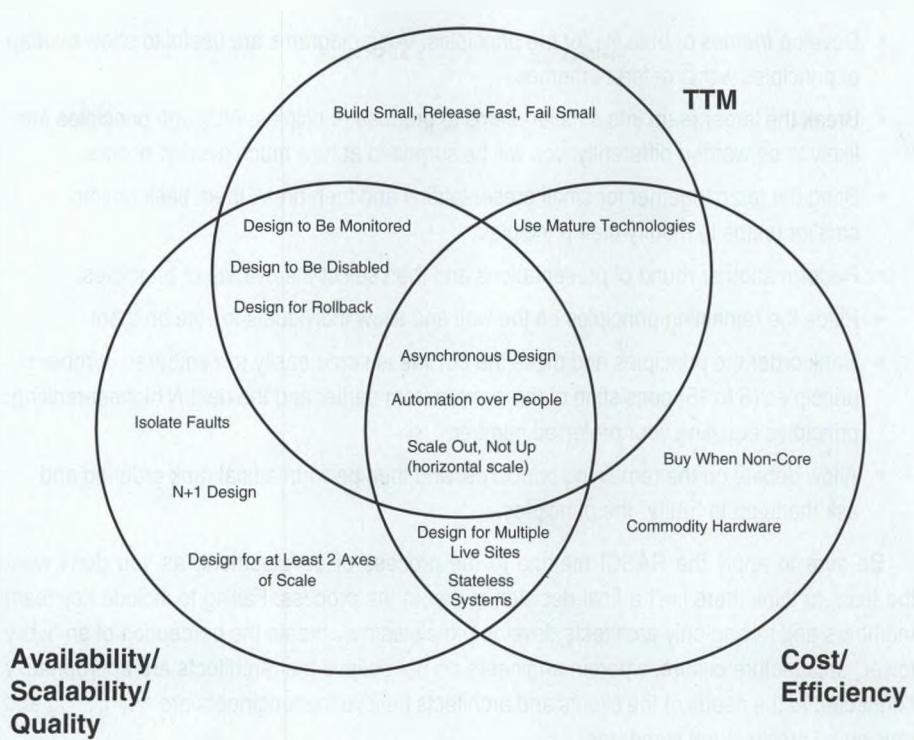


Figure 12.2 *Architecture Principles Venn Diagram*

we decided that efficiency and cost were tightly correlated and could be presented within one region of the Venn diagram (higher levels of efficiency are correlated with lower costs). While scalability was not part of the goal tree, it is highly correlated with availability and deserved a place in our diagram. Quality is also clearly highly correlated with availability; thus, quality, availability, and scalability are presented together. With these relationships, we reduced our Venn diagram to three regions: availability/scalability/quality, cost/efficiency, and TTM (time to market). Figure 12.2 shows a potential outcome of such a Venn diagram plot.

Engendering Ownership of Principles

You want the team to “own” the architectural principles that guide your scalability initiatives. The best way to achieve this is to involve them in the development and selection of the principles. A good process to follow in establishing principles involves these steps:

- Ensure representation from all parties who would apply the principles.
- Provide context for the principles by tying them into the vision, mission, and goals of the organization and the company. Create a *causal map to success*.

- Develop *themes* or buckets for the principles. Venn diagrams are useful to show overlap of principles within different themes.
- Break the larger team into smaller teams to propose principles. Although principles are likely to be worded differently, you will be surprised at how much overlap occurs.
- Bring the team together for small presentations and then break them back up into smaller teams to modify their principles.
- Perform another round of presentations and then select the overlap of principles.
- Place the remaining principles on the wall and allow individuals to vote on them.
- Rank order the principles and make the cut line at some easily remembered number of principles (8 to 15) consisting of the overlap from earlier and the next N highest-ranking principles equaling your preferred number.
- Allow debate on the remaining principles, and then perform a final rank ordering and ask the team to “ratify” the principles.

Be sure to apply the RASCI method to the process of development, as you don't want the team to think there isn't a final decision maker in the process. Failing to include key team members and having only architects develop principles may create the perception of an “ivory tower” architecture culture, wherein engineers do not believe that architects are appropriately connected to the needs of the clients and architects believe that engineers are not owning and abiding by architectural standards.

AKF's Most Commonly Adopted Architectural Principles

In this section, we introduce the 15 most commonly adopted architectural principles from our client base. Many times during engagements, we will “seed” the architectural principle gardens of our clients with these 15 principles. We then ask them to conduct their own process, taking as many as they like, discarding any that do not work, and adding as many as needed. We also update this list if our clients come up with an especially ingenious or useful principle. The Venn diagram shown in Figure 12.2 depicts our principles as they relate to scalability, availability, and cost. We will discuss each of the principles at a high level and then dig more deeply into those that are identified as having an impact on scalability.

N + 1 Design

Simply stated, this principle expresses the need to ensure that anything you develop has at least one additional instance of that system in the event of failure. Apply the

rule of three (which we will discuss in Chapter 32, Planning Data Centers), or what we sometimes call *ensuring that you build one for you, one for the customer, and one to fail*. This principle holds true for everything from large data center design to Web services implementations.

Design for Rollback

This is a critical principle for Web services, Web 2.0, or Software as a Service (SaaS) companies. Whatever you build, ensure that it is backward compatible. In other words, make sure that you can roll it back if you find yourself in a position of spending too much time “fixing it forward.” Some companies offer that they can roll back within a specific window of time—say, the first few hours after a change is made. Unfortunately, some of the worst and most disastrous failures won’t show up for several days, especially when those failures involve customer data corruption. In the ideal case, you will design to allow something to be rolled, pushed, or deployed while your product or platform is still “live.” The rollback process is covered in more detail in Chapter 18, Barrier Conditions and Rollback.

Design to Be Disabled

When designing systems, especially very risky systems that communicate to other systems or services, make them capable of being “marked down” or disabled. This will give you additional time to “fix forward” or ensure that your system doesn’t go down as a result of a bug that introduces strange, out-of-bounds demand characteristics to the system.

Design to Be Monitored

Monitoring, when done well, goes beyond the typical actions of identifying services that are alive or dead, examining or polling log files, collecting system-related data over time, and evaluating end-user response times. When done well, applications and systems are designed from the ground up to be if not self-healing, then at least self-diagnosing. If a system is designed to be monitored and logs the correct information, you can more easily determine the headroom remaining for the system and take the appropriate action to correct scalability problems earlier.

For instance, at the time of design of any system, you know which services and systems it will need to interact with. Perhaps the service in question repeatedly makes use of a database or some other data store. Potentially, the service makes a call, preferably asynchronously, to another service. You also know that from time to time you will be writing diagnostic information and potentially errors to some sort of volatile or stable storage system. You can apply all of this knowledge to design

a system that can give you more information about future scale needs and thereby increase the system's availability.

We argue that you should build systems that will help you identify potential or future issues. Returning to our example system and its calls to a database, we should log the response time of that database over time, the amount of data obtained, and maybe the rate of errors. Rather than just reporting on that data, our system could be designed to show "out of bounds" conditions plotted from a mean of the last 30 Tuesdays (assuming today is Tuesday) for our five-minute time of day. Significant standard deviations from the mean could be "alerted" for future or immediate action depending on the value. This approach leverages a control chart from statistical process control.

We could do the same with our rates of errors, the response time from other services, and so on. We could then feed this information into our capacity planning process to help us determine where demand versus supply problems might arise. In turn, we can identify which systems are likely candidates for future architectural changes.

Design for Multiple Live Sites

Having multiple sites is a must to assure your shareholders that you can weather any geographically isolated disaster or crisis. The time to start thinking about strategies for running your data centers isn't when you are deploying those data centers, but rather when you are designing them. All sorts of design tradeoffs will impact whether you can easily serve data out of more than one geographically dispersed data center while your system is live. Does your application need or expect that all data will exist in a monolithic database? Does your application expect that all reads and writes will occur within the same database structures? Must all customers reside within the same data structures? Are other services called in synchronous fashion, and are they intolerant to latency?

Ensuring that your product designs allow services to be hosted in multiple locations and operate independently of one another is critical to achieving rapid deployment. Such designs also allow companies to avoid the constraints of power and space within a single facility and enable cloud infrastructure deployments. You may have an incredibly scalable application and platform, but if your physical environment and your operating contracts keep you from scaling, you are just as handicapped as if you had nearly infinite space and a platform that needs to be re-architected for scale. Scalability is about more than just system design; it is about ensuring that the business environment in which you operate, including contracts, partners, and facilities, is also scalable. Therefore, your architecture must allow you to make use of several facilities (both existing and potentially new) on an on-demand basis.

Use Mature Technologies

We all love to play with and implement the newest and sexiest technologies available. Oftentimes the use of such technology can be useful in helping to reduce our costs, decrease time to market, decrease costs of development, increase ease of scalability, and decrease end-user response times. In many cases, new technology can even create a short-lived competitive advantage. Unfortunately, new technology, by definition, also tends to have a higher rate of failure; thus, if it is implemented in critical portions of your architecture, it may result in significant hits to availability. When availability is important within a solution or service, you should base it on technology that is proven and reliable.

In many cases, you may be tempted by the competitive edge promised by a new vendor technology. Be cautious here: As an early adopter, you will also be on the leading edge of finding bugs with that software or system. If availability and reliability are important to you and your customers, try to be an early majority or late majority adopter of systems that are critical to the operation of your service, product, or platform. Implement newer technology for new features that are less critical to the availability of your solution, and port that technology to mission-critical areas once you've proven it can reliably handle the daily traffic volume.

Asynchronous Design

Simply put, synchronous systems have a higher failure rate than those designed to act asynchronously. Furthermore, the scalability of a synchronous system is limited by the slowest and least scalable system in the chain of communications. If one system or service slows, the entire chain slows, lowering throughput. Thus, synchronous systems are more difficult to scale in real time.

Asynchronous systems are far more tolerant of slowdowns. As an example, consider the case where a synchronous system can serve 40 simultaneous requests. When all 40 requests are in flight, no more can be handled until at least one completes. In contrast, asynchronous systems can handle the new request and do not block for the response. They have a service that waits for the response while handling the next request. Although throughput is roughly the same, they are more tolerant to slowness, as requests can continue to be processed. Responses may be slowed in some circumstances, but the entire system does not grind to a halt. Thus, if you have only a periodic slowness, an asynchronous system will allow you to work through that slowness without stopping the entire system. This approach may buy you several days to "fix" a scale bottleneck, unlike the immediate action required with a synchronous system.

In many places, however, you are seemingly "forced" to use a synchronous system. For instance, many database calls would be hampered and the results potentially

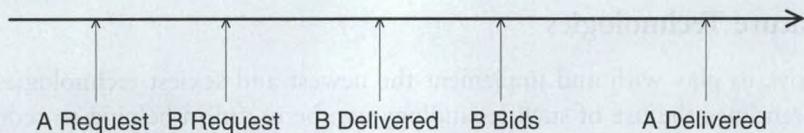


Figure 12.3 Asynchronous Ordering of Bidding Events

flawed if subjected to an asynchronous passing of messages. Imagine two servers requesting similar data from a database—for example, both of them asking for the current bid price on a car, as depicted in Figure 12.3.

In Figure 12.3, system A makes a request, after which system B makes a request. B receives the data first and then makes a bid on the car, thereby changing the car's price. System A then receives data that is already out of date. Although this seems undesirable, we can make a minor change to our logic that allows this to happen without significant impact to the entire process.

We need merely change the case in which A subsequently makes a bid. If the bid value by A is less than the bid made by B, we simply indicate that the value of the car has changed and display the now-current value. A can then make up her mind as to whether she wants to continue bidding. In this way, we take something that most people would argue needs to be synchronous and make it asynchronous.

Stateless Systems

Stateful systems are those in which operations are performed within the context of previous and subsequent operations. As such, information on the past operations of any given thread of execution or series of requests must be maintained somewhere. In maintaining state for a series of transactions, engineering teams typically start to gather and keep a great deal of information about the requests. State costs money, processing power, availability, and scalability. Although there are many cases where statefulness is valuable, it should always be closely evaluated for return on investment. State often implies the need for additional systems and synchronous calls that would otherwise not be required. Furthermore it makes designing a system for multiple live data centers more difficult: How can you possibly handle a transaction with state-related information stored in data center X in data center Y without replicating state between data centers? The replication would not only need to occur in near real time (implying a further requirement that data centers be relatively close), but also doubles the space needed to store relatively transient data.

Whenever possible, avoid developing *stateful* products. Where necessary, consider storing state with the end user rather than within your system. If that is not possible, consider a centralized state caching mechanism that keeps state data off of the application servers and allows for its distribution across multiple servers. If the state

needs to be *multitenant* for any reason, attempt to segment the state information by customer or transaction class to facilitate distribution across multiple data centers, and try to maintain persistency for that customer or class of transaction within a single data center, with only the data that is necessary for failover being replicated.

Scale Out, Not Up

A good portion of this book deals with the need to scale horizontally. If you want to achieve near-infinite scale, you must disaggregate your systems, organization, and processes to allow for that scale. Forcing transactions through a single person, computer, or process is a recipe for disaster. Many companies rely on Moore's law for their scale and as a result continue to force requests into a single system (or sometimes two systems to eliminate single points of failures), relying upon faster and faster systems to scale. Moore's law isn't so much a law as it is a prediction that the number of transistors that can be placed on an integrated circuit will double roughly every two years. The expected result is that the speed and capacity of these transistors (in our case, a CPU and memory) will double within the same time period. But what if your company grows faster than this rate, as did eBay, Yahoo, Google, Facebook, MySpace, and so on? Do you really want to become the company that is limited in growth when Moore's law no longer holds true?

Could Google, Amazon, Yahoo, or eBay run on a single system? Could any of them possibly run on a single database? Many of these companies started out that way, but the technology of the day simply could not keep up with the demands that their users placed on them. Some of them faced crises of scale associated with attempting to rely upon bigger, faster systems. All of them would have faced those crises had they not started to scale out rather than up.

The ability to scale out, rather than up, is essential when designing to leverage the elasticity and auto-scaling features present within many Infrastructure as a Service (IaaS) providers. If one cannot easily add read copies of databases, or additional Web and application servers, the benefits of renting capacity within "the cloud" (IaaS) cannot be fully realized. There may be no other principle as important to scaling a product as the ensuring that one can always scale horizontally (or out, rather than up).

Design for at Least Two Axes of Scale

Leaders, managers, and architects are paid to think into the future. You are not designing just for today, but rather attempting to piece together a system that can be used, with some modification, in the future. As such, we believe that you should always consider how you will execute your next set of horizontal splits even before that need arrives. In Chapter 22, we introduce the AKF Scale Cube for this purpose.

For now, suffice it to say there are multiple ways to split up both your application and databases. Each of these approaches will help you scale differently.

“Scaling out, not up” speaks to the implementation of the first set of splits. Perhaps you are splitting transaction volume across cloned systems. You might have five application servers with five duplicate read-only caches consisting of startup information and nonvolatile customer information. With this configuration, you might be able to scale to 1 million transactions per hour across 1,000 customers and service 100% of all your transactions from login to logout and everything in between. But what will you do when you have 75 million customers? Will the application’s startup times suffer? Will memory access times begin to degrade? More worrisome, can you even keep all of the customer information within memory?

For any service, you should consider how you will perform your next type of split. In this case, you might divide your customers into N separate groups and service customers out of N separate pools of systems, with each pool handling $1/N$ th of your customers. Or perhaps you might move some of the transactions (e.g., login and logout, updating account information) to separate pools if that will lower the number of startup records needed within the cache. Whatever you decide to do, for major systems implementations, you should think about it during the initial design even if you implement only one axis of scale.

Buy When Non-Core

We will discuss this principle a bit more in Chapter 15, Focus on Core Competencies: Build Versus Buy. Although “Buy when non-core” is a cost initiative, it also affects scalability and availability as well as productivity. The basic premise is that regardless of how smart you and your team are, you simply aren’t the best at everything. Furthermore, your shareholders expect you to focus on the things that *really* create competitive differentiation and, therefore, enhance shareholder value. So build things only when you are really good at it *and* it makes a significant difference in your product, platform, or system.

Use Commodity Hardware

We often get a lot of pushback on this principle, but it fits in well with the rest of the principles we’ve outlined. It is similar to the principle of using mature technologies. Hardware, especially servers, moves rapidly toward commoditization, a trend characterized by the market buying predominately based on cost. If you can develop your architecture such that you can scale horizontally with ease, you should buy the cheapest hardware you can get your hands on, assuming the cost of ownership for that hardware (including the cost of handling higher failure rates) is lower than the cost for higher-end hardware.

Build Small, Release Small, Fail Fast

Small releases have a lower probability of causing failures because the probability of failure is directly correlated to the number of changes in any given solution. Building small and making fast iterations also helps us understand if and how we need to change course with our product and architectural initiatives before we spend a great deal of money on them. When our initiatives don't work, the cost of failure is small and we can walk away from an initiative that does not work without great damage to our shareholders. While we can and should think big, we must have a principle that forces us to implement in small and iterative ways.

Isolate Faults

This principle is so important that it has an entire chapter dedicated to it (Chapter 21, Creating Fault-Isolative Architectural Structures). At its heart, it recognizes that while engineers are great at defining how things should work, we often spend too little time thinking about how things should fail. A great example of fault isolation exists in the electrical system of your house or apartment. What happens when your hair dryer draws too many amps? In well-designed homes, a circuit breaker pops, and the power outlets and electrical devices on that circuit cease to function. The circuit typically serves a single area within a location of the house, making troubleshooting relatively easy. One circuit, for example, does not typically serve your refrigerator, a single power outlet in a bathroom, and a light within the living room; rather, it covers contiguous areas. The circuit breaker serves to protect the remainder of the appliances and electrical devices in your house from harm and to keep greater harm from coming to the house.

The principle of isolating faults (i.e., creating fault isolation zones) is analogous to building an electrical system with circuit breakers. Segment your product such that the failure of a service or subservice does not affect multiple other services. Alternatively, segment your customers such that failures for some customers do not affect failures for your customer base in its entirety.

Automation over People

People frequently make mistakes; even more frustratingly, they tend to make the same mistake in different ways multiple times. People are also prone to attention deficiencies that cause them to lose track or interest in menial projects. Finally, while incredibly valuable and crucial to any enterprise, good people are costly.

Automation, by comparison, is comparatively cheap and will always make the same mistakes or have the same successes in exactly the same way each time. As such, it is easy to tune and easy to rely upon automation for simple repetitive tasks.

For these reasons, we believe all systems should be built with automation in mind at the beginning. Deployment, builds, testing, monitoring and even alerting should be automated.

Common Architectural Principles

AKF's 15 most adopted architectural principles are summarized here:

1. *N + 1 Design*. Never have less than two of anything, and remember the rule of three.
2. *Design for Rollback*. Ensure you can roll back any release of functionality.
3. *Design to Be Disabled*. Be able to turn off anything you release.
4. *Design to Be Monitored*. Think about monitoring during the design phase, not after the design is complete.
5. *Design for Multiple Live Sites*. Don't box yourself into one-site solutions.
6. *Use Mature Technologies*. Use things you know work well.
7. *Asynchronous Design*. Communicate synchronously only when absolutely necessary.
8. *Stateless Systems*. Use state only when the business return justifies it.
9. *Scale Out, Not Up*. Never rely on bigger, faster systems.
10. *Design for at Least Two Axes*. Think one step ahead of your scale needs.
11. *Buy When Non-Core*. If you aren't the best at building it and it doesn't offer competitive differentiation, buy it.
12. *Commodity Hardware*. Cheaper is better most of the time.
13. *Build Small, Release Small, Fail Fast*. Build everything small and in iterations that allow the company to grow.
14. *Isolate Faults*. Practice fault-isolative design—implement circuit breakers to keep failures from propagating.
15. *Automation over People*. Build everything to be automated—never rely on people to do something that a robot can do.

Conclusion

In this chapter, we discussed architectural principles and saw how they impact the culture of an organization. Your principles should be aligned with the vision, mission, and goals of your unique organization. They should be developed with your

team to create a sense of shared ownership. These principles will serve as the foundation for scalability-focused processes such as the joint architecture design process and the Architecture Review Board.

Key Points

- Principles should be developed from your organization's goals and aligned to its vision and mission.
- Principles should be broad enough that they are not continually revised, but should also be SMART and thematically bundled or grouped.
- To ensure ownership and acceptance of your principles, consider having your team help develop them.
- Ensure that your team understands the RASCI of principle development and modification.
- Keep your principles to a number that is easily memorized by the team to increase utilization of the principles. We suggest having no more than 15 principles.

Chapter 13

Joint Architecture Design and Architecture Review Board

Thus it is that in war the victorious strategist only seeks battle after the victory has been won,
whereas he who is destined to defeat first fights and afterwards looks for victory.

—Sun Tzu

In this chapter, we will introduce two proactive processes that are both cross-functional in nature and interwoven within the product or software development life cycle: joint architecture design (JAD) and the Architecture Review Board (ARB). JAD is a collaborative design process wherein all engineering personnel necessary to develop some new major functionality work together to define a design consistent with the architectural principles of the organization. The ARB is a review board of select architects from each of the functional or business areas, who ensure that, prior to the final signoff of a design, all company architectural principles have been incorporated and best practices have been applied.

Fixing Organizational Dysfunction

Most software development shops have at least one engineer who believes that the architects, database administrators, systems administrators, and network engineers either aren't knowledgeable about coding or don't fully understand the software development process. The same holds true for each of the other disciplines, with at least one architect or administrator believing that software engineers only know how to code and do not understand higher-level design or holistic systems concepts. Even worse, each party believes that the other's agenda is in direct opposition to accomplishing his or her own. Operations staff can often be heard mumbling, "We could keep the servers up if the developers would stop putting code on them," and developers mumble back, "We could develop and debug much faster if we didn't

have operations making us follow silly policies.” These perceptions and misgivings are destructive to the scalability of the application and organization.

In dysfunctional organizations, the implementation of a cross-functional JAD is challenging but absolutely necessary to help cure the dysfunction. As we discussed in Chapter 3, Designing Organizations, function-based organizations often experience affective conflict (the bad kind of conflict, as described in Chapter 1). Team members associate their social identity with their functional team rather than with the cross-functional team that delivers the product or service. This naturally creates an “us versus them” mentality. The different teams might also suffer from the “experiential chasm,” as discussed in Chapter 6, Relationships, Mindset, and the Business Case. Likewise, personnel handling different technology functions can find it difficult to communicate and may have widely differing skill sets, creating a gap that separates them just like the one between the business and technology teams. When a company’s subdivisions are organized by function, the implementation of a process to bring teams together to jointly design the product is essential. (We’ll talk in the next chapter about how the Agile organization helps resolve this issue without using process.)

Designing for Scale Cross-Functionally

JAD is a collaborative design process wherein all engineering assets necessary to develop some new major functionality or architectural modification work together to define a design consistent with the architectural principles and best practices of the company. JAD should be used whenever teams are not organized cross-functionally as described in Chapter 3. Its purpose in functionally oriented groups is to focus the teams on the shared outcomes of the products they build and to increase innovation through cross-functional cognitive conflict. Refer to Chapter 3 for an explanation or refresher on these concepts.

A JAD group should include the software engineer responsible for ultimately coding the feature, an architect, at least one but possibly multiple operations people, and—as needed based on the feature—the product manager, a project manager, and a quality assurance engineer. As mentioned earlier, each person on such a team brings unique knowledge, perspectives, experiences, and goals that complement as well as counter-balance the others. With a cross-functional team, the parties recognize more needs than just their own. For example, although the operations engineer still has the goal from her own organization of maintaining availability, she now also has the goal of designing a feature that meets the business requirements. This retention of the original goal combined with the embrace of a cross-functional goal helps ensure that she is vigilant as ever about what goes into production.

The JAD approach is not limited to waterfall development methodologies, where one phase of product development must take place before another phase

begins. JAD can and has been successfully used in conjunction with all types of development methodologies, such as iterative or Agile approaches, in which specifications, designs, and development evolve as greater insights into the product feature are gained. Each time a design is modified or extended, a JAD can be called to help with it. The type of architecture does not preclude the use of JAD, either. Whether it is a traditional three-tier Web architecture, service-oriented architecture, or simply a monolithic application, the collaboration of engineering, operations, and architects to arrive at a better design is simply taking advantage of the fact that solutions arrived at by teams are better than those produced by individuals. The more diverse the background of the team members, the more holistic the solution is likely to be.

The actual structure of the JAD team is very informal. After the team has been assigned to the feature, one person takes the lead on coordinating the design sessions; this individual is typically the software engineer or the project manager, if one is assigned. Usually multiple design sessions are held; they can last between one and several hours depending on people's schedules. For very complex features, multiple design sessions for various components of the feature should be scheduled. For example, a session focusing on the database and another addressing the cache servers should be set up separately.

Typically, the sessions start with a discussion covering the background of the feature and the business requirements. During this phase, it is a good idea to have the product manager present and then on call for any clarifications as questions come up. After the product requirements have been discussed, a review of the architectural principles that relate to this area of the design is usually a good idea. Next, the team brainstorms and typically arrives at a few possible solutions. These options are written up at the end of the meeting and sent around for people to ponder until the next session. Usually only one or two sessions are required to come to an agreement on the best approach for the design of the feature. The final design is written down and documented for presentation to the ARB.

JAD Checklist

Here is a quick checklist regarding the conduct of JAD sessions. As you become more comfortable with this process, you can modify and create your own JAD checklist for your organization to follow:

1. Assign participants.

- Mandatory: software engineer, architect, operations engineer (database administrator, systems administrator, and/or network engineer).
- Optional: product manager, project manager, quality assurance engineer.

2. Schedule one or more sessions. Divide sessions by component if possible: database, server, cache, storage, and so on.
3. Start the session by covering the specifications.
4. Review the architectural principles related to this session's component.
5. Brainstorm approaches. There is no need to identify complete details.
6. List the pros and cons of each approach.
7. If multiple sessions are needed, have someone write down all the ideas and send them around to the group.
8. Arrive at a consensus for the design. Use voting, rating, ranking, or any other decision-making technique that everyone can support.
9. Create the final documentation of the design in preparation for the ARB.

JAD Entry and Exit Criteria

With the JAD process, we recommend that specific criteria be met before a feature can begin the JAD process. Likewise, certain criteria should be met for that feature to move out of JAD. By holding fast to these entrance and exit criteria, you will preserve the integrity of the design process. Situations in which someone might suggest that these criteria could be bypassed include introducing features that aren't large enough to require a team effort to design or allowing a feature design process to start without an operations engineer on the team because the operations team is swamped handling a crisis. Giving in to one-off requests will ultimately devalue the JAD, however, and participants will believe that they can stop attending meetings or that they are not being held accountable for the outcome. Do not start down this slippery slope; make the entrance and exit criteria rigorous and unwavering, without exception.

The entrance criteria for JAD are the following:

- *Feature Significance.* The feature must be significant enough to require the focus of a cross-functional team. The exact nature of significance can be debated. We suggest measuring it in three ways:
 - Size. For size, we use the amount of effort needed for the feature development as the measurement. Features requiring a sum of more than 10 total engineering days are considered significant.
 - Potential impact on the overall application or system. If the feature touches many of the core components of the system, it should be considered significant enough to design cross-functionally.

- Complexity of the feature. If the feature requires components that are not typically involved in features such as caching or storage, it should go through JAD. A feature that runs on the same type of application server as the rest of the site and retrieves data from the database is not complex enough to meet this requirement.
- *Established Team.* The feature must have representatives assigned and present from engineering, architecture, and operations (database and system administrators, possibly a network administrator). If needed, personnel from the quality assurance, project management, and product management areas should be assigned to the JAD team. If these required team members are not assigned and made available to attend the meetings, the feature should not be allowed to undergo JAD.
- *Product Requirements.* The feature must have product requirements and a business case that warrant use of JAD. Tradeoffs are likely to be made based on different architectural solutions, and the team will need to be able to distinguish the critical requirements from the nice-to-have ones. Understanding the revenue generated will also help the team decide how much investment should be considered for different solutions.
- *Empowerment.* The JAD team must be empowered to make decisions that will not be second-guessed by other engineers or architects. The only team that can approve or deny the JAD design is the ARB, which performs the final review of the architecture. In RASCI terminology, the JAD team is the R (Responsible) for the design and the ARB is the A (Accountable).

The exit criteria for a feature coming out of JAD are the following:

- *Architectural Principles.* The final design of the feature must follow all architectural principles that have been established in the organization. If there are exceptions to this rule, they should be documented and questioned by the ARB, resulting in a possible rejection of the design. We will talk more about the ARB process in the next chapter.
- *Consensus.* The entire team should be in agreement and support the final design. The time to voice dissent in regard to the design is during the team meetings, not afterward. If someone from the JAD team begins second-guessing team decisions, this should be grounds for requiring the JAD process to be conducted again, and any development of the feature should be stopped immediately.
- *Documentation of Tradeoffs.* If any significant tradeoffs were made in the design with respect to the requirements, cost, or principles, they should be clearly articulated to the ARB and for any other team member to reference when reviewing the design of the feature.

- *Documentation of the Final Design.* The final design must be documented and posted for reference. It may or may not be reviewed by the ARB, but the design must be made available for all teams to review and reference in the future. These designs will soon become system documentation as well as design patterns that engineers, architects, and operations folks can reference when they are participating in future JAD processes.
- *ARB.* The final step in the JAD process is to decide whether the feature needs to go to the ARB for final review and approval. We will talk in more detail in the next chapter about which features should be considered for ARB review, but here are our basic recommendations for appropriate criteria:
 - Noncompliance with architectural principles. If any of the architectural principles were violated, the feature should go through ARB review.
 - Projects that cannot reach consensus on design. If the team fails to reach consensus, the project can be either reassigned to a new JAD team or sent to the ARB for a final decision on the competing designs.
 - Significant tradeoffs. If tradeoffs had to be made in terms of product requirements, cost, or other non-architectural principles, this compromise should flag a feature as needing ARB review.
 - High-risk features. We will discuss how to assess risk in much more detail in Chapter 16, Determining Risk; for now, simply recognize that if the feature is considered high risk, its design should go through ARB review. A quick way of identifying high-risk features is to look at how many core components the feature touches or how different it is from other features. The more core components that are touched and the greater the difference from other features, the higher the risk.

From JAD to ARB

The most important consideration when forming the ARB is the traits of the individuals on the board. To start with, you want people who are respected in the organization. They may be respected because of their position, their tenure, or their expertise in a particular area of technology or business.

People who hold a particular position can be important to the ARB to provide the gravitas to uphold the board's decisions. The ARB needs to include the right people to make the right decision and to be given the final authority over making that decision. In turn, the ARB may need to have executive participation. If the VPs delegate the ARB responsibilities to managers or architects, then the VPs need to support and not second-guess their subordinates. The ARB, in these matters, would be considered the A (Accountable) within the RASCI process.

Many leaders within an organization do not hold a manager or executive title. These are the people to whom the team looks in meetings to sway opinions and provide guidance. This trait of peer leadership is one that you want to look for when selecting people to place on the ARB.

Expertise, whether in engineering disciplines, architecture, or business, is a characteristic that people should display if they are participants on the ARB. Such individuals usually are in roles such as architects, senior engineers, or business/product owners, though they can be found elsewhere as well. These people's input is sought when a tough question must be answered or a crisis resolved. Their expertise comes in a variety of subjects; for example, it might include expertise on the platform itself because of a long tenure working with the product, expertise with a specific technology such as caching, or even expertise with a specific large customer of the business.

Ideally, then, ARB members will be leaders who are respected and who have domain expertise. Some members may have a greater amount of one characteristic than another. For instance, a senior director of engineering may be well respected and display great leadership, yet not have true domain expertise. This senior director may still be an excellent candidate for the review board. Here are some roles within the organization that you should consider when assessing candidates for membership on the ARB:

- Chief architects
- Scalability architects
- Infrastructure architects
- VP or directors of engineering
- VP or directors of operations or infrastructure
- Senior systems administrators, database administrators, or network engineers
- Senior engineers
- CTO or CIO
- General manager of business unit
- Product or business owners

This list is not inclusive but should provide you with an idea of where to look for individuals who display the three key traits of respectability, leadership, and domain expertise. As with most topics that we have discussed, the real test is whether the ARB approach works for and within your organization. The number of members on the ARB can vary depending on the organization, the number of people available, and the variety of skill sets required. We recommend the board consist of between four and eight members.

Membership on the ARB should be considered an additional responsibility to an individual's current role. It is always considered voluntary, so if necessary a member can ask

to be excused. Ideally, we would like to see the ARB remain in place for a significant period of time so that this team can establish tenure in terms of assessing projects and designs. You can modify the permanent or nonpermanent nature of this membership in many ways, however, and you may want to consider several factors when deciding who should be a permanent member versus who should fill a rotating seat.

One factor to take into account is how many suitable members are present in your organization. If there are only four people who display the traits necessary for serving on this board, you will probably have to insist that these individual occupy permanent positions on the ARB. Another factor that will determine how permanent, semi-permanent, or rotational this role should be is how often you have features that need to proceed through ARB review. If your organization has enough engineers and enough JAD projects that the board must meet more than once per week, you may need to rotate people on the board or even consider having two different ARBs that can alternate in fulfilling this responsibility. A third factor, besides the number of viable candidates and the number of ARB meetings, is the specificity of the members' expertise. If there are multiple technologies or technology stacks or separate applications, you should consider rotating people in and out of the board depending on the characteristics of the feature being discussed.

Several different methods of rotation for the ARB positions may be used. One straightforward method is to change the constituency of the board every quarter of the year. Depending on how many people are fit for this service, they could rotate on the board every six months or once each year or even beyond. Another approach is to have some individuals be permanent members of the board, such as the architects, and rotate the management representative (e.g., VP of engineering, VP of operations, CTO) and the team members (e.g., senior engineer, senior systems administrator, senior network engineer). Any of these methods will work well as long as there is consistency in how each team approaches its decisions and the team is given the authority of approving or rejecting JAD proposals.

Conducting the Meeting

Our experience is that ARB participation can be very intimidating for line engineers, database administrators, and other junior members of the staff. As such, we prefer the meetings to be informal in nature. Any formality should come from the fact that a go or no-go decision will be made on the architecture of the feature.

Each meeting should include the following elements:

1. *Introduction.* Some members of the design team may not know members of the ARB if the engineering organization is large.

2. *State Purpose.* Someone on the ARB should state the purpose of the meeting so that everyone understands what the goal of the meeting is. We suggest you point out that the ARB will be making a judgment on the proposed architecture and not on people on the design team. If the design is sent back with major or minor revisions requested, the decision should not be taken as a personal attack. The agenda for everyone in the organization should be to ensure the proper governance of the IT systems and support the scalability of the system.
3. *Architecture Presentation.* The design team should present the design to the ARB. A well-structured presentation should walk the ARB members through the thought process starting with the business requirements; it should follow this with an outline of the tradeoffs, alternative designs, and finally the recommended design, including its strengths and weaknesses.
4. *Q&A.* The ARB should spend some time asking questions about the design to clarify any points that were vague in the presentation.
5. *Deliberation.* The ARB members should dismiss the design team and deliberate on the merits of the proposed design. This review can take many forms. For example, the board members might cast an initial vote to determine where each member stands, or they might choose someone to lead a point-by-point discussion of the pros and cons of the design before casting ballots.
6. *Vote.* The ARB should have an established process for determining when product features are approved and when they are rejected. We have often seen ARBs reject a design if a single member votes “no.” You may want to adopt a three-fourths rule if you believe reaching 100% agreement is unlikely and will be unproductively arduous. In such a case, we recommend that you reconsider who makes up the constituency of the board. Members should most strongly advocate for what is best for the company. Someone who abuses his or her power and consistently hassles design teams is not looking out for the company and should be replaced on the ARB.
7. *Conclusion.* When a decision is made, the ARB should recall the design team and explain its decision. This decision could be one of four courses:
 - *Approval.* The ARB could approve the design to move forward as outlined in the proposal.
 - *Rejection.* The ARB could reject the design and request a completely new design be developed. This second choice is an absolute rarity. Almost always, there is something from the proposed design that can be salvaged.
 - *Conditional Approval.* The ARB could approve the design to move forward, albeit with some changes. This choice would indicate that the team does not need to resubmit the design to ARB but can proceed under its own guidance.

- *Rejection of Components.* The ARB could reject the proposed design but make specific requests for either more information or redesigned components. This fourth option is the most common form of rejection, and the specific request for more information or a change in the design usually comes from specific members of the ARB. Such a decision does require a resubmission of the design to the ARB for final approval prior to beginning development on the feature.

These steps can be modified as necessary to accommodate your team size, expertise, and culture. The most important item to remember (and you should remind the team of this point) is that ARB members should first and foremost put what is best for the company before their personal likes, distastes, or agendas, such as something causing more work for their own organizational team. Of course, what is best for the company is to get more products in front of the customers while ensuring the scalability of the system.

ARB Entry and Exit Criteria

Similar to the JAD process, we recommend that specific criteria must be met before a product feature can begin the ARB process. As such, certain criteria should be established for when that feature can move out of ARB review and development can begin. These criteria should be upheld as strict standards to ensure that the ARB process is respected and decisions emanating from the board are accepted. Failure to do so results in a weak process that wastes everyone's time and is eventually bypassed in favor of quicker routes to design and development.

The entry criteria for a feature coming out of JAD into ARB review are the following:

- *Established Board.* The Architecture Review Board must be established based on the criteria mentioned earlier in terms of roles and behaviors that the members should demonstrate.
- *Design Complete.* The feature should meet the exit criteria outlined for JAD:
 - *Consensus.* The design team should be in agreement and all members should support the final design. If this is absolutely not possible, a feature may be submitted to the ARB with this fact acknowledged and the two or more proposed designs each presented.
 - *Documentation of Tradeoffs.* If any significant tradeoffs were made in the design with respect to the requirements, cost, or principles, they should be documented.
 - *Documentation of Final Design.* The final design must be documented and posted for reference.

- *Feature Selection.* The feature having completed design should be considered as a candidate for ARB review. If this feature has any of the following characteristics, it should proceed through ARB review:
 - *Noncompliance with Architectural Principles.* If any of the architectural principles were violated, this feature should go through ARB review.
 - *Projects That Cannot Reach Consensus on Design.* If the team fails to reach consensus, the feature can be either reassigned to a new design team or sent to the ARB for a final decision on the competing designs.
 - *Significant Tradeoffs.* If tradeoffs had to be made in terms of product requirements, cost, or other non-architectural principles, this compromise signals that the feature should proceed to ARB review.
 - *High-Risk Features.* We will discuss how to assess risk in much more detail in Chapter 16, Determining Risk; for now, recognize that any high-risk feature should go through ARB review. A quick way of determining if the feature is high risk is to look at how many core components it touches or how different it is from other features. Either of these factors increases the amount of risk associated with the feature.

Depending on the decision made by the ARB, different exit criteria exist for a feature. Here are the four possible decisions on these criteria and what must be done following the ARB session:

- *Approval.* Congratulations—nothing more is required of the team by the ARB. Now the tough part begins, as the team must develop and implement the project as it has been designed.
- *Rejection.* If the design is completely rejected, the ARB should provide a rationale for its decision. In this case, the same design team may be asked to redesign the feature or a new team may be formed to create the second design. The team should remain in place to provide the design if the current team has the right expertise and if it is still motivated to succeed.
- *Conditional Approval.* If the ARB has conditionally approved the design, the team should incorporate the board's conditions into its design and begin to produce the feature. The team may return to the ARB in person or via email if any questions arise or if it needs further guidance.
- *Rejection of Components.* If the ARB rejects the design of certain components, the same design team should come up with alternative designs for these components. Because the ARB review is most often treated as a discussion, the design team should have a good idea of why each component was rejected and what would be needed to satisfy the board. In this case, the design team needs to schedule a subsequent presentation to the ARB to receive final signoff on its design.

Checklist: Keys for ARB Success

The following is a checklist of key attributes or actions that you should follow to ensure that your ARB review process is a success:

- Proper board composition
 - Successfully complete team design with the feature
 - Ensure the right features get sent to the ARB
 - Do not allow political pressures to be exerted such that features bypass the ARB review
 - Ensure everyone understands the purpose of the ARB—improved scalability through rigorous design
 - The design team should be well prepared for its presentation and Q&A session
 - Establish in advance the ARB criteria for passing (100% agreement among members is recommended)
 - No petitions allowed on the board's decisions
-

Conclusion

In this chapter, we covered both the joint architecture design process and the Architecture Review Board in detail.

All too often, dysfunction in technology organizations causes features to be designed in silos. Such problems are especially likely to arise in function-based organizations as well as when an experiential chasm is present. The fix is forcing the teams to work together for a shared goal—something that can occur through the JAD process.

The JAD process includes both mandatory participants and optional team members. The design meetings should be structured based on components, and it is important to start by making sure every team member is familiar with the business requirements of the feature as well as the applicable architecture principles of the organization.

A JAD checklist can help you and your organization get started quickly with the JAD process. We recommend that you start with our standard steps as outlined on the checklist, but fill it out as necessary to make it part of your organization. Of course, you should also document this process so it becomes fixed in your organization's culture and processes.

The entry criteria for JAD focus on the preparation needed to ensure the JAD effort will be successful and the integrity of the process remains intact. Letting

features slip into a JAD without all the required team members being on board is a surefire way to cause the process to lose focus and effectiveness. The exit criteria for JAD address the need for all members of the team to agree on the feature design and, if necessary, to present the design for ARB review.

Three traits are essential for members of the ARB: respect, leadership, and expertise. The amount of formality that characterizes the ARB members' review depends on the culture of the organization, but we recommend keeping it as informal as possible to avoid intimidating design team members and to foster a healthy, productive discussion about the design.

The entry criteria for ARB review are focused on ensuring that the right feature is being sent to the ARB, that the right ARB team is formed, and that the feature is prepared as well as possible to proceed through ARB. Selecting the right feature is not always easy. Recognizing this fact, we recommend four tests for whether a feature should proceed through ARB review: noncompliance with architectural principles, significant tradeoffs having to be made to the business requirements, inability of the JAD team to reach consensus, and high-risk features.

Here is one final thought about how to implement JAD and ARB. The company Shutterfly, an Internet-based image publishing service based in Redwood City, California, implemented JAD and ARB review. It utilized JAD for features that "some other team would notice." If the developers could implement a feature without anyone noticing, it was deemed not worthy of JAD. If another team would notice the feature (e.g., the network team would notice a spike in traffic), then it might be considered JAD-worthy. Similarly, if a feature was big enough that most or all teams would notice its implementation, then it was considered ARB-worthy. Such simple criteria are easy for everyone to understand in regard to when to implement each process, but strict enough to ensure the needed features get the attention they deserve.

Through the use of JAD and the ARB, your organization can be ensured of better designs that are purposefully made for improving scalability.

Key Points

- Designing applications in a vacuum leads to problems; the best designs are created by involving multiple groups offering different perspectives.
- The JAD is the best way to pull together a cross-functional team whose members may not otherwise be incented to work together.
- The JAD team must include members from engineering, architecture, and operations (database administrators, systems administrators, or network engineers).
- JAD is most successful when the integrity of the process is respected and entry and exit criteria are rigorously upheld.

- A final review of the application's architecture ensures buy-in and acknowledgement as well as prevents finger pointing.
- The proper constituency of the ARB is critical if it is to uphold the purpose of a final architecture signoff and if its decisions are to be respected.
- Members of the ARB should be seen as leaders, be well respected, and have expertise in some area of the application or architecture.
- ARB membership can be assigned on a rotational basis, but the position is always seen as incremental to each member's current duties.
- All ARBs should start with a discussion of the purpose of the ARB—to ensure designs support the organization's business needs, including scalability.
- Entry into ARB review should be granted only to features that are sufficiently prepared.
- Decisions by the ARB should be considered final. Some organizations may include an appeals process if it is deemed necessary.

Chapter 14

Agile Architecture Design

We shall be unable to turn natural advantage to account unless we make use of local guides.

—Sun Tzu

You may recall from Chapter 9, Managing Crises and Escalations, that one of our favorite technology companies is Etsy, the marketplace for handmade or vintage items. While this company has a stellar technology team, with John Allspaw at the helm of the technical operations team, it also experiences site outages occasionally. One such event occurred on July 30, 2012, while employees were performing a database upgrade to support languages that required multi-byte character sets.¹ They upgraded one of the production databases and everything went as planned. A database upgrade, especially one that changes the encoding of the data at rest, runs the risk of corrupting or losing data. The team then put together a careful plan for slowly upgrading the remaining servers over a period of time. The upgrade had gone out to only one server, and the upgrades for the other servers were basically placed “on deck.”

Etsy often has many changes queued up to be introduced to production. The company was having problems with site slowness during the nightly backups and had queued up an improvement to allow for faster backups. To release the backup fixes, Etsy’s engineers were using an automated tool whose job was to make sure all of the servers were consistent. After testing, they pushed the fix to the backups using the tool, and expected to confirm later that night that backups successfully ran without interfering with the site’s performance. What they didn’t know at the time was that deploying the improvement to the backups also deployed the database language upgrade.

1. One byte can represent 256 characters, which is enough for the combined languages of English, French, Italian, German, and Spanish. The character sets of other languages, such as Chinese, Japanese, and Korean, include a larger set of ideographic characters that require two or more bytes to represent such a great number of these complex characters. The term for mixing single-byte characters alongside two-or-more-byte characters is “multi-byte.”

Once the team realized that approximately 60% of Etsy's database servers were upgrading their character sets while actively serving traffic, they quickly brought the site down to protect against corruption. The team that responded to this event was multidisciplinary in its composition. As Allspaw states, "No one team owns response to an incident." At Etsy, many teams participate in a 24/7 on-call group. There are no problem managers necessary because the teams own the services they produce. In some cases, such as the search team, the multidisciplinary Agile development team acts as first responders. The search team receives the alerts first and escalates the problem to technical operations only when necessary.

Once the team was able to confirm that the databases were correct and behaving normally, they brought the site back up. All the while, they communicated to the community through the EtsyStatus site (<http://etsystatus.com/>).² This example shows how Agile teams (5- to 12-person cross-functional teams of product managers, engineers, QA personnel, and DevOps staff) can own the service or product they produce from architectural design all the way through production support.

In the last chapter we discussed two processes, joint architecture design (JAD) and the Architecture Review Board (ARB). The purpose of using these processes was to ensure we had cross-functional teams designing our products (JAD) and multidisciplinary high-level engineers and executives ensuring the consistency of standards (ARB). In this chapter, we will approach the design of features and systems in a different manner. We'll explain how the new Agile Organization is responsible for the design of its part of the system and how these independent Agile teams are capable of following standards.

Architecture in Agile Organizations

Let's start by reviewing the Agile Organization that we described in detail in Chapter 3, Designing Organizations. In functionally aligned organizations, individuals are organized by their skill, discipline, or specialty. However, almost every project requires coordination across teams, which means coordination across functions. This is especially true of SaaS offerings, where the responsibility of not only developing and testing the software, but also hosting and supporting it, falls on the company's technology team. This results in some amount of affective conflict. The amount obviously depends on many factors but we want to minimize all of it if possible. Recall that affective conflict is "bad" conflict centered on roles and ownership and often involves questions of "who" owns something or "how" a task should be done.

2. This case study was taken from John Allspaw's posting "Demystifying Site Outages," <https://blog.etsy.com/news/2012/demystifying-site-outages/>.

This type of conflict is destructive and leads to physical and emotional stress on team members. Physically, it can leave us drained as our sympathetic nervous system (the same system involved in the fight-or-flight syndrome kicked off by the hypothalamus) releases the stress hormones cortisol, epinephrine, and norepinephrine. Organizationally, teams may fight over the ownership of products and approaches to problems, leading to closed minds and suboptimal results. Agile Organizations, in contrast, break down the organizational boundaries that functional organizations struggle with and empower the teams, eliminating the problem that matrix organizations face.

Agile Organizations are 5- to 12-person teams made up of personnel with the skill sets necessary to design, develop, deliver, and support a product or service for customers. These teams are cross-functional (multidisciplinary) and self-contained. They are empowered to make their own decisions without seeking approval from people outside their teams. They are capable of handling the full life cycle of their products or services. When teams are aligned by services, are autonomous, and are cross-functionally composed, there is a significant decrease in affective conflict. When team members are aligned by shared goals and no longer need to argue about who is responsible or who should perform certain tasks, the team wins or loses together. Everyone on the team is responsible for ensuring the service provided meets the business goals.

Recall the theory of innovation we presented in Chapter 3, which informed multi-disciplinary team construction with the purpose of driving higher levels of innovation. In a SaaS product offering, this increased innovation is often measured in terms of faster time to market with features, better quality of the product, and higher availability. The drivers of this innovation are the decreased level of affective conflict and the increased levels of cognitive conflict, network diversity, and empowerment. The Agile Organization structure provides all of these drivers, and the result is often a significant increase in innovation.

Ownership of Architecture

All of this increase in innovation, greater autonomy, and less affective conflict sounds great. However, along with great power comes great responsibility. These Agile teams own the architecture of their services or products. In other words, instead of relying on a separate architecture team, this team owns the design and implementation of its products. How does this work in practice?

The Agile team is composed of personnel who have all the skill sets necessary for the team to remain autonomous and produce its product. The typical members of the team include a product manager, several software developers, a quality assurance

engineer, and a DevOps engineer. If the organization has software architects, these individuals are placed on Agile teams as well. The composition should remind you of the JAD process composition from Chapter 13. If the Agile team is to design and develop highly available and scalable products and services, its members need the same skills that any other group would need to accomplish the task. JAD is a Band-Aid meant to create the multidisciplinary design within functionally oriented teams that is inherent to Agile teams.

We know that experiential, network, and skill set diversity allows individuals to develop better solutions to problems—and architecting a feature, story, solution, bug fix, or product offering is no different. The best solutions come from teams that include representatives from multiple disciplines who can draw on their different viewpoints and experiences to attack the problem. Ask a software engineer to design a door, and he or she immediately begins thinking about how a door works—how do the hinges work, how does the handle turn, and so on? Ask a product manager to design a door, and he or she might begin thinking about the benefits the door should achieve, how other competing doors function, and what the last round of user testing revealed. Ask a system administrator to design a door, and he or she will probably start thinking about how to secure it or how to recover from common failures within latches or locks. When these different viewpoints are collected together, they make a stronger, more scalable, and more available product.

Network diversity is a measure of how individuals on a team have different personal or professional networks. This becomes important with regard to innovation because almost all projects run into roadblocks. Teams with diverse networks are better able to identify potential roadblocks or likely problems early in the project because they can seek advice from a wide variety of people outside of the team. When the team actually does encounter roadblocks, those teams with the most diverse networks are better suited to finding resources outside of their teams to get around the obstacle.

Limited Resources

When an Agile team has a diverse skill set, a broad network of contacts, and a wide variety of experiences to draw upon, the team members are more able to design, develop, deploy, and support highly reliable and scalable products. But what happens when the company can't afford to put an architect on every team, or when there are only two DevOps engineers and six Agile teams? This problem arises not only with cash-constrained startups but also with the flush-with-cash, hyper-growing companies. In the first case, the company cannot afford to hire as many architects, DevOps

engineers, and other technical professionals as it would like. In the second case, the company might not be able to attract and retain as many architects or DevOps engineers as quickly and as long as it would like. It is very common to grow teams asymmetrically by hiring software developers first, then product managers, then perhaps a QA engineer, then a DevOps engineer, and finally an architect. This hiring process to fill out the team might take a year to 18 months to accomplish. No company will let the software developers sit idle for this amount of time while waiting for an architect to join their team. But what is a team to do in this situation?

When faced with limited personnel and resources, teams often try to make do with what the organization has at the time. This approach can result in considerable risk for the company. Consider the situation where there are not enough product owners. The outcome may be hastily written user stories and poor prioritization, which can in turn result in costly mistakes made by the Agile teams.

The first step for the Agile teams is to ensure they have the necessary resources to perform their function well. Anytime a key resource is missing, the team is in jeopardy. One way to make the business case for necessary resources is to use the Kanban board to visually display bottlenecks in the development process. Another approach is for the team to share resources across teams. The limited resource, such as DevOps engineers, can be shared across teams. These individuals should be assigned to multiple teams but not all teams. Think multitenant but not all-tenant. For example, if you have two DevOps engineers and six Agile teams, assign DevOps engineer 1 to Agile teams A, B, and C. Assign DevOps engineer 2 to Agile teams D, E, and F. This way the teams feel some sense of connectedness with the DevOps engineers assigned to them. They start to think, “DevOps engineer 1 is my go-to person” rather than “We have a pool of two DevOps engineers to whom I make requests by submitting a ticket to a queue.” See the difference? In once scenario we’re breaking down walls; in the other we’re ensuring that they remain up between the teams.

Standards

One topic of discussion that invariably comes up with multiple teams is maintaining standards across those teams. If we allow each Agile team to autonomously decide which patterns, libraries, frameworks, or even technologies it will rely upon, how does the company benefit from economies of scale? How will engineers who transfer between teams have any common or shared knowledge? The answer to these questions, as in a lot of cases, is “It depends.” It depends on the organization, the leaders, and the team members. Let’s look at a few different approaches.

Economies of Scale

The term “economies of scale” refers to the cost advantages that companies realize due to the relative size or scale of their operations. The basic concept is that the cost per unit of output decreases with increasing scale because fixed costs are spread out over more units of output. This idea dates back to Adam Smith (1723–1790) and the notion of obtaining larger production returns through the use of division of labor.

For an engineering team, economies of scale come from items such as shared knowledge about a technology or way of operating. If we break down the cost of developing a software service into fixed and variable costs from a project perspective (not an accounting perspective, which would be different), we get the following result. The fixed costs consist of the knowledge and skills developed ahead of time by the software developers. We pay for this overhead with every sprint (typical 2-week development cycle) whether we code one story or 50 stories. The more stories we code per sprint, the more broadly the fixed costs are distributed. Instead of one story costing the full 100% of the fixed costs, if we code 20 stories, each pays only 5% of the fixed costs. If each software development team (Agile team) has to be an expert on different languages, technologies, or processes, the fixed costs of this knowledge are shared only across those stories that a single team codes. If all the teams can leverage one another’s knowledge, however, the costs can be shared across all teams.

Some organizations hold the beliefs that teams should be allowed to decide independently on all things and that the best ideas will permeate across teams. Other organizations take a different approach, bringing members of teams together to make decisions about standards that these members are expected to uphold when they return to their teams. We’ll start by looking at how Spotify, the digital music service, addresses the design of architecture by Agile teams. We’ll then contrast this approach with that of Wooga, a social gaming company that takes a slightly different approach.

We highlighted Spotify’s Agile team structure in Chapter 3, Designing Organizations, and want to return to that organization to take a deeper dive into how it coordinates across teams. If you recall, Spotify organizes around small teams called squads that are similar to a Scrum team (in our vernacular, an Agile team). Squads are designed to feel like a mini-startup, containing all the skills and tools necessary to design, develop, test, and release their services into production. As Kniberg and Ivarsson state in their October 2012 paper “Scaling Agile @ Spotify with Tribes, Squads, Chapters, and Guilds,” “There is a downside to everything, and the potential downside to full autonomy is a loss of economies of scale. The tester in squad A may be wrestling with a problem that the tester in squad B solved last week. If

all testers could get together, across squads and tribes, they could share knowledge and create tools for the benefit of all squads.” They continue with the question, “If each squad was fully autonomous and had no communication with other squads, then what is the point of having a company?” For these reasons, Spotify has adopted chapters and guilds.

A chapter is a small group of people who have similar skills. Each chapter meets regularly to discuss its area of expertise and challenges. The chapter lead is a line manager as well as a member of a squad, involved in the day-to-day work. A guild is a more organic and wide-reaching “community of interest”—that is, a group of people who want to share knowledge, tools, code, and practices. Chapters are always local to a tribe (a collection of squads working in a related area), while a guild usually cuts across the entire organization. By utilizing chapters and guilds, Spotify can ensure that architectural standards, development standards, libraries, and even technologies are shared across teams. The chapter and guild leaders facilitate the discussions, experiments, and ultimately the decisions by which all the teams will comply. Chapter and guild members who participate in the process are responsible for bringing the knowledge and decisions back to their team and ensure their colleagues abide by the chapter’s or guild’s decisions. This approach offers a nice balance between an autocratic top-down approach and a democratic bottoms-up approach. Even so, it isn’t the only way for small, independent Agile teams to design and architect their services and products within a larger organization.

In Jesper Richter-Reichhelm’s article “Using Independent Teams to Scale a Small Company: A Look at How Games Company Wooga Works”³ posted on *The Next Web* (<http://thenextweb.com/>) on September 8, 2013, the author outlines how he approaches fostering independent teams and challenges some of the Spotify ideas. Richter-Reichhelm is the head of engineering at Wooga, a social gaming company founded in 2009. Wooga has grown from around 20 employees in its first year to more than 250 employees in 2013. Richter-Reichhelm states, “In the early days everyone in the company worked closely together and were not slowed down having to wait for approvals. Normally as a company grows, this changes as management layers are added, and work simply becomes less efficient. How did we hold onto that culture? The answer: centering everything around independent game teams.”

Similar to Spotify and our model of the Agile team, Richter-Reichhelm created small autonomous, cross-functional teams that were responsible for independent games. These teams write and operate the games themselves, not relying on a centralized technical operations team or a centralized framework. “Engineers are not forced to share or reuse code” is how Richter-Reichhelm describes the level of

3. <http://thenextweb.com/entrepreneur/2013/09/08/using-independent-teams-to-scale-a-small-company-a-look-at-how-games-company-wooga-works/>.

independence under which each team operates. With regard to input from individuals outside the team, including the company founders, “It’s completely up to them [the team] if they want to listen or ignore outside advice.”

However, to leverage knowledge and gain some economies of scale, the Wooga teams actively share knowledge. They accomplish this through weekly status updates, lightning talks, brown bag talks, and other interactions. This shared knowledge helps the teams not have to relearn the same lessons over and over again. As Richter-Reichhelm describes it, “This way we can try out new things in one game, and when they work, that knowledge is spread to other teams. This works quite organically.” It should be pointed out that teams have shared results such as key performance indicators (KPIs) but these are not used competitively across teams.

The approach at Wooga is very different from the approach at Spotify, where chapter and guild leaders are tasked with ensuring knowledge and standards are shared across teams. So which one is the best? Both approaches are necessary, and companies need to evaluate which standards need to be in place across their own teams and which standards can be established by each of these teams. For your own organization, it depends on the organization’s culture, maturity, and processes. What are you as a leader comfortable with? Do you have a culture mix that includes experienced enough individuals to act independently in the best interest of the company, or do you need stronger oversight? The answers to these questions will lead you down the path to the best answer for your organization at a particular time. As your organization matures and grows, the approach to this might need to change as well.

ARB in the Agile Organization

As we mentioned earlier, the Agile team provides the cross-functional design that the JAD process attempts to achieve; thus JAD is not necessary when employees are organized into true autonomous, cross-functional Agile teams. The next logical question is, “Do you need an ARB in an Agile Organization?” The answer, once again, is “It depends.” Many companies with which we consult that have multi-disciplinary teams still have an ARB process. The primary benefit of the ARB is that it provides a third-party view of the team and as such is not subject to the group-think phenomenon that sometimes plagues autonomous teams. However, if we put the ARB within the product life-cycle development process, then we are impacting the benefit of autonomy and time to market benefits engendered by Agile teams. One potential fix to gain the pros and mitigate the cons is to perform the ARB review after the sprint as part of the retrospective. Another option is to convene the ARB daily (perhaps at lunch) for any projects that need to be reviewed. This way, teams are not stopped for any substantial period of time waiting for the ARB to convene.

When using an ARB in an Agile Organization, the primary goal is to ensure the architecture principles agreed to are being followed. This helps ensure consistency across teams. The board's second goal is to teach the teams' engineers and architects through interaction. This becomes increasingly important as team sizes grow quickly or upon acquisition of new companies where prior standards exist. Lastly, the board helps evaluate team members individually in terms of how they understand and enforce standards. They also evaluate the teams themselves—evaluating how they come together to create designs to help correct deficiencies.

Conclusion

Agile teams, which are designed to be small and autonomous, can be independently responsible for the architecture and design of their services and products. This autonomy in regard to their designs is critical if the teams are to be as nimble as possible. While the JAD process can be effective at ensuring cross-functional designs are taking place, a cross-functional Agile team ensures this outcome occurs by its very composition.

Sometimes, organizations face the prospect of limited resources, such as a smaller number of DevOps engineers than teams. Our advice is to assign individual DevOps personnel to a number of teams. Ideally, teams should know their go-to person by name and not have to submit tickets to a queue. This helps break down the walls that organizational structure can put up.

One of the questions we often get when discussing Agile teams with clients is how standards can be shared across teams if the teams are completely autonomous. One approach (used by Spotify) involves chapters and guilds that cross Agile teams (squads) to determine and enforce these standards. A different approach (used by Wooga) is for teams to be very independent but actively share knowledge through various forums. Finally, you can consider using an ARB to validate principle adoption and compliance by either convening frequently or using their meetings as a retrospective on recently released designs.

Key Points

- Agile teams should act autonomously, which means they should own the design and architecture of their services and products.
- The ARB and JAD processes ensure a cross-functional design of services. The Agile team, by its constitution or makeup, ensures this outcome as well.
- When resources such as DevOps engineers or architects are limited, assign them to multiple teams as named individuals. Do not revert back to tickets and queues fronting a pool of nameless DevOps resources.

- Sharing standards and knowledge across teams to achieve economies of scale can still be accomplished with Agile teams. There are various approaches that can be used in such cases. Choosing the right approach for your organization depends on multiple factors, including your teams' maturity, your products' complexity, and your comfort with distributed command and control.

Chapter 15

Focus on Core Competencies: Build Versus Buy

While we've made the point that you should never rely on a vendor as the solution for scaling your product, we do not mean to imply that third-party solutions do not have a place in your product. In fact, as we will discuss in this chapter, we believe that companies that specialize in developing specific components should provide most of the components within your product. The difference between allowing a vendor to provide scale of your product and allowing a vendor to provide components is simple: You are the owner of your product and must architect it to both achieve the business case and scale to meet end-user demand. No third party will ever have your best interests in mind; every entity has its own shareholder expectations to meet. Let's turn our attention to when and how we should source product components from third-party providers.

Building Versus Buying, and Scalability

Everything that we build within our products has a long-term cost and effect on our product spending. For each component we build, we will spend some small portion of our future engineering resources maintaining or supporting that component. As the number of components that we build increases, if we do not obtain additional funding (resources), our ability to build new solutions starts to dwindle. Obviously, if you were to build everything from your computers to your operating systems and databases, you would find yourself with little to no capacity remaining to work on those features that competitively differentiate your product.

The impact of these decisions on scalability is obvious. The more time an organization spends on supporting existing solutions, the less time it has to spend on creating growth and scale. As we will discuss in Chapter 20, Designing for Any Technology, two keys to architecting scalable solutions are to always ensure that you

can scale horizontally (“out rather than up”) and agnostically. Scaling agnostically requires that we design solutions such that components are replaceable; in other words, we should be able to obtain authentication solutions, databases, load balancers, and persistence engines from a number of vendors. When architectures are designed to scale horizontally and agnostically, the question of whether to build or buy something becomes one of competitive differentiation, company focus, and cost.

Anytime you buy rather than build, you free up engineering resources that can be applied to differentiating and growing your business. Engineers are one of your most precious and critical resources—why would you ever want to focus them on things that do not create superordinate shareholder value?

You may recall from past education or even from some of the earlier discussions in this book that shareholder value is most closely associated with the profits of the company. Rising profits often result in changes to dividends, increases in stock prices, or both. Profits, in turn, are a direct result of revenues minus costs. As such, we need to focus our build versus buy discussions along the paths of decreasing cost and increasing revenue through focusing on strategy and competitive differentiation.

Focusing on Cost

Cost-focused approaches center on lowering the total cost to the company for any build versus buy analysis. These approaches range from a straight analysis of total capital employed over time to a discounted cash flow analysis that factors in the cost of capital over time. Your finance department likely has a preferred method for deciding how to determine the lowest-cost approach.

Our experience in this area is that most technology organizations have a bias toward building components. This bias most often shows up in an incorrect or incomplete analysis showing that building a certain system is actually less expensive to the company than purchasing the same component. The most common mistakes in this type of analysis are underestimating the initial cost of building the component, and missing or underestimating the future costs of maintenance and support. It is not uncommon for a company to underestimate the cost of support by an order of magnitude, as it does not have the history or institutional DNA to know how to truly develop or support critical infrastructure on a 24/7 basis.

If you adopt a cost-focused strategy for the build versus buy analysis of any system, a good way to test whether your strategy is working is to evaluate how often the process results in a build decision. Your decision process is probably spot on if nearly all decisions result in a buy decision. The exception to this rule is in the areas where your company produces the product in question. Obviously, you are in

business to make money, and to make money you must produce something or provide a service to someone.

A major weakness of cost-focused strategies is that they do not focus on strategic alignment or competitive differentiation. The focus is solely on reducing or limiting the cost incurred by the company for anything that is needed from a technology perspective. Very often, this strategy is employed by groups implementing back-office information technology systems. Focusing on cost alone, though, can lead to decisions to build something when a commercial off-the-shelf (COTS) or vendor-provided system will be more than adequate.

Focusing on Strategy

Strategy-focused approaches look at the build versus buy decision from a perspective of alignment with the organization's vision, mission, supporting strategy, and goals. In most cases, a two-part question is asked with the strategy-focused approach:

- Are we the best or among the best (top two or three) providers or builders of the technology in question?
- Does building or providing the technology in question provide sustainable competitive differentiation?

To be able to answer the first question, you need to be convinced that you have the right and best talent to be the best at what you are doing. Unfortunately, we find that too many technology organizations believe that they are the best at providing, well, *you name it!* Not everyone can be the best at everything. If your company doesn't specialize in something, you will not be the best in anything; and if you believe you are the best in everything, you are guaranteed, due to lack of focus, to be the best at nothing for a very long time. In the real world, only two or three providers of anything can claim to be the best or at least in the top two to three. Given the number of candidates out there for nearly any service or component, unless you are the first provider of some service, the chances are slim that your team really is the best. It can be good, but it probably is not the best.

To be able to answer the second question, you need to be able to explain how, by building the system in question, you are raising switching costs, lowering barriers to exit, increasing barriers to entry, and the like. How are you making it harder for your competitors to compete against you? How does building the system help you win new clients, keep existing clients, and operate more cost-effectively than any of your competitors? What keeps them from copying what you are doing in very short order?

“Not Built Here” Phenomenon

If we seem a little negative in this chapter, it is because we are. We see a lot of value destroyed in a lot of companies from bad build versus buy decisions. In our consulting practice, AKF Partners, we’ve had clients running commerce sites that built their own databases from the ground up! And these aren’t slightly modified open source databases—they are brand-new solutions that only the company owns and uses. It is very common to find software developers building their own load balancers instead of using any of the myriad of offerings from providers that specialize in this technology. Most often, the argument is “We can build it better” or “We needed something better, so we built it,” followed closely by “It was cheaper for us to build it than to buy it.”

We call this the “Not Built Here” (NBH) phenomenon. Not only is the NBH attitude dangerous from the perspective of scalability, it is crippling from a shareholder perspective. When applied to very small things that take only a portion of your development capacity, it is just an annoyance. When applied to critical infrastructure, this mindset very often becomes the source of the company’s scalability crisis. Too much time is spent managing the proprietary system that provides “incredible shareholder value,” and too little time is devoted to making and creating business functionality and working to really scale the platform.

To clarify this point, let’s take a well-known real-world example like eBay. If eBay had a culture that eschewed the use of third-party or COTS products, it might focus on building critical pieces of its software infrastructure such as application servers. Application servers are a commodity and can typically be acquired and implemented at very little cost. Assuming that eBay spends 6% to 8% of its revenue on building applications critical to the buying and selling experience, a portion of that amount will now be spent on building and maintaining its proprietary application server. As a consequence, either less new product functionality will be created for that 6% to 8%, or eBay will need to spend more than 6% to 8% of its revenue to both maintain its current product roadmap and build its proprietary application server. Either way, shareholders suffer. eBay, by the way, does not have such a culture; in fact, it has a very robust build versus buy analysis process to keep just such problems from happening.

Merging Cost and Strategy

Now that we’ve presented the two most common approaches to analyzing build versus buy decisions, we’d like to present what we believe to be the most appropriate solution. Cost-centric approaches miss the question of how a potential build decision supports the company’s objectives and do not consider the lost opportunity of development associated with applying finite resources to noncompetitive differentiating

technologies. Strategy-centric approaches fail to completely appreciate the costs of such a decision and as such may end up being dilutive to shareholder value.

The right approach is to merge the two approaches and develop a set of tests that can be applied to nearly any build versus buy decision. We also want to acknowledge a team's and company's natural bias to build—and we want to protect against it at all costs. We've developed a very simple, non-time-intensive, four-part test to help decide whether you should build or buy the "thing" you are considering.

Does This Component Create Strategic Competitive Differentiation?

This is one of the most important questions within the build versus buy analysis process. At the heart of this question is the notion of shareholder value. If you are not creating something that will lead to competitive differentiation, thereby making it more difficult for your competition to win deals or steal customers, why would you possibly want to build the object in question? Building something that makes your system "faster" or reduces customer-perceived response time by 200 milliseconds might sound like a great argument for building the component in question, but how easy is it for your competitors to get to the same place? Put another way, is it a sustainable competitive advantage? Does 200 milliseconds really make a big difference in customer experience?

Are you increasing switching costs for your customers and making it harder for them to leave your product or platform? Are you increasing the switching costs for your suppliers? Are you changing the likelihood that your customers or suppliers will use substitutes rather than your product or that of a competitor? Are you decreasing exit barriers for the competition or making it harder for new competitors to compete against you? These are only a few of the questions you should be able to answer to be comfortable with a build versus buy decision. In answering these questions or going through a more formal analysis, recognize your natural bias toward believing that you can create competitive differentiation. You should answer "No" more often than "Yes" to this question and stop your analysis in its tracks. There is no reason to incur the lost opportunity associated with dedicating engineers to something that will not make your company significantly better than its competition.

Are We the Best Owners of This Component or Asset?

Simply stated, do you really have the right team to develop and maintain this item? Can your support staff provide the support you need when the solution breaks? Can

you ensure that you are always covered? Very often, a good question to truly test this ability is this: “If you are the best owner of this asset, should you consider selling it?” Think long and hard about this follow-up question, because many people get the answer wrong or at best half right.

If you answer, “No, we won’t sell it, because it creates differentiation for us and we want to win with our primary product,” you are only half right. A fully correct answer would also include “The value in selling it does not offset the cost of attempting to sell it,” or something along those lines.

Here’s something else to consider: Given that there can be only one company, team, or entity that’s best at any given “thing,” how likely is it that your team can be the best at both what you do to make money and the new component you are considering building? If your organization is a small company, the answer is “Very unlikely.” If it is statistically unlikely you are the best at the thing you started on, how can it be more probable that you will be the best at both that old thing and this new thing?

If your organization is an online commerce company, it’s entirely possible that it is the best at logistics planning or the best at presenting users with what they are most likely to buy. It is not very likely at all that the company can be the best at one of those things *and* be the best developer of databases for its internal needs or the best fraud detection system, or the best at developing its special firewall or its awesome load balancer. More than likely someone else is doing it better.

What Is the Competition for This Component?

If you have gotten this far in the questionnaire, you already believe that the component you are building creates competitive differentiation and that your company is the best owner and developer of the component. Now the question becomes, “How much differentiation can we truly create?” To answer it, you really need to dig in and find out who is doing what in this space and ensure that your component is so sufficiently different from its competitors as to justify using your valuable engineering resources for its development and maintenance. Recognize that over time most technologies become commodities, meaning that the feature set converges with very little differentiation from year to year and that buyers purchase mostly on price. How long do you have before a competing builder of this technology can offer it to your primary competitors at a lower cost than you need to maintain your proprietary system?

Can We Build This Component Cost-Effectively?

Our last question concerns costs. We hinted at the cost component within the analysis of the existing competition for our new component, but here we are talking about

the full-fledged analysis of costs over time. Ensure that you properly identify all of the maintenance costs that you will incur. Remember that you need at least two engineers to maintain anything, even if at least part-time, as you need to ensure that someone is around to fix problems when the other person is on vacation. Evaluate your past project delivery schedules and make sure you adjust for the likelihood that you are overly aggressive in your commitments. Are you generally off by 10% or 100%? Factor that into the cost analysis.

Make sure you treat the analysis as you would a profit and loss statement. If you are dedicating engineers to this project, which projects are they not working on and which revenues are you deferring as a result, or which scalability projects won't get done and how will that impact your business?

Checklist: Four Simple Questions

Use this simple checklist of four questions to help you in your build versus buy decisions:

- Does this component create strategic competitive differentiation? Are we going to have long-term sustainable differentiation as a result of this in terms of switching costs, barriers to entry, or something else?
- Are we the best owners of this component or asset? Are we the best equipped to build it and why? Should we sell it if we build it?
- What is the competition for this component? How soon until the competition catches up to us and offers similar functionality?
- Can we build this component cost-effectively? Are we reducing our costs and creating additional shareholder value, and are we avoiding losing opportunities for revenues?

Remember that you are always likely to be biased toward building, so do your best to protect against that bias. The odds are against you if you try to build a better product than those already available. Instead, you should tune your bias toward continuing to do what you do well today—your primary business.

The Best Buy Decision Ever

Seattle Computer Products (SCP) was one of the first manufacturers of computer systems based on the Intel 8086 processor, shipping some of its first CPU boards to customers in 1979.¹ Chances are that you've never heard of this company. But

1. Seattle Computer Products. *Wikipedia*. http://en.wikipedia.org/wiki/Seattle_Computer_Products.

we guarantee you that you know at least a portion of its story, because SCP had a fundamental role in the creation of one of the largest and most successful technology companies of all time.

In 1980, as IBM was starting to look for partners to produce software for its new PCs ("Project Chess"), it came calling on Bill Gates at Microsoft to determine if Microsoft would be willing to license its programming languages (e.g., Basic, Fortran, Pascal, COBOL) to IBM. IBM also enquired as to whether Microsoft would be willing to develop an operating system. Microsoft didn't have any operating systems at the time—the company focused on building programming languages—and it referred IBM to the largest operating system vendor at the time, DRI. Accounts vary as to why DRI didn't win the contract initially, with most of them circling around the unavailability of the owner for discussions. IBM was in a hurry and came back to Microsoft with its need for an operating system.²

Gates and Paul Allen discussed the opportunity. Microsoft didn't have the in-house expertise to build the operating system in a time frame that would meet IBM's need. As most folks with a computer science degree will understand, the development of a compiler or interpreter is significantly different from the development of an operating system. But Allen did know that Tim Paterson of SCP, while waiting for DRI to port its Control Program for Microcomputers (CP/M) operating system to the 8086, had developed his own "quick and dirty" operating system dubbed "QDOS." Microsoft initially licensed the software for \$10,000 and ultimately purchased it outright for \$50,000 while also hiring Paterson away from SCP.

We think you know the rest of the story: Microsoft went on to dominate the PC industry. Looking through our questions, you can see several elements. Even though this was clearly a strategic move for Microsoft, the company's leaders knew that they didn't have the right skills in house to do it right. So rather than "build it," they "bought it." But in this case, they bought the whole product and the experience along with it.

Anatomy of a Build-It-Yourself Failure

Not every poor decision to "build-it-yourself" results in immediate failure. In fact, most such mistakes don't really begin to impact the company for a number of years. Maintenance of legacy products or components starts to steal engineering capacity necessary to fuel new growth. The drain on resources that these components represent

2. The rise of DOS: How Microsoft got the IBM PC OS contract. *PC Magazine*, August 10, 2011. <http://forwardthinking.pcmag.com/software/286148-the-rise-of-dos-how-microsoft-got-the-ibm-pc-os-contract>.

causes a lack of focus in the company. Apple Computer circa 1996 and Steve Jobs's NeXT are both emblematic of the long-term results of poor build decisions and the lack of focus and constraint in engineering resources that such decisions can create.

The "Classic Mac OS" that premiered in 1984 had, by the early 1990s, started to become bloated and slow. In addition to being slow, it lacked critical features such as memory protection and preemptive multitasking. Apple as a company probably had little choice except to focus on both hardware and the operating system for the original Macintosh. But Apple's attention span drifted into creating multiple versions of the Macintosh and several peripherals, creating an absence of focus and sparse resources applied to any single product.^{3,4} Apple attempted several times to update its Mac operating system, including the failed Copland initiative of the early 1990s. By 1996, it had become clear to then CEO Gil Amelio that Apple needed to purchase its next operating system.

Enter NeXT, the company Jobs founded after being ousted from Apple. NeXT was experiencing its own issues with lack of focus. Having decided to both create the world's first GUI-enabled UNIX operating system and an innovative hardware solution, NeXT was beset with a number of product delays. Marketing and distribution failures crippled the company's ability to persuade customers to adopt its hardware solutions. Businessland, once heralded as the largest retailer of personal computers, had entered into a relationship to distribute the NeXT platform but shortly thereafter filed for bankruptcy, leaving NeXT with no distribution partners of any significance.⁵ Jobs was largely funding NeXT with his own resources; given the company's cash burn rate, he needed to focus on doing one thing well. While he considered it a personal failure, he decided to focus on the NeXT operating system (NeXTStep) and creating licensing relationships for distributing the operating system for use on other platforms. In 1993, NeXT exited the hardware business.

NeXT's operating and financial problems collided with Apple's need for a next-generation operating system. Both companies had suffered from a lack of focus. NeXT was a limited-resources startup attempting to create an innovative hardware and operating system. Apple was a technology behemoth spread thinly across multiple platforms, software, and devices. Neither company was successful attempting to do everything. The drain of resources was causing growth and viability concerns for both. As Jobs relayed to Walter Isaacson, "Deciding what not to do is as important

-
3. The real leadership lessons of Steve Jobs. *Harvard Business Review*, April 2012. <http://hbr.org/2012/04/the-real-leadership-lessons-of-steve-jobs/>.
 4. Avoiding Copland 2010. *ARS Technica*, September 27, 2005. <http://arstechnica.com/staff/2005/09/1372/>.
 5. Loss could force Businessland into bankruptcy. *New York Times*, May 15, 1991. <http://www.nytimes.com/1991/05/15/business/loss-could-force-businessland-into-bankruptcy.html>.

as deciding what to do.” Apple announced its purchase of NeXT in 1996, ultimately creating an opportunity for Jobs to take control of Apple and “fix it.”⁶

Conclusion

Build versus buy decisions have an incredible capacity to destroy shareholder value if not approached carefully. Incorrect decisions can steal resources to scale your platform, increase your cost of operations, take resources away from critical customer-facing and revenue-producing functionality, and destroy shareholder value. There is a natural bias toward building over buying, but we urge you to guard strongly against that bias.

The two most common approaches to making build versus buy decisions are the cost-centric and strategy-centric strategies. A third approach, merging the benefits of both approaches, avoids most of the problems each approach has individually. You can also use a four-part checklist for your build versus buy decisions to help you make the right choice.

Key Points

- Making poor build versus buy choices can destroy your ability to scale cost-effectively and can destroy shareholder value.
- Cost-centric approaches to build versus buy focus on reducing overall costs but suffer from a lack of focus on lost opportunities and strategic alignment of the component being built.
- Strategy-centric approaches to build versus buy focus on aligning the decision with the long-term needs of the company but do not account for total costs.
- Merging the two approaches results in a four-part test that has the advantages of both approaches but eliminates the disadvantages.
- To reach a build versus buy decision, you should answer the following questions:
 - Does this component create strategic competitive differentiation?
 - Are we the best owners of this asset?
 - What is the competition for the component?
 - Can we build the component cost-effectively?

6. Apple acquires NeXT, Jobs. CNet, December 20, 1996. http://news.cnet.com/Apple-acquires-Next-Jobs/2100-1001_3-256914.html.

Chapter 16

Determining Risk

Hence in the wise leader's plans, considerations of advantage and disadvantage will be blended together.

—Sun Tzu

While we've often mentioned risk management, we have not offered our view of risk and how to manage it. This chapter broadly covers how to determine and manage risk in any technology or business decision. Managing risk is one of the most fundamentally important aspects of increasing and maintaining availability and scalability.

Importance of Risk Management to Scale

Business is inherently a risky endeavor. Some examples of risks are that your business model will not work or will become outdated, that customers won't want your products, that your products will be too costly, or that your products will fall out of favor with customers. To be in business, you must be able to identify and balance the risks with the associated rewards. Pushing a new release, for instance, clearly has inherent risks, but it should also have rewards.

Most organizations look at risk and focus on reducing the probability of an incident occurring. Introducing more testing into the development life cycle is the typical approach to reducing risk. The problem with this approach is that testing can limit the probability of a bug or error to only a finite and non-zero amount. Testing over extended periods of time, such as is done with satellite software, still does not ensure bug-free code. Recall that on September 23, 1999, NASA lost a \$125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while NASA's team used the more conventional metric system for a key spacecraft operation. How was this discrepancy not caught during testing? The answer, as quality professionals know, is that it is mathematically impossible to ensure defect-free software for even moderately complex systems.

At AKF Partners, we prefer to view risk as a multifactor product, depicted in Figure 16.1. We divide risk into the probability of an incident occurring and the impact of an incident should it occur. The probability of an incident is driven partly by the amount of change involved in any release and the amount of testing performed on that change. The greater the effort or amount of change, the higher the probability of an incident. The more testing that we do, the lower the probability that we will have an incident. Key efforts to drive down the probability of incidents should include smaller releases and more effective testing (such as through thorough automation).

Impact, in contrast, is driven by the breadth of the incident (measured by percentage of customers impacted or percentage of transactions) and the duration of the incident. As Figure 16.1 shows, architectural decisions such as fault isolation and x -, y -, and z -axis splits (all covered in Part III, “Architecting Scalable Solutions”) help drive down the breadth of impact. Effective monitoring, as well as problem and incident management, helps reduce the duration of an impact. We covered these areas in Part I, “Staffing a Scalable Organization.”

If business is inherently risky, are successful companies then simply better at managing risk? We think that the answer is that these companies are either effective at managing risk or have just been lucky thus far. And luck, as we all know, runs out sooner or later.

Being simply “lucky” should have you worried. One can argue that risk demonstrates a Markov property, meaning that the future states are determined by the

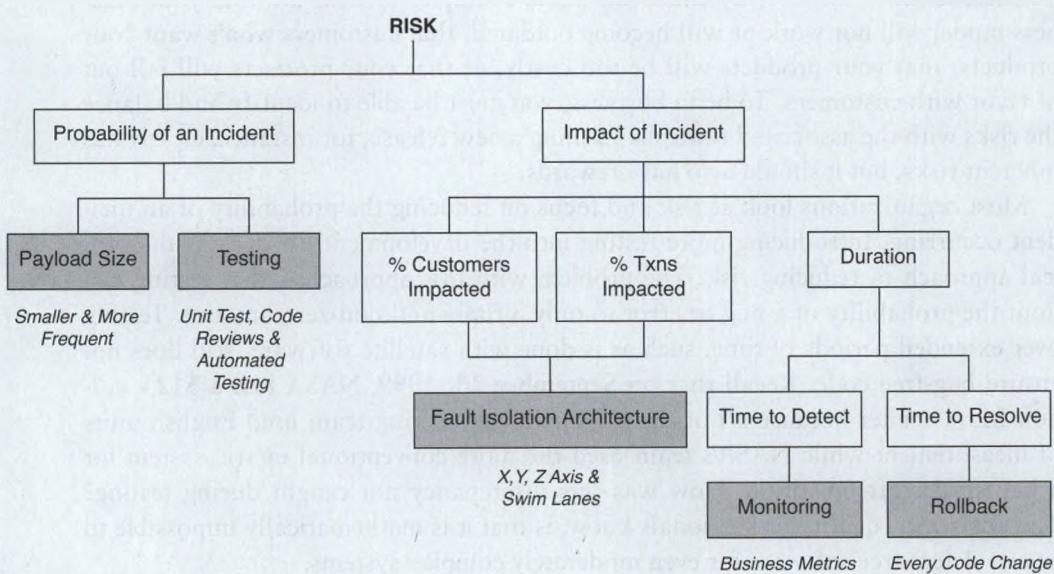


Figure 16.1 Risk Model

present state and are independent of past states. We would argue that risk is cumulative to some degree, perhaps with an exponential decay but still additive. A risky event today can result in failures in the future, either because of direct correlation (e.g., today's change breaks something else in the future) or via indirect methods (e.g., an increased risk tolerance by the organization leads to riskier behaviors in the future). Either way, actions can have near- and long-term consequences.

Some people can naturally feel and manage risk. These people may have developed this skill from years of working around technology. They also might just have an innate ability to sense risk. While having such a person is great, such an individual still represents a single point of failure; as such, we need to develop the rest of the organization to better measure and manage risk.

Because risk management is important to scalability, we need to understand the components and steps of the risk management process. There are many ways to go about trying to accurately determine risk—some more involved than others, and some often more accurate than others. The important thing is to select the right process for your organization, which means balancing the rigor and required accuracy with what makes sense for your organization. After estimating the amount of risk, you must actively manage both the acute risk and the overall risk. *Acute risk* is the amount of risk associated with a particular action, such as changing a configuration on a server. *Overall risk* is the amount that is cumulative within the system because of all the actions that have taken place over the previous days, weeks, or possibly even months.

Measuring Risk

The first step in being able to manage risk is to—as accurately as *necessary*—determine what amount of risk is involved in a particular action. The reason we use the term *necessary* and not *possible* is that you may be able to more accurately determine risk, but it might not be necessary given the current state of your product or your organization. For example, a product in beta testing, where customers expect some glitches, may dictate that a sophisticated risk assessment is not necessary and that a cursory analysis is sufficient at this point. There are many different ways to analyze, assess, or estimate risk. The more of these approaches that are in your tool belt, the more likely you are to use the most appropriate one for the appropriate time and activity. Here, we will cover three methods of determining risk. For each method, we will discuss its advantages, disadvantages, and level of accuracy.

The first approach is the *gut feel* method. People often use this method when they believe they can *feel* risk, and are given the authority to make important decisions regarding risk. As we mentioned earlier, some people inherently have this ability, and it is certainly great to have someone like this in the organization. However, we would caution you about two very important concerns. First, does this person *really* have

the ability to understand risk at a subconscious level, or do you just wish he did? In other words, have you tracked this person's accuracy? If you haven't, you should do so before you consider this as anything more than a method of guessing. If you have someone who claims to "feel" the risk level, have that person write his or her predictions on the team whiteboard . . . for fun. Second, heed our prior warning about single points of failure. You need multiple people in your organization to understand how to assess risk. Ideally, everyone in the organization will be familiar with the significance of risk and the methodologies that exist for assessing and managing it.

Thin Slicing

Thin slicing is a term used in psychology and philosophy to describe the ability to find patterns in events based only on "thin slices," or narrow windows, of experience. Malcolm Gladwell, in his book *Blink: The Power of Thinking Without Thinking*, argues that this process of spontaneous decision making is often as good as, or even better than, making carefully planned and considered decisions.

Research in classrooms has shown that experts can distinguish biased teachers from unbiased teachers from brief clips of teachers' behaviors. Additionally, research in the courtroom has shown that brief excerpts of judges' instructions to jurors in trials can allow experts to predict the judge's expectations for the trial.¹

Gladwell claims that experts often make better decisions with snap judgments than they do with volumes of analysis. Sometimes having too much information can interfere with the accuracy of a judgment, resulting in the colloquialism of "analysis paralysis." While this ability to make decisions based on very limited information seems ideal, Gladwell also cautions that an expert's ability to thin slice can be corrupted by the individual's likes, dislikes, prejudices, and stereotypes.

The key advantage of the gut feel method of risk assessment is that it is very fast. A true expert who fundamentally understands the amount of risk inherent in certain tasks can make decisions in a matter of a few seconds. The disadvantages of the gut feel method include that, as we discussed, the person might not have this ability but may be fooled into thinking he does because of a few key saves. Another disadvantage is that this method is rarely replicable. People tend to develop this ability over years of working in the industry and honing their expertise; it is not something that can be taught in an hour-long class. Yet another disadvantage of this method is that it leaves a lot of decision making up to the whim of one person as opposed to a team or

1. Albrechtsen, J. S., Meissner, C. A., & Susa, K. J. Can intuition improve deception detection performance? *Journal of Experimental Social Psychology*, 2009;45(4):1052–1055.

group that can brainstorm and troubleshoot data and conclusions. The accuracy of this method is highly variable depending on the person, the action, and a host of other factors. This week a person might be very good at assessing the risk, but next week she might strike out completely. As a result, you should use this method sparingly, when time is of the essence, risk is at worst moderate, and a time-proven expert is available.

The second method of measuring risk is the *traffic light* method. In this method, you determine the risk of an action by breaking down the action into the smallest components and assigning a risk level to them of green, yellow, or red. The smallest component could be a feature in a release or a configuration change in a list of maintenance steps. The granularity depends on several factors, including the time available and the amount of practice the team has in performing these assessments. Next we determine the overall or collective risk of the action. Assign a risk value to each color, count the number of each color, and multiply the count by the risk value. Then, sum these multiplied values and divide by the total count of items or actions. Whatever risk value the result is closest to gets assigned the overall color. Figure 16.2 depicts the risk rating of three features that provides a cumulative risk of the overall release.

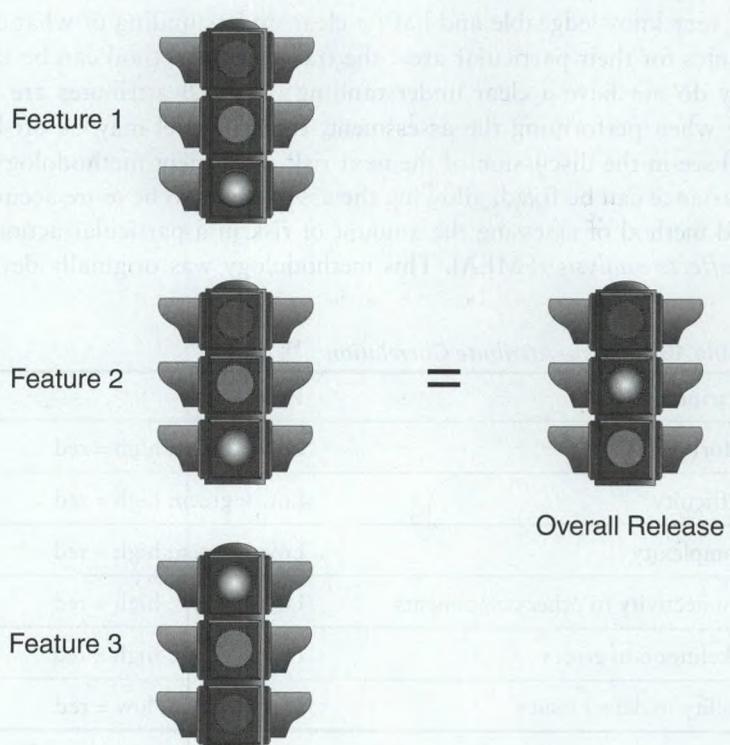


Figure 16.2 Traffic Light Method of Risk Assessment

Someone who is intimately familiar with the micro-level components should assess risk values and assign colors for individual items in the action. This assignment should be made based on the difficulty of the task, the amount of effort required for the task (the more effort, generally the higher the risk), the interaction of this component with others (the more connected or centralized the item, the higher the risk), and so on. Table 16.1 shows some of the most common attributes and their associated risk factors that can be used by engineers or other experts to gauge the risk of a particular feature or granular item in the overall list.

One significant advantage of the traffic light method is that it begins to become methodical, which implies that it is repeatable, able to be documented, and able to be trained. Repeatability, in turn, implies that we can learn and improve upon the results. Many people can conduct the risk assessment, so you are no longer dependent on a single individual. Again, because many people can perform the assessment, there can be discussion about the decisions that people arrive at, and as a group they can decide whether someone's argument has merit. The disadvantage of this method is that it takes more time than the gut feel method and is an extra step in the process. Another disadvantage is that it relies on each expert to choose which attributes he or she will use to assess the risk of individual components. Because of this possible variance among the experts, the accuracy of this risk assessment is mediocre. If the experts are very knowledgeable and have a clear understanding of what constitutes risky attributes for their particular area, the traffic light method can be fairly accurate. If they do not have a clear understanding of which attributes are important to examine when performing the assessment, the risk level may be off by quite a bit. We will see in the discussion of the next risk assessment methodology how this potential variance can be fixed, allowing the assessments to be more accurate.

The third method of assessing the amount of risk in a particular action is *failure mode and effects analysis* (FMEA). This methodology was originally developed for

Table 16.1 Risk-Attribute Correlation

Attribute	Risk
Effort	Low = green; high = red
Difficulty	Low = green; high = red
Complexity	Low = green; high = red
Connectivity to other components	Low = green; high = red
Likelihood of errors	Low = green; high = red
Ability to detect issues	High = green; low = red

use by the military in the late 1940s.² Since then, it has been used in a multitude of industries, including automotive, manufacturing, aerospace, and software development companies. The method of performing the assessment is similar to the traffic light method, in that components are broken up into the smallest parts that can be assessed for risk; for a release, this could be features, tasks, or modules. Each of these components is then identified with one or more possible failure modes. Each failure mode has an effect that describes the impact if this particular failure were to occur.

For example, a signup feature may fail by not storing the new user's information properly in the database, by assigning the wrong set of privileges to the new user, or through several other failure scenarios. The effect would be the user not being registered or having the ability to see data she was not authorized to see. Each failure scenario is scored on three factors: likelihood of failure, severity of that failure, and the ability to detect if that failure occurs (note the similarity to the risk model identified in Figure 16.1). We choose to use a scoring scale of 1, 3, and 9 for these elements because it allows us to be very conservative and differentiate items with high risk factors well above those items with medium or low risks. The *likelihood* of failure is essentially the probability of this particular failure scenario coming true. The *severity* of the failure is the total impact to the customer and the business if the failure occurs. This impact can take the form of monetary losses, loss of reputation (goodwill), or any other business-related measurement. The *ability to detect the failure* rates whether you will be likely to notice the failure if it occurs. As you can imagine, a very likely failure that has disastrous consequences and that is practically undetectable is the worst possible outcome.

After the individual failure modes and effects have been scored, the scores are multiplied to provide a total risk score—that is, likelihood score \times severity score \times ability to detect score. This score shows the overall risk that a particular component has within the overall action.

The next step in the FMEA process is to determine the mitigation steps that you can perform or put in place that will lower the risk of a particular factor. For instance, if a component of a feature had a very high ability to detect score, meaning that it would be hard to notice if the event occurred, the team might decide ahead of time to write some queries to check the database every hour after the product or service release for signs of this failure, such as missing data or wrong data. This mitigation step has a lowering effect on this risk factor of the component and should then indicate what the risk was lowered to.

In Table 16.2, there are two example features for a human resources management (HRM) application: a new signup flow for the company's customers and changing

2. Procedure for performing a failure mode effect and criticality analysis. November 9, 1949. United States Military Procedure, MIL-P-1629.

Table 16.2 Failure Mode and Effect Analysis Example

Feature	Failure Mode	Effect	Likelihood of Failure Occurring (1 = Low, 3 = Medium, 9 = High)	Severity If Failure Occurs (1 = Minimal 3 = Significant, 9 = Extreme)	Ability to Detect Should Failure Occur (1 = Easy, 3 = Medium, 9 = Difficult)	Total Risk Score	Remediation Actions	Revised Risk Score
Signup	User data not inserted into the database properly	Users not registered	3	3	3	27	Test all registration paths (reduce likelihood score to 1) Write queries to use post-launch for data validation (reduce detection to 1)	3
	Users given the wrong set of privileges	Users have access to other users' information	1	9	3	27	Write a query to run every hour post-launch to check for new registrants with unusual privileges (reduce detection score to 1)	9
	Users not sent passwords	Users unable to log in	3	1	1	3	N/A	3
Credit Card	Credit card billed incorrectly	Charges to credit cards are too much or too little	1	9	3	27	Roll credit card feature out in beta testing for limited customers (reduce severity score to 3)	9
	Authorization number not stored	Unable to recharge credit card without reentering number	1	1	1	1	N/A	1
	Credit card numbers not encrypted	Allows for the possibility of someone grabbing credit card numbers	1	9	1	9	N/A	9

to a new credit card processor. Each of these features has several failure modes identified. Walking through one as an example, let's look at the Credit Card Payment feature and focus on the Credit Card Billed Incorrectly failure mode, which has the effect of either a too-large or too-small payment being charged to the card. In our example, an engineer might have scored this as a 1, or very unlikely to occur. Perhaps this feature received extensive code review and quality assurance testing due to the fact that it was dealing with credit cards, which lowered the risk. The engineer also feels that this failure mode has a disastrous severity, so it receives a 9 for this score. This seems reasonable because a wrongly billed credit card would result in angry customers, costly chargebacks, and potential refunding of license fees. The engineer feels that the failure mode would be of moderate difficulty to detect and so gives it a score of 3 on this variable. The total risk score for this failure mode is 27, arrived at by multiplying $1 \times 3 \times 9$. A remediation action has been identified for this feature set—rolling out the payment processor in beta testing for a limited customer set. Doing so will reduce the severity because the breadth of customer impact will be limited. If this remediation action is taken, the risk would be lowered to a 3 for severity and the revised risk score would be only 9, much better than before.

The advantage of FMEA as a risk assessment process is that it is very methodical, which allows it to be documented, trained, evaluated, and modified for improvement over time. Another advantage is the accuracy. Especially over time as your team becomes better at identifying failure scenarios and accurately assessing risks, this approach will become the most accurate way for you to determine risk. The disadvantage of the FMEA method is that it takes time and thought. The more time and effort put into this analysis, however, the better and more accurate the results. This method is very similar and complementary to test-driven development. Performing FMEA in advance of development allows us to improve designs and error handling.

As we will discuss in the next section, risk measurement scores, especially ones from FMEA, can be used to manage the amount of risk in a system across any time interval or in any one release/action. The next step in the risk assessment is to have someone or a team of people review the assessment for accuracy and to question any decision. This is the great part about using a methodical approach such as FMEA: Everyone can be trained on it and, therefore, can police each other to ensure the highest-quality assessment is performed. The last step in the assessment process is to revisit the assessment after the action has taken place to see how accurate you and the experts were in determining the right failure modes and assessing their factors. If a problem arose that was not identified as possible, have that expert review the situation in detail and provide a reason why this potential scenario was not identified ahead of time and a warning to other experts to watch out for this type of failure.

Risk Assessment Steps

If you are planning on using any methodical approach to risk assessment, these are the steps for a proper risk assessment. These steps are appropriate for the traffic light method or the FMEA method:

1. Determine the proper level of granularity to assess the risk.
2. Choose a method that you can reproduce.
3. Train the individuals who will be performing the risk assessment.
4. Have someone review each assessment; alternatively, the team can review the entire assessment.
5. Choose an appropriate scoring scale (e.g., 1, 3, 9) that takes into account how conservative you need to be.
6. Review the risk assessments after the action, release, or maintenance has occurred to determine how good the risk assessment was at identifying the types of failures as well as how likely, severe, and detectable they were.

Whether you are using the traffic light method, FMEA, or another risk assessment methodology, be sure to follow these steps to ensure a successful risk assessment that can be used in the overall management of risk.

Managing Risk

We fundamentally believe that risk is cumulative. The greater the unmitigated risks you take, the more likely you are to experience significant problems. We teach our clients to manage both acute and overall risk in a system. Acute risk is how much risk exists from a single change or combination of changes in a release. Overall level of risk represents the accumulation of risk over hours, days, or weeks of performing risky actions on the system. Either type of risk—acute or overall—can result in a crisis scenario.

Acute risk is managed by monitoring the risk assessments performed on proposed changes to the system, such as releases. You may want to establish ahead of time some limits to the amount of risk that any one concurrent action can have or that you are willing to allow at a particular time of day or customer volume (review Chapter 10, Controlling Change in Production Environments, for more details). For instance, you may decide that any single action associated with a risk greater than 50 points, as calculated through the FMEA methodology, must be remediated below

this amount or split into two separate actions. Alternatively, you may want only actions scoring below 25 points to take place on the system before midnight; everything higher must occur after midnight. Even though this is a discussion about the acute risk of a single action, this, too, is cumulative, in that the more risky items contained in a risk, the higher the likelihood of a problem, and the more difficult the detection or determination of the cause because so many things changed.

As a thought experiment, imagine a release with one feature that has two failure modes identified, compared to a release with 50 features, each with two or more failure modes. First, it is far more likely for a problem to occur in the latter case because of the number of opportunities. As an analog, consider flipping 50 pennies at the same time. While each coin has an independent probability of landing on heads, you are more likely to have at least one heads in the total results. Second, with 50 features, the likelihood of changes affecting one another or touching the same component, class, or method in an unexpected way is higher. Therefore, both from a cumulative opportunity standpoint and from a cumulative probability of negative interactions, there is an increased likelihood of a problem occurring in the 50-feature case compared to the one-feature case. If a problem arises after these releases, it will also be much easier to determine the cause of the problem when the release contains one feature than when it contains 50 features, assuming that all the features are somewhat proportional in complexity and size.

For managing acute risk, we recommend that you construct a chart such as the one in Table 16.3 that outlines all the rules and associated risk levels that are acceptable. This way, the action for each risk level is clear-cut. You should also establish an exceptions policy—for example, anything outside of these rules must be approved by the VP of engineering *and* the VP of operations *or* the CTO alone.

For managing overall risk, there are two factors you should consider. The first is the cumulative amount of changes that have taken place in the system and the corresponding increase in the amount of risk associated with each of these changes.

Table 16.3 Acute Risk Management Rules

Rule	Risk Level
New feature release	< 150 points
Bug fix release	< 50 points
6 a.m.–10 p.m.	< 25 points
10 p.m.–6 a.m.	< 200 points
Maintenance patches	< 25 points
Configuration changes	< 50 points

Just as we discussed in the context of acute risk, combinations of actions can have unwanted interactions. The more releases or database splits or configuration changes that are made, the more likely that one will cause a problem or the interaction of them will cause a problem. If the development team has been working in a development environment with a single database and two days before the release the database is split into a master and a read host, it's fairly likely that the next release will have a problem unless a ton of coordination and remediation work has been done.

The second factor that should be considered in the overall risk analysis is the human factor. As people perform increasingly more risky activities, their level of risk tolerance goes up. This human conditioning can work for us very well when we need to become adapted to a new environment, but when it comes to controlling risk in a system, it can lead us astray. If a saber-toothed tiger has moved into the neighborhood and you still have to leave your cave each day to hunt, the ability to adapt to the new risk in your life is critical to your survival. Otherwise, you might stay in your cave all day and starve. Conversely, adding risk because you have yet to be eaten and feel invincible is a good way to cause serious issues.

We recommend that to manage the overall amount of risk in a system, you adopt a set of rules such as those shown in Table 16.4. This table lays out the amount of risk, as determined by FMEA, for specific time periods. If you are using a different methodology than FMEA, you need to adjust the risk level column with some scale that makes sense. For instance, instead of "less than 150 points" you could use "fewer than 5 green or 3 yellow actions." As in the acute risk management process, you will need to account for objections and overrides. You should plan ahead and have an escalation process established. One idea might be that a director can grant an extra 50 points to any risk level, a VP can grant 100 points, and the CTO can grant 250 points, but these additions are not cumulative. Any way you decide to set up this set of rules, what matters most is that it makes sense for your organization and that it is documented and strictly adhered to. As another example, suppose a feature

Table 16.4 Overall Risk Management Rules

Rules	Risk Level
6-hour period	< 150 points
12-hour period	< 250 points
24-hour period	< 350 points
72-hour period	< 500 points
7-day period	< 750 points
14-day period	< 1,200 points

release requires a major database upgrade. The combined FMEA score is 200, which exceeds the maximum risk for a feature release (150 points in Table 16.3). The CTO can then either approve the additional risk or else require the database upgrade to be done separately from the code release.

Conclusion

In this chapter, we focused on risk. Our discussions started by exploring the purpose of risk management and how it relates to scalability. We concluded that risk is prevalent in all businesses, but especially startups. To be successful, you have to take risks in the business world. In the SaaS world, scalability is part of this risk–reward structure. You must take risks in terms of your system’s scalability or else you will overbuild your system and not deliver products that will make the business successful. By actively managing your risk, you can increase the availability and scalability of your system.

Although many different approaches are used to assess risk, this chapter examined three specific options. The first method is gut feeling. Unfortunately, while some people are naturally gifted at identifying risk, many others are credited for but actually lack this ability.

The second method is the traffic light approach, which assessed components as low risk (green), medium risk (yellow), or high risk (red). The combination of all components in an action, release, change, or maintenance provides the overall risk level.

The third approach is the failure mode and effect analysis methodology—our recommended approach. In this method, experts are asked to assess the risk of components by identifying the failure modes that are possible with each component or feature and the impending effect that this failure would cause. For example, a credit card payment feature might fail by charging a wrong amount to the credit card, with the effect being a charge to the customer that is too large or too small. These failure modes and effects are scored based on their likelihood of occurrence, the severity if they were to occur, and the ability to detect if they did occur. These scores are then multiplied to provide a total risk score. Using this score, the experts would then recommend remediation steps to reduce the risk of one or more of the factors, thereby decreasing the overall risk score.

After assessing risk, we must manage it. This step can be broken up into the management of acute risk and the management of overall risk. Acute risk deals with single actions, releases, maintenances, and so on, whereas overall risk deals with all changes over periods of time such as hours, days, or weeks. For both acute and overall risk, we recommend adopting rules that specify predetermined amounts of risk that will be tolerated for each action or time period. Additionally, in preparation for

objections, an escalation path should be established ahead of time so that the first crisis does not create its own path without thought and proper input from all parties.

As with most processes, the most important aspect of both risk assessment and risk management is the fit within your organization at this particular time. As your organization grows and matures, you may need to modify or augment these processes. For risk management to be effective, it must be used; for it to be used, it needs to be a good fit with your team.

Key Points

- Business is inherently risky; the changes that we make to improve scalability of our systems can be risky as well.
- Managing the amount of risk in a system is key to availability and ensuring the system can scale.
- Risk is cumulative, albeit with some degree of degradation occurring over time.
- For best results, you should use a method of risk assessment that is repeatable and measureable.
- Risk assessments, like other processes, can be improved over time.
- There are both advantages and disadvantages to all of various risk assessment approaches.
- There is a great deal of difference in the accuracy of various risk assessment approaches.
- Risk management can be viewed as addressing both acute risk and overall risk.
- Acute risk management deals with single instances of change, such as a release or a maintenance procedure.
- Overall risk management focuses on watching and administering the total level of risk in the system at any point in time.
- For the risk management process to be effective, it must be used and followed.
- The best way to ensure a process is adhered to is to make sure it is a good fit with the organization.

Chapter 17

Performance and Stress Testing

If you know neither the enemy nor yourself, you will succumb in every battle.

—Sun Tzu

After peripherally mentioning performance and stress testing in previous chapters, we now turn our full attention to these tests. In this chapter, we discuss how these tests differ in purpose and output as well as how they impact scalability. Whether you use performance tests, stress tests, neither, or both, this chapter should give you some fresh perspectives on the purpose and viability of testing that you can use to either revamp or initiate a testing process in your organization.

As is the case with the quality of your product, scalability is something that must be designed early in the development life cycle. Testing, while a necessary evil, is an additional cost in our organizations meant to uncover problems and oversights with our designs.

Performing Performance Testing

Performance testing covers a broad range of engineering evaluations, where the emphasis is on the final measurable performance characteristics instead of the actual material or product.¹ With respect to computer science, performance testing focuses on determining the speed, throughput, or effectiveness of a device or piece of software. Performance testing is often called load testing; to us, these two terms are interchangeable. Some professionals will argue that performance testing and load testing have different goals but similar techniques. To avoid a pedantic argument, we will use a broader goal for defining performance testing so that it incorporates both.

1. Performance testing. *Wikipedia*. http://en.wikipedia.org/wiki/Performance_testing.

According to our definition, the goal of performance testing is to identify, document, and, where possible, eliminate bottlenecks in the system. This is done through a strict controlled process of measurement and analysis. Load testing is utilized as a method in this process.

Handling the Load with Load Testing

Load testing is the process of putting load or user demand on a system to measure its response and stability, the purpose of which is to verify that the application can meet the desired performance objectives, which are often specified in a service level agreement (SLA). A load test measures such things as response time, throughput, and resource utilization. It is not intended to identify the system's breaking point unless this point occurs below the peak load condition that is expected by the specifications, requirements, or normal operating conditions. If that should occur, you have a serious issue that must be addressed prior to release.

Example load tests include the following

- Test a mail server with the load of the expected number of users' email accounts.
- Test the same mail server with the expected load of email messages.
- Test a SaaS application by sending many and varied simulated user requests to the application over an extended period of time—the more like production traffic, the better.
- Test a load-balanced pair of app servers with a scaled-down load of user traffic.

Establish Success Criteria

The first step in performance testing is to establish the success criteria. For SaaS systems, this is often based on the concurrent usage and response time metrics. For existing solutions, most companies use baselines established over time in a production environment and/or previous tests within a performance or load testing environment. For new products, you should increase demand on the solution until either the product stops responding or it responds in an unpredictable or undesirable manner. This becomes the benchmark for the as-yet-unreleased new product.

When replacing systems, the benchmarks of the old (to-be-replaced system) are often used as a starting point for the expectations of the new system. Usually such replacements are predicated on creating greater throughput for the purpose of reducing costs at equivalent transaction volumes or to allow the company to grow more cost-effectively in the future.

Establish the Appropriate Environment

After establishing a benchmark, the next step is to establish an appropriate environment in which to perform testing. The environment encapsulates the network, servers, operating system, and third-party software contained with the product. The performance testing environment ideally will be separate from other environments, including development, QA, staging, and the like. This separation is important because you need a stable, consistent environment to conduct tests repeatedly over some extended duration. Mixing the performance environment with other environments will mean greater levels of changes to the environment and, as a result, lower levels of confidence in the results. Furthermore, some of these tests need to be run over extended time periods, such as 24 hours, to produce the load expected for batch routines. As such, the environment will be largely unavailable for use for other purposes for extended periods of time. To achieve the best results, the environment should mirror production as closely as possible, with all of the obvious financial constraints.

The performance testing environment should mimic the production environment to the greatest extent possible because environmental settings, configurations, different hardware, different firewall rules, and much more can all dramatically affect test results. Even different patch versions of the operating system, which might seem a trivial concern, can have dramatically different performance characteristics for applications. This does not mean that you need a full copy of your production environment; although that would be nice, few companies can afford such a luxury. Instead, make wise tradeoffs but stick to the same basic architecture and implementation as much as possible. For example, pools of servers that in production include 40 servers can be scaled down in a test environment to only two or three servers. Databases are often very difficult to scale down because the amount of data affects the query performance. In some cases, you can “trick” the database into believing it has the same amount of data as the production database to ensure the queries execute with the same query plans. Spend some time pondering the performance testing environment, and discuss the tradeoffs that you are making. If you can sufficiently balance the cost with the effectiveness, you will be able to make the best decisions in terms of what the environment should look like and how accurate the results will be.

An additional reason to create a separate test environment may arise if you are practicing continuous delivery or have plans to do so. For your automated systems to be able to easily schedule delivery of packages, they ideally should run those packages through stages of environments, where each stage is focused on some aspect of the potential quality issues (such as a performance testing environment in this case). The more constrained or shared your environments, the more changes that will become backed up waiting for automated environment reconfiguration to perform

automated testing. Splitting out these environments helps ensure a fluid pipeline with fastest possible delivery (assuming no major blocking issues) into the production environment.

Define the Tests

The third step in performance planning is to define the tests. As mentioned earlier, a multitude of tests can be performed on the various services and features. If you try to run all of them, you may never release any products. The key is to use the Pareto distribution, also known as the 80/20 rule: Find the 20% of the tests that will provide you with 80% of the needed information. Product tests almost always follow some similar distribution when it comes to the amount or value of information provided. This situation arises because the features are not all used equally, and some are more critical than others. For example, the feature handling user payments may be more important than the feature handling a user's search for friends, so it should be tested more vigorously.

Vilfredo Pareto

Vilfredo Federico Damaso Pareto (1848–1923) was an Italian economist who was responsible for making several important contributions to economics. One of his most notable insights is the Pareto distribution. Fascinated by power and wealth distribution in societies, Pareto studied property ownership in Italy. He observed in a 1909 publication that 20% of the population owned 80% of the land, thus giving rise to his Pareto distribution.

Technically, the Pareto distribution is a power law of probability distribution, meaning that it states a special relationship between the frequency of an observed event and the size of the event. Another power law is Kleiber's law of metabolism, which states that the metabolic rate of an animal scales to the $\frac{3}{4}$ power of the mass. As an example, a horse that is 50 times larger than a rabbit will have a metabolism 18.8 times greater than the rabbit.

Lots of other rules of thumb exist, but the Pareto distribution is very useful, when it applies, for getting the majority of a result without the majority of the effort. The caution, of course, is to make sure the probability distribution applies before using it. If you have a scenario in which the information is one for one with the action, you cannot get 80% of the information by performing only 20% of the action; instead, you will have to perform the percentage work that you need to achieve the equivalent percentage information.

When you define the tests, be sure to include tests of various types. Some types or categories of tests include endurance, load, most used, most visible, and component

(app, network, database, cache, and storage). An endurance test is used to ensure that a standard load experienced over a prolonged period of time does not have any adverse effects due to such problems as memory leaks, data storage, log file creation, or batch jobs. A normal user load with as realistic traffic patterns and activities as possible is used here. It is often difficult to come up with actual or close-to-actual user traffic. A minimum substitute for this input is a series of actions—such as a signup process followed by a picture upload, a search for friends, and a logout—written into a script that can be executed over and over. A more ideal scenario is to gather actual users' traffic from a network device or app server and replay it in the exact same order while varying the time period. That is, first you can run the test over the same time period in which the users generated the traffic, and then you can increase the speed and ensure the application performs as expected with the increased throughput.

Remember to be mindful of the test definition as it relates to both continuous integration and—more importantly—continuous delivery (refer back to Chapter 5, Management 101, for a definition or refresher on this topic). To be successful with continuous delivery, we need to ensure that the tests we define can be automated and that the success criteria for them can be evaluated by the automation.

Execute the Tests

The load test essentially puts a user load on the system up to the expected or required level to ensure the application is stable and responsive according to internal or external service level agreements. A commonly used test scenario is testing the path that most users take through the application. In contrast, a most visible test scenario is testing the part of the application that is seen the most by users, such as the home page or a new landing page. The component test category is a broad set of tests that are designed to test individual components in the system. One such test might be to exercise a particularly long-running query on the database to ensure it can handle the prescribed amount of traffic. Similarly, traffic requests through a load balancer or firewall are other component tests that you might consider.

In the test execution step, you work through the test plan, executing the tests methodically in the environment established for this testing and recording various measurements such as transaction times, response times, outputs, and behavior. Gather everything that you can. Data is your friend in performance testing. It is important to keep this data from release to release. As described in the next step, comparison between various releases is critical to understanding the data and determining if the data indicates normal operating ranges or the potential presence of a problem.

In organizations practicing continuous delivery, there are a few ways to think about how to execute performance tests. The first is to have the performance testing

occur nearly continuously and outside the critical path to delivery into your production environment. This approach has the beneficial effect of not stalling releases waiting for past submissions to the performance test environment to complete. An unfortunate side effect is that should any release identify a significant performance problem that may cause availability problems, you will not find it prior to release. As a result, outages may increase and availability decrease in exchange for the benefit of decreasing time to market.

A second approach is to have releases move through the performance environment sequentially prior to release. While this protects you against potential performance-related outages, it can significantly decrease your delivery velocity. Imagine that you have several releases in the automated delivery queue. If you will perform endurance tests including overnight batch testing, each may need to wait for its own cycle. The result is that an approach meant to be a faster and low-risk introduction to a production environment starts to slow down relative to your old way of doing things.

A hybrid approach is likely to work best, with some level of testing (exercising the code for a short period of time) done in series prior to release and longer-endurance testing happening with batches of releases once a day. This approach allows you to mitigate much of the risk associated with outages, even as you continue to enjoy the time-to-market benefits of continuous delivery. When practicing the hybrid approach, you will likely need at least two performance testing environments: one for sequential testing (in-line and prior to release) and one for prolonged-endurance testing (post release).

Analyze the Data

Step 5 in the performance testing process is to analyze the data gathered. This analysis can be done in a variety of manners, depending on factors such as the expertise of the analyst, the expectations of thoroughness, the acceptable risk level, and the time allotted. Perhaps the simplest analysis is a comparison of this candidate release with past releases. A query that executes 25 times per second without increased response time in the current release may be a problem if it could execute 50 times per second with no noticeable degradation in performance in the last release. The fun begins in the next step—trying to figure out why this change has occurred. Although decreases in capacity of throughput or increases in response time are clearly items that should be noted for further investigation, the opposite is true as well. A sudden dramatic increase in capacity might indicate that a particular code path has been dropped or SQL conditional statements have been lost; such a change should be noted as a potential target of investigation as well. We hope that in these scenarios an engineer has refactored and improved the performance, but it is best to document this change and ask follow-up questions to confirm it.

A more detailed analysis involves graphing the data for visual reference. Sometimes, it is much easier when data is graphed on line, bar, or pie charts to recognize anomalies or differences. Although these may or may not be truly significant, such graphs are generally quick ways of making judgments about the release candidate.

A further detailed analysis involves performing statistical analysis on the data. Statistical tests such as control charts, *t*-tests, factor analysis, main effects plots, analysis of variance, and interaction plots can all be helpful. These tests help identify the factors causing the observed behavior and help to determine whether you should be concerned about their overall effect.

In the case of continuous delivery, failure of the automated performance test cases (e.g., on a percentage of capacity loss for a given attribute such as queries per second) should either stall the release for evaluation or deliver the release but open a work ticket for analysis. You might decide that you are willing to accept an automated release if the change in performance falls below one threshold (e.g., 2%) and that the release should be stalled for evaluation above that threshold.

Report to Engineers

If the individuals performing the tests are not part of the Agile team, then an additional step of communicating with the engineers who wrote the software must be undertaken. We would prefer to have the Agile team who wrote the software also perform the tests, but sometimes the team performing these tests is still functionally aligned.

The goal of sharing is to ensure that each item or anomaly from the report gets worked to closure. Closure may occur in one of at least two ways. The first case is to identify the anomaly as an expected outcome of the changes. In this case, the engineer responsible for the explanation should be able to support why the performance deviation is not only expected but actually warranted (as in the case where the increase in revenue will offset the resulting increase in cost). The second case is for a bug to be filed so that the engineering team can investigate the issue further and ideally fix it. It is entirely possible that more tests (with the help of engineering) may need to be run to make a solid business case for no action being taken or to fix a possible bug. In the case of continuous delivery workflows, all reporting should be automated.

Repeat the Tests and Analysis

The last step in the performance process is to repeat the testing and reanalyze the data. This can be needed either because a fix was provided for a bug that was logged in step 6 or because there is additional time and the code base is likely always

changing due to functional bug fixes. If sufficient time and resources are available, these tests should definitely be repeated to ensure the results have not changed dramatically from one build to another for the candidate release and to continue probing for potential anomalies.

Summary of Performance Testing Steps

When conducting performance testing, the following steps are the critical steps to completing it properly. You can add steps as necessary to fit your organization's needs, but these are the ones you must have to ensure you achieve the results that you expect:

1. *Establish success criteria.* Establish which criteria are expected from the application, component, device, or system that is being tested.
2. *Establish the appropriate environment.* Make sure your testing environment is as close to production as possible to ensure that your test results are accurate.
3. *Define the tests.* There are many different categories of tests that you should consider for inclusion in the performance testing, including endurance, load, most used, most visible, and component tests.
4. *Execute the tests.* This step is where the tests are actually being executed in the environment established in step 2.
5. *Analyze the data.* Analyzing the data can take many forms—some as simple as comparing to previous releases, others including stochastic models.
6. *Report to engineers.* If the individuals performing the tests are not part of the Agile team, then an additional step of communicating with the engineers who wrote the software must be undertaken. Provide the analysis to the engineers and facilitate a discussion about the relevant points.
7. *Repeat the tests and analysis.* Continue testing and analyzing the data as necessary to validate bug fixes. If time and resources permit, testing should also continue.

Follow these seven steps and any others that you need to add for your specific situations and organization. The key to a successful process is making it fit the organization.

Performance testing covers a broad range of testing evaluations, but they share a focus on the necessary characteristics of the system rather than on the individual materials, hardware, or code. Concentrating on ensuring the software meets or exceeds the specified requirements or service level agreements is what performance testing is all about.

Don't Stress over Stress Testing

Stress testing is a process that is used to determine an application's stability when subjected to above-normal loads. By comparison, in load testing, the load is only as much as specified or normal operations require. Stress testing goes well beyond these levels, often to the breaking point of the application, to observe the behaviors.

Although several different methods of stress testing exist, the two most commonly used are positive testing and negative testing. In *positive testing*, the load is progressively increased to overwhelm the system's resources. In *negative testing*, resources such as memory, threads, or connections are taken away. Besides determining the exact point of demise or (in some instances) the degradation curve of the application, a major purpose of stress testing is to drive the application beyond its capacity to make sure that when it fails, it can recover gracefully. This approach tests the application's recoverability.

An extreme example of negative stress testing is Netflix's Chaos Monkey. Chaos Monkey is a service that runs on Amazon Web Services (AWS). It seeks out auto-scaling groups (ASGs) and terminates virtual machines (instances within AWS) for each of the groups. Netflix has taken it a step further with Chaos Gorilla, another service that shuts down entire Amazon Availability zones to make sure healthy zones can successfully handle system load with no impact to customers. The company does this to understand how such a loss of resources will impact its solution. According to the Netflix blog, "Failures happen and they inevitably happen when least desired or expected."² The idea is that by scheduling the Chaos Monkey or Chaos Gorilla to "work" during the normal business day, the team can respond to, analyze, and react to issues that would otherwise catch them by surprise in the middle of the night.

Why do we call this example extreme? The folks at Netflix run it in their production environment! Considered one way, this is really a parallel evolution of continuous delivery practices. To be successful in emulating Netflix, a company first needs to ensure that it has all of the incident and crisis management procedures identified earlier in this book nailed. The good news is that should you be interested, the kind folks at Netflix have released the Chaos Monkey into the wild under the project Simian Army on GitHub. Go check it out!

Identify the Objectives

The first step in stress testing is to identify what you want to achieve with the test. As with all projects, time and resources are limited for this sort of testing. By

2. Chaos Monkey released into the wild. *Netflix Techblog*, July 30, 2012. <http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>.

identifying goals up front, you can narrow the field of tests that you will perform and maximize your return on the time and resources invested.

Stress testing can help identify baselines, ease of recoverability, and system interactions, in addition to the results of negative testing. Broadly speaking, ease of recoverability and baselines are considered positive testing. Stress testing to establish a baseline helps to identify the peak utilization possible or degradation curve of a product. Recoverability testing helps to understand how a system fails and recovers from that failure. Testing systems' interactions attempts to ensure that some given functionality continues to work when one or more other services are overloaded.

Identify the Key Services

Next we need to create an inventory of the services to be tested. As we won't be able to test everything, we need a way to prioritize our testing. Some factors that you should consider are criticality to the overall system, service issues most likely to affect performance, and service problems identified through load testing as bottlenecks. Let's talk about each one individually.

The first factor to use in determining which services should be selected for stress testing is the criticality of each service to the overall system performance. If there is a central service such as a database abstraction layer (DAL) or user authorization, it should be included as a candidate for stress testing because the stability of the entire application depends on this service. If you have architected your application into fault-tolerant "swim lanes" (discussed in depth in Chapter 21, Creating Fault-Isolative Architectural Structures), you are likely to still have core services that have been replicated across the lanes.

The second consideration for determining services to stress test is the likelihood that a service will affect performance. This decision will be influenced by knowledgeable engineers but should also be somewhat scientific. You can rank services by the usage of processes such as synchronous calls, I/O, caching, locking, and so on. The more of these higher-risk processes that are included in the service, the more likely they are to have an effect on performance.

The third factor for selecting services to be stress tested is identification of services during load testing as a bottleneck. With any luck, if a service has been identified as a bottleneck, this constraint will have already been fixed—but you should recheck it during stress testing.

Collectively, these three factors should provide you with strong guidelines for selecting the services on which you should focus your time and resources to ensure you get the most out of your stress testing.

Determine the Load

The third step in stress testing is to determine how much load is actually necessary. Determining the load is important for a variety of reasons. First, it is helpful to know at approximately what load the application will start exhibiting strange behaviors so that you don't waste time on much lower loads. Second, you need to understand whether your test systems have enough capacity to generate the required load. The load that you decide to place upon a particular service should stress it sufficiently beyond the breaking point, thereby enabling you to observe the behavior and consequences of the stress. One way to accomplish this is to identify the load under which the service begins to exhibit poor behavior, and then to incrementally increase the load beyond this point.

The important thing is to be methodical, record as much data as possible, and create a significant failure of the service. Stress can be placed upon the service in a variety of manners, such as by increasing the number of requests, shortening any delays, or reducing the hardware capacity. An important factor to remember is that loads, whether identified in production or in load testing, should always be scaled to the appropriate level based on the differences in hardware between the environments.

Establish the Appropriate Environment

As with performance testing, establishing the appropriate environment is critical to effective stress testing. The environment must be stable, consistent, and as close to production as possible. This last criterion might be hard to meet unless you have an unlimited budget. If you are one of the less fortunate technology managers who is constrained by a budget like the rest of us, you will have to scale this expectation down. For example, large pools of servers in production can be scaled down to small pools of two or three servers, but the important consideration is that there are multiple servers load balanced using the same rules. The class of servers should be the same if at all possible, or else a scale factor must be introduced. A production environment with solid-state disks and a test environment with hybrid flash and 15,000-rpm disks, for example, will likely cause the product to exhibit different performance characteristics and different load capacities in the two environments.

It is important to spend some time pondering the appropriate stress testing environment, just as you did for the performance testing environment. Understand the tradeoffs that you are making with each difference between your production and testing environments. Balance the risks and rewards to make the best decisions in terms of what the environment should look like and how useful the tests will be. Unlike with performance testing, you need not be concerned about how the environment affects continuous delivery with stress testing. Generally stress testing is a point-in-time phenomenon and need not be performed prior to each release.

Identify the Monitors

The fifth step in the stress testing process is to identify what needs to be monitored or which data needs to be collected. It is as important to identify what needs to be monitored and captured as it is to properly choose the service, load, and tests. You certainly do not want to go to the trouble of performing the tests, only to discover that you did not capture the data that you needed to perform a proper analysis.

Some items that might be important to consider as potential data points are the results or behavior of the service, response time, CPU load, memory usage, disk usage, thread deadlocks, SQL count, transactions failed, and so on. The results of the service are important in the event that the application provides erroneous results. Comparison of the expected and actual results should be considered as a very good measure of the behavior of the service under load.

Create the Load

The next step in the stress testing process is to create the simulated load. This sixth step is important because it often takes more work than running the actual tests. Creating sufficient load to stress the service may be very difficult if your services have been well architected to handle especially high loads. The best loads are those that are replicated from real user traffic. Sometimes, it is possible to gather this from application or load balancer logs. If this is possible and the source of your load data, then you will likely need to coordinate other parts of the system, such as the database, to ensure they match the load data. For example, if you are testing a signup service and plan to replay actual user registrations from your production logs, you will need to not only extract the registration requests from your logs, but also have the data in the test database set to a point before the user registrations began. The reason for this is that if the user is already registered in the database, a different code path will be executed than is normally the case for a user registration. This difference will significantly skew your testing results and yield inaccurate results. If you cannot get real user traffic to simulate your load, you can revert to writing scripts that simulate a series of steps that exercise the service in a manner as close to normal user traffic as possible.

Execute the Tests

After you have finalized your test objectives, identified the key services to be tested, determined the load necessary, set up your environment, identified what needs to be monitored, and created the simulated load that will be used, you are ready for the seventh step—actually executing the tests. In this step, you methodically progress through your identified services performing the stress tests under the loads determined and meticulously record the data that you identified as being important to perform a proper analysis. As with performance testing, you should keep data from

release to release. Comparing the results from various releases is a great way to quickly understand the changes that have taken place from one release to another.

Analyze the Data

The last step in stress testing is to perform the analysis on the data gathered during the tests. The analysis for the stress test data is similar to that done for the performance tests, in that a variety of methods can be implemented depending on factors such as the amount of time allocated, the skills of the analyst, the acceptable amount of risk, and the level of details expected.

The other significant determinant in how the data should be analyzed is the objectives or goals determined in step 1. If the objective is to establish a baseline, little analysis needs to be done—perhaps just enough to validate that the data accurately depicts the baseline, that it is statistically significant, and that it has only common cause variation. If the objective is to identify the failure behavior, the analysis should focus on comparing the results from the case where the load was below the breaking point and the case where the load was above it. This will help identify warning signs of an impending problem as well as the emergence of a problem or inappropriate behavior of the system at certain loads. If the objective is to test for the behavior when the resource is removed completely from the system, the analysis will probably want to include a comparison of response times and other system metrics between various resource-constrained scenarios and post load to ensure that the system has recovered as expected. For the interactivity objective, the data from many different services may have to examined together. This type of examination might include multivariate analyses such as principal component analysis or factor analysis. The objective identified in the very first step will be the guidepost for this analysis.

A successful analysis will meet the objectives set forth for the tests. If a gap in the data or missing test scenario prevents you from completing the analysis, you should reexamine your steps and ensure you have accurately followed the eight-step process outlined earlier.

Summary of Stress Testing Steps

When performing stress testing, the following steps are the critical ones to completing it properly. As with performance testing, you can add additional steps as necessary to fit your organization's needs.

1. *Identify the objectives.* Identify why you are performing stress testing. These goals usually fall into one of four categories: establish a baseline, identify behavior during failure and recovery, identify behavior during loss of resources, and determine how the failure of one service will affect the entire system.

2. *Identify the key services.* Time and resources are limited, so you must select only the most important services to test.
3. *Determine the load.* Calculate or estimate the amount of load that will be required to stress the application to the breaking point.
4. *Establish the appropriate environment.* The environment should mimic production as much as possible to ensure the validity of the tests.
5. *Identify the monitors.* You don't want to execute tests and then realize you are missing data. Plan ahead by using the objectives identified in step 1 as criteria for what must be monitored.
6. *Create the load.* Create the actual load data, preferably from user data.
7. *Execute the tests.* In this step, the tests are actually executed in the environment established previously.
8. *Analyze the data.* The last step is to analyze the data.

Follow these eight steps and any others that you need to add for your specific situations and organization. Ensure the process fits the needs of the organization.

We need to take a break in our description and praise of the stress testing process to discuss the downside of such testing. Although we encourage the use of stress testing, it is admittedly one of the hardest types of testing to perform properly—and if you don't perform it properly, the effort is almost always wasted. As we discussed in step 4 about setting up the proper environment, if you switch classes of storage or processor speeds, these changes can completely destroy the validity of the test results. Unfortunately, establishing the appropriate environment is a relatively easy step to get correct, especially when compared to the sixth step, creating the load. Load creation is by far the hardest task and the most likely place that you or your team will mess up the process and cause erroneous or inaccurate results. It is very, very difficult to accurately capture and replay real user behavior. As discussed earlier, doing so often necessitates synchronization of data within caches and stores, such as databases or files, because inconsistencies will exercise different code paths and render inaccurate results. Additionally, creating a very large load itself can often be problematic from a capacity standpoint, especially when you're trying to test the interactivity of multiple services.

Given these challenges, we caution you about using stress testing as your only safety net. As we will discuss in the next chapter on go/no-go decisions and rollback, you must have multiple relief valves in the event that problems arise or disaster strikes. We will also cover this subject more fully in Part III, "Architecting Scalable Solutions," when we discuss how to use swim lanes and other application-splitting methods to improve scalability and stability.

As we stated at the beginning of this section, the purpose of stress testing is to determine an application's stability when subjected to above-normal loads. It is clearly differentiated from load testing, where the load is only as much as specified; in stress testing, we go well beyond this level to the breaking point and watch the failure and the recovery of the service or application. We recommend an eight-step stress testing process starting with defining objectives and ending with analyzing the data. Each step in this process is critical to ensuring a successful test yielding the results that you desire. As with our other processes, we recommend starting with this one intact and adding to it as necessary for your organization's needs.

Performance and Stress Testing for Scalability

As we discussed in Chapter 11, Determining Headroom for Applications, it is critical to scalability that you know where you are in terms of capacity for a particular service within your system. Only then can you calculate how much time and growth you have left to scale. This knowledge is fundamental for planning headroom or infrastructure projects, splitting databases/applications, and making budgets. The way to ensure your calculations remain accurate is to conduct performance testing on all your releases to ensure you are not introducing unexpected load increases. It is not uncommon for an organization to implement a maximum load increase allowed per release. As your capacity planning becomes more sophisticated, you will come to see the load added by new features and functionality as a cost that must be accounted for in the cost–benefit analysis.

Additionally, stress testing is necessary to ensure that the expected breakpoint or degradation curve is still at the same point as previously identified. It is possible to leave the normal usage load unchanged but decrease the total load capacity through new code paths or changes in logic. For instance, an increase in a data structure lookup of 90 milliseconds would likely go unnoticed if included in the total response time for a user's request. If this service is tied synchronously to other services, however, as the load builds, hundreds or thousands of 90-millisecond delays will add up and decrease the peak capacity that services can handle.

When we talk about change management, as defined in Chapter 10, Controlling Change in Production Environments, we are really discussing more than the light-weight change identification process for small startup companies. That is, we are referring to the fuller-featured process by which a company attempts to actively manage the changes that occur in its production environment. We have previously defined change management as consisting of the following components: change proposal, change approval, change scheduling, change implementation and logging, change validation, and change efficacy review. Performance testing and stress testing augment this change management process by providing a practice implementation

and—most importantly—a validation of the change. You would never expect to make a change without verifying that it actually affected the system the way that you think it should, such as by fixing a bug or providing a new piece of functionality. As part of performance and stress testing, we validate the expected results in a controlled environment prior to production. This additional step helps ensure that when a change is made in production, it will also work as it did during testing under varying loads.

The most significant factor that we should consider when relating performance testing and stress testing to scalability is the management of risk. As outlined in Chapter 16, Determining Risk, risk management is one of the most important processes when it comes to ensuring your systems will scale. The precursor to risk management is risk analysis, which attempts to calculate the amount of risk associated with various actions or components. Performance testing and stress testing are two methods that can significantly decrease the risk associated with a particular service change. For example, if we were using a failure mode and effects analysis tool and identified a failure mode of a particular feature as being an increase in query time, the mitigation recommended could be to test this feature under actual load conditions, as with a performance test, to determine the actual behavior. This could also be done with extreme load conditions, as with a stress test, to observe behavior above normal conditions. Both of these tests would provide much more information with regard to the actual performance of the feature and, therefore, would lower the amount of risk. Clearly, these two testing processes are powerful tools when it comes to reducing, and thereby managing, the amount of risk within the release or the overall system.

From these three areas—headroom, change control, and risk management—we can see the inherent relationship between successful scalability of a system and the adoption of the performance and stress testing processes. As we cautioned previously in the discussion of stress testing, the creation of the test load is not easy, and if done poorly can lead to erroneous data. However, this challenge does not mean that it is not worth pursuing the understanding, implementation, and (ultimately) mastery of these processes.

Conclusion

In this chapter, we discussed in detail the performance testing and stress testing processes, both of which have important implications for scalability of a system. For the performance testing process, we defined a seven-step process. The key to completing this process successfully is to be methodical and scientific about the testing.

For the stress testing process, we defined an eight-step process. These were the basic steps we felt necessary to have a successful process. You can add other steps as necessary to ensure a proper fit of this process within your organization.

We concluded this chapter with a discussion of how performance testing and stress testing fit with scalability. Based on the relationship between these testing processes and three factors—headroom, change control, and risk management—these processes also are directly responsible for scalability.

Key Points

- Performance testing covers a broad range of engineering evaluations where the emphasis is on the final measurable performance characteristic.
- The goal of performance testing is to identify, document, and (where possible) eliminate bottlenecks in the system.
- Load testing is a process used in performance testing.
- Load testing is the process of putting load or user demand on a system so as to measure its response and stability.
- The purpose of load testing is to verify that the application can meet a desired performance objective, often one specified in a service level agreement.
- Load and performance testing are not substitutes for proper architecture.
- The seven steps of performance testing are as follows:
 1. Establish the criteria expected from the application.
 2. Establish the proper testing environment.
 3. Define the right tests to perform.
 4. Execute the tests.
 5. Analyze the data.
 6. Report to the engineers, if they are not organized into Agile teams.
 7. Repeat as necessary.
- Stress testing is a process that seeks to determine an application's stability when subjected to above-normal loads.
- Stress testing, as opposed to load testing, goes well beyond the normal traffic—often to the breaking point of the application—and observes the behaviors that occur.
- The eight steps of stress testing are as follows:
 1. Identify the objectives of the test.
 2. Choose the key services for testing.

3. Determine how much load is required.
 4. Establish the proper test environment.
 5. Identify what must be monitored.
 6. Create the actual test load.
 7. Execute the tests.
 8. Analyze the data.
- Performance testing and stress testing impact scalability through the areas of headroom, change control, and risk management.

Chapter 18

Barrier Conditions and Rollback

He will conquer who has learned the artifice of deviation. Such is the art of maneuvering.

—Sun Tzu

Whether your development methodology is Agile, waterfall, or some hybrid, implementing the right processes for deploying updates into your production environment can protect you from significant failures. Conversely, poor processes are likely to cause painful and persistent problems. Checkpoints and barrier conditions within your product development life cycle can increase quality and reduce the cost of developing your product. However, even the best teams, armed with the best processes and great technology, may make mistakes and incorrectly analyze the results of certain tests or reviews. If your platform implements a service, you need to be able to quickly roll back significant releases to keep scale-related events from creating availability incidents.

Developing effective go/no-go processes or *barrier conditions* and coupling them with a process and capability to roll back production changes are necessary components within any highly available and scalable service. The companies focused most intensely on cost-effectively scaling their systems while guaranteeing high availability create several checkpoints in their development processes. These checkpoints represent an attempt to guarantee the lowest probability of a scalability-related event and to minimize the impact of any such event should it occur. Companies balancing scalability and high availability also make sure that they can quickly get out of any event created through recent changes by ensuring that they can always roll back from any major change.

Barrier Conditions

You might read this heading and immediately assume that we are proposing that waterfall development cycles are the key to success within highly scalable environments. Very often, barrier conditions or entry and exit criteria are associated with

the phases of waterfall development—and sometimes identified as a reason for the inflexibility of a waterfall development model. Our intent here is not to promote the waterfall methodology, but rather to discuss the need for standards and protective measures regardless of which approach you take for development.

For the purposes of this discussion, assume that a barrier condition is a standard against which you measure success or failure within your development life cycle. Ideally, you want to have these conditions or checkpoints established within your cycle to help you decide whether you are, indeed, on the right path for the product or enhancements that you are developing. Recall our discussion of goals in Chapter 4, Leadership 101, and Chapter 5, Management 101, and the need to establish and measure these goals. Barrier conditions are static goals within a development methodology that ensure your product aligns with your vision and need. A barrier condition for scalability might include checking a design against your architectural principles within an Architecture Review Board process before the design is implemented. It might include completing a code review to ensure the code is consistent with the design or performance testing and implementation. In a continuous delivery environment, it might be executing the unit test library before releasing into production.

Example Scalability Barrier Conditions

We often recommend that the following barrier conditions be inserted into the development methodology or life cycle. The purpose of each of these conditions is to limit the probability of occurrence and resulting impact of any scalability issues within the production environment:

1. *Architecture Review Board.* As described in Chapter 13, Joint Architecture Design and Architecture Review Board, the ARB exists to ensure that designs are consistent with architectural principles. Architectural principles, in turn, should ideally address one or more key scalability tenets within your platform. The intent of this barrier is to ensure that time isn't wasted implementing or developing systems that are difficult or impossible to scale to your needs.
2. *Code Reviews.* Modifying what is (we hope) an existing and robust code review process to include steps that ensure architectural principles are followed within the implementation of the system in question is critical to making sure that code can be fixed to address scalability problems. The goal is identify these problems before the code reaches the QA process and must be fixed later, at a likely more expensive stage.
3. *Performance Testing:* As described in Chapter 17, Performance and Stress Testing, performance testing helps you identify potential issues of scale before you introduce the system into a production environment. The goal is to avoid impacting your customers with a scalability-related issue.

4. *Unit Tests.* Ideally, you will have a robust unit test library with code coverage greater than 75%. If you are working in a continuous delivery (CD) environment, this type of resource is essential. A step in the CD process should be to execute that library to ensure the developer's change did not break some other piece of code.
5. *Production Monitoring and Measurement.* Ideally, your system will have been designed to be monitored, as discussed within Chapter 12, Establishing Architectural Principles. Even if it is not, capturing key performance data from the user, application, and system perspectives after release and comparing it to data from previous releases can help you identify potential scalability-related issues early on, before they impact your customers.

Your processes may include additional barrier conditions that you've found useful over time, but we consider these to be the bare minimum needed to manage the risk of releasing systems that negatively impact customers.

Barrier Conditions and Agile Development

In our practice, we have found that many of our clients have a mistaken perception that including or defining standards, constraints, or processes in Agile processes is a violation of the Agile mindset. The very notion that process runs counter to Agile methodologies is flawed from the outset—any Agile method is itself a process. Most often, we find the Agile Manifesto quoted out of context as a reason for eschewing any process or standard.¹ As a review, and from the Agile Manifesto, Agile methodologies value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Organizations often take the “individuals and interactions over processes and tools” edict out of context without reading the line that follows these bullets: “That is, while there is value in the items from the right, we value the items on the left more.”² This statement clarifies that processes add value, but that people and interactions should take precedent over them where we need to make choices. We absolutely agree with this approach. As such, we prefer to inject process into Agile development most often in the form of barrier conditions to test for an appropriate

1. This information is from the Agile Manifesto, www.agilemanifesto.org.

2. Ibid.

level of quality, scalability, and availability, or to help ensure that engineers are properly evaluated and given guidance. Let's examine how some key barrier conditions enhance our Agile method.

We'll begin with the notion of valuing working software over comprehensive documentation. None of the suggestions we've made—from ARB and code reviews to performance testing and production measurement—violate this rule. The barrier conditions represented by the ARB and joint architecture design (JAD) processes are used within Agile methods to ensure that the product under development can scale appropriately. The ARB and JAD processes can be performed verbally in a group and with limited documentation and, therefore, are consistent with the Agile method.

The inclusion of barrier conditions and standards to help ensure that systems and products work properly in production actually supports the development of working software. We have not defined comprehensive documentation as necessary in any of our proposed activities, although it is likely that the results of these activities will be logged somewhere. Remember, we are interested in improving our processes over time. Logging performance results, for instance, will help us determine how often we are making mistakes in our development process that result in failed performance tests in QA or scalability issues within production. This is similar to the way Scrum teams measure velocity to get better at estimations.

The processes we've suggested also do not in any way hinder customer collaboration or support contract negotiation over customer collaboration. In fact, one might argue that they foster a better working environment with the end customer by inserting scalability barrier conditions to better serve the customer's needs. Collaborating to develop tests and measurements that will help ensure that your product meets customer needs and then inserting those tests and measurements into your development process is a great way to take care of your customers and create shareholder value.

Finally, the inclusion of the barrier conditions we've suggested helps us respond to change by helping us identify when a change is occurring. The failure of a barrier condition is an early alert to issues that we need to address immediately. Identifying that a component is incapable of being scaled horizontally (“scale out, not up” from our recommended architectural principles) in an ARB session is a good indication of potential problems that might be encountered by customers. Although we may make the executive decision to launch the feature, product, or service, we had better ensure that future Agile cycles are used to fix the issue we've identified. However, if the need for scale is so dramatic that a failure to scale out will keep us from being successful, should we not respond immediately to that issue and fix it? Without such a process and series of checks, how would we ensure that we are meeting our customers' needs?

We hope this discussion has convinced you that the addition of criteria against which you can evaluate the success of your scalability objectives is a good idea within your Agile implementation. If not, please remember the “board of directors” test introduced in Chapter 5, Management 101: Would you feel comfortable stating that you absolutely would not develop processes within your development life cycle to ensure that your products and services could scale? Imagine yourself saying to the board, “In no way, shape, or form will we ever implement barrier conditions or criteria to ensure that we don’t release products with scalability problems!” How long do you think you would have a job?

Cowboy Coding

Development without any process, without any plans, and without measurements to ensure that the results meet the needs of the business is what we often refer to as *cowboy coding*. The complete lack of process in cowboy-like environments is a significant barrier to success for any scalability initiatives.

Often, we find that teams attempt to claim that cowboy implementations are “Agile.” This simply isn’t true. The Agile methodology is a defined life cycle that is tailored to adapt to your needs over time, whereas other non-Agile models tend to be more predictive. The absence of processes, such as any cowboy implementation, is neither adaptive nor predictive. Agile methodologies are not arguments against measurement or management, but rather are tuned to release small components or subsets of functionality quickly. They were developed to help control chaos by addressing small, easily managed components; attempting to predict and control very large complex projects, in contrast, is highly vulnerable to failure.

Do not allow yourself or your team to fall prey to the misconception that Agile methodologies should not be measured or managed. Using a metric such as velocity to improve the estimation ability of engineers, but not wielding it as stick to beat them up over their inaccuracy, is a fundamental part of the Agile methodology. A failure to measure dooms you to never improving, and a failure to manage dooms you to getting lost en route to your goals and vision. Being a cowboy when it comes to designing highly scalable solutions is a sure way to get thrown off of the bucking scalability bronco!

Barrier Conditions and Waterfall Development

The inclusion of barrier conditions within waterfall models is not a new concept. Most waterfall implementations include a concept of entry criteria and exit criteria for each phase of development. For instance, in a strict waterfall model, design may not start until the requirements phase is completed. The exit criteria for the

requirements phase, in turn, may include a signoff by key stakeholders, a review of requirements by the internal customer (or an external representative), and a review by the organizations responsible for producing those requirements. In modified, overlapping, or hybrid waterfall models, requirements may need to be complete for the systems to be developed first but may not be complete for the entire product or system. If prototyping is employed, potentially those requirements may need to be mocked up in a prototype before major design starts.

For our purposes, we need inject only the four processes we identified earlier into the existing barrier conditions. The Architecture Review Board lines up nicely as an exit criterion for the design phase of our project. Code reviews, including a review consistent with our architectural principles, might create exit criteria for our coding or implementation phase. Performance testing that specifies a maximum percentage change allowed for critical systems should be performed during the validation or testing phase. Production measurements being defined and implemented should be the entry criteria for the maintenance phase. Significant increases in any measured area, if not expected, should trigger new work items to reduce the impact of the implementation or changes in architecture to allow for more cost-effective scalability.

Barrier Conditions and Hybrid Models

Many companies have developed models that merge Agile and waterfall methodologies, and some continue to follow the predecessor to Agile methods known as rapid application development (RAD). For instance, some companies may be required to develop software consistent with contracts and predefined requirements, such as those that interact with governmental organizations. These companies may wish to have some of the predictability of dates associated with a waterfall model, but desire to implement chunks of functionality quickly as in Agile approaches.

The question for these models is where to place the barrier conditions for the greatest benefit. To answer that question, we need to return to the objectives when using barrier conditions. Our intent with any barrier condition is to ensure that we catch problems or issues early in our development so that we reduce the amount of rework needed to meet our objectives. It costs us less in time and work, for instance, to catch a problem in our QA organization than it does in our production environment. Similarly, it costs us less to catch an issue in the ARB review than if we allow that problem to be implemented and subsequently catch it only in a code review.

The answer to the question of where to place the barrier conditions, then, is simple: Put the barrier conditions where they add the most value and incur the least cost to our processes. Code reviews should be placed at the completion of each coding cycle or at the completion of chunks of functionality. The architectural review

should occur prior to the beginning of implementation, production metrics obviously need to occur within the production environment, and performance testing should happen prior to the release of a system into the production environment.

Rollback Capabilities

You might argue that the inclusion of an effective set of barrier conditions in the development process should obviate the need to roll back major changes within the production environment. We can't really argue with that thought or approach—technically, it is correct. However, arguing against having the capability to perform a rollback is really an argument against having an insurance policy. You might believe, for instance, that you don't need health insurance because you are a healthy individual. But what happens when you develop a treatable cancer and don't have sufficient funds to cover the treatment? If you are like most people, your view of whether you need this insurance changes immediately when it would become useful. The same holds true when you find yourself in a situation where fixing forward is going to take quite a bit of time and have quite an adverse impact on your clients.

Rollback Window

Rollback windows—that is, how much time must pass after a release before you are confident that you will not need to roll back the change—differ significantly by business. The question to ask yourself when determining how to establish your specific rollback window is how you will know when you have enough information about performance to determine if you need to undo your recent changes. For many companies, the bare minimum is that a weekly business day peak utilization period is needed to have great confidence in the results of their analysis. This bare minimum may be enough for modifications to existing functionality, but when new functionality is added, it may not be enough.

New functions or features often have adoption curves that take more than one day to shuffle enough traffic through that feature to determine its true impact on system performance. The amount of data gathered over time within any new feature may also have an adverse performance impact and, as a result, may negatively impact your scalability.

Another consideration when determining your rollback window deals with the frequency of your releases and the number of releases you need to be capable of rolling back. Perhaps you have a release process that involves releasing new functionality to your site several times a day. In this case, you may need to roll back more than one release if the adoption rate of any new functionality extends into the next release

cycle. If this is the case, your process needs to be slightly more robust, as you are concerned about multiple changes and multiple releases, rather than just the changes from one release to the next.

Rollback Window Checklist

To determine the time frame necessary to perform a rollback, you should consider the following things:

- How long is the period between your release and the first heavy traffic period for your product?
- Is this a modification of existing functionality or a new feature?
- If this is a new feature, what is the adoption curve for it?
- How many releases do you need to consider rolling back based on your release frequency? We call this the rollback version number requirement.

Your rollback window should allow you to roll back after significant adoption of a new feature (e.g., up to 50% adoption) and after or during your first time period of peak utilization.

Rollback Technology Considerations

We often hear during our discussions around the *rollback insurance policy* that clients in general agree that being able to roll back would be great, but doing so is technically not feasible for them. Our response is that rollback is almost always possible; it just may not be possible with your current team, processes, or architecture.

The most commonly cited reason for an inability to perform a rollback in Web-enabled platforms and back-office IT systems is database schema incompatibility. The argument usually goes as follows: For any major development effort, there may be significant changes to the schema resulting in an incompatibility with the way old and new data are stored. This modification may result in table relationships changing, candidate keys changing, table columns changing, tables being added, tables being merged, tables being disaggregated, and tables being removed.

The key to fixing these database issues is to grow your schema over time and to keep old database relationships and entities for at least as long as you would need to roll back to them should you run into significant performance issues. In the case where you need to move data to create schemas of varying normal forms, either for functionality reasons or performance reasons, consider using either data movement programs that are potentially launched by a database trigger or a data movement daemon or

third-party replication technology. This data movement can cease whenever you have met or exceeded the *rollback version number* limit identified in your requirements. Ideally, you can turn off such data movement systems within a week or two after implementation and validation that you do not need to roll back.

Ideally, you will limit such data movement, and instead populate new tables or columns with new data, while leaving old data in its original columns and tables. In many cases, this approach is sufficient to meet your needs. In the case where you are reorganizing data, simply move the data from the new positions to the old ones for the period of time necessary to perform the rollback. If you need to change the name of a column or its meaning within an application, you must first make the change in the application, leaving the database alone; in a future release, you can then change the database. This is an example of the general rollback principle of making the change in the application in the earlier release and making the change in the database in a later release.

Cost Considerations of Rollback

If you've gotten to this point and determined that designing and implementing a rollback insurance policy has a cost, you are absolutely right! For some releases, the cost can be significant, adding as much as 10% or 20% to the cost of the release. In most cases and for most releases, we believe that you can implement an effective rollback strategy for less than 1% of the cost or time of the release. In many cases, you are really just talking about different ways to store data within a database or other storage system. Insurance isn't free, but it exists for a reason.

Many of our clients have implemented procedures that allow them to violate the rollback architectural principle as long as several other risk mitigation steps or processes are in place. We typically suggest that the CEO or general manager of the product or service in question sign off on the risk and review the risk mitigation plan (see Chapter 16, Determining Risk) before agreeing to violating the rollback architectural principle. In the ideal scenario, the principle will be violated only with very small, very low-risk releases where the cost of being able to roll back exceeds the value of the rollback given the size and impact of the release. Unfortunately, what typically happens is that the rollback principle is violated for very large and complex releases to satisfy time-to-market constraints. The problem with this approach is that these large complex releases are often precisely the ones for which you need the rollback capability the most.

Challenge your team whenever team members indicate that the cost or difficulty to implement a rollback strategy for a particular release is too high. Often, simple solutions, such as implementing short-lived data movement scripts, may be available to help mitigate the cost and increase the possibility of implementing the rollback strategy. Sometimes, implementing markdown logic for complex features rather

than seeking to ensure that the release can be rolled back can significantly mitigate the risk of a release. In our consulting practice at AKF Partners, we have seen many team members who start by saying, “We cannot possibly roll back.” After they accept the fact that it is possible, they are then able to come up with creative solutions for almost any challenge.

Markdown Functionality: Design to Be Disabled

Another of our architectural principles from Chapter 12, Establishing Architectural Principles, is designing a feature to be disabled. This concept differs from rolling back features in at least two ways.

First, if this approach is implemented properly, it is typically faster to turn a feature off than it is to replace it with the previous version or release of the system. When done well, the application may listen to a dedicated communication channel for instructions to disallow or disable certain features. Other approaches may require the restart of the application to pick up new configuration files. Either way, it is typically much faster to disable functions causing scalability problems than it is to replace the system with the previous release.

A second way that functionality disabling differs from rolling back is that it might allow all of the other functions within any given release, both modified and new, to continue to function as normal. For example, if in a dating site we had released both a “Has he dated a friend of mine?” search and another feature that allows the rating of any given date, we would need to disable only our search feature until a problem with these features is fixed, rather than rolling back and in effect turning off both features. This obviously gives us an advantage in releases containing multiple fixes directed toward modified and new functionality.

Designing all features to be disabled, however, can sometimes add an even more significant cost than designing to roll any given release back. The ideal case is that the cost is low for both designing to be disabled and rolling back, and the company chooses to do both for all new and modified features. Most likely, you will identify features that are high risk, using the failure mode and effects analysis process described in Chapter 16, to determine which should have markdown functionality enabled. Code reuse or a shared service that is called asynchronously may significantly reduce the cost of implementing functions that can be disabled on demand. Implementing both rollback and feature disabling helps enable Agile methods by creating an adaptive and flexible production environment rather than one relying on predictive methods such as extensive, costly, and often low-return performance testing.

If implemented properly, designing to be disabled and designing for rollbacks can actually improve your time to market by allowing you to take some risks in

production that you would not take in their absence. Although not a replacement for load and performance testing, these strategies allow you to perform such testing much more quickly, confident in the knowledge that you can easily move back from implementations once released.

The Barrier Condition, Rollback, and Markdown Checklist

Do you have the following?

- Something to block bad scalability designs from proceeding to implementation?
- Reviews to ensure that code is consistent with a scalable design or principles?
- A way to test the impact of an implementation before it goes to production?
- Ways to measure the impact of production releases immediately?
- A way to roll back a major release that impacts your ability to scale?
- A way to disable functionality that impacts your ability to scale?

Answering “Yes” to all of these questions puts you on a path to identifying scale issues early and being able to recover from them quickly when they happen.

Conclusion

This chapter covered topics such as barrier conditions, rollback capabilities, and markdown capabilities, all of which are intended to help companies manage the risks associated with scalability incidents and recover quickly from these events if and when they occur. Barrier conditions (i.e., go/no-go processes) focus on identifying and eliminating risks to future scalability early within a development process, thereby lowering the cost of identifying the issue and eliminating the threat of it in production. Rollback capabilities allow for the immediate removal of any scalability-related threat, thereby limiting its impact on customers and shareholders. Markdown and disabling capabilities allow features impacting scalability to be disabled on a per-feature basis, removing them as threats when they cause problems. Many other mechanisms for facilitating rollback are also available, including changing DNS records or alternating pools of virtual machines with different versions of code.

Ideally, you will consider implementing all of these measures. Sometimes, on a per-release basis, the cost of implementing either rollback or markdown capabilities is exceptionally high. In these cases, we recommend a thorough review of the risks and all of the risk mitigation steps possible to help minimize the impact on your

customers and shareholders. If both markdown and rollback have high costs, consider implementing at least one unless the feature is small and not complex. Should you decide to forego implementing both markdown and rollback, ensure that you perform adequate load and performance testing and that you have all of the necessary resources available during product launch to monitor and recover from any incidents quickly.

Key Points

- Barrier conditions (go/no-go processes) exist to isolate faults early in the development life cycle.
- Barrier conditions can work with any development life cycle. They do not need to be documented extensively, although data should be collected to learn from past mistakes.
- Architecture Review Board reviews, code reviews, performance testing, and production measurements can all be considered examples of barrier conditions if the result of a failure of one of these conditions is to rework the system in question.
- Designing the capability to roll back your application helps limit the scalability impact of any given release. Consider it an insurance policy for your business, shareholders, and customers.
- Designing to disable, or mark down, features complements designing for rollback and adds the flexibility of keeping the most recent release in production while eliminating the impact of offending features or functionality.

Chapter 19

Fast or Right?

Thus, though we have heard of stupid haste in war, cleverness has never been seen associated with long delays.

—Sun Tzu

You have undoubtedly heard that when given the choices of speed (time), cost, and quality, we can only ever choose two. This classic statement is a nod to the tradeoffs inherent in attempting to optimize the time, cost, and quality triad. In maximizing one of these variables—say, time (as in focusing on a fast delivery)—we must either relax our quality or cost boundary. Put another way, we need to either hire more people to deliver faster or cut corners on quality to deliver on time.

In this chapter, we discuss the general tradeoffs made in business, and specifically in the product development life cycle. We also discuss how these tradeoffs relate to scalability and availability. Finally, we provide a framework for thinking through decisions on how to balance these three objectives or constraints, depending on how you view them. This framework will give you a guide by which you can assess situations in the future and make the best decisions possible.

Tradeoffs in Business

The speed, quality, and cost triumvirate is often referred to as the *project triangle*. It provides a good visual representation of how these three attributes are inextricably connected, and how decisions with regard to one attribute impact the others. Several variations on this illustration also include scope as a fourth element. This variation is depicted by a triangle, in which the legs are speed, quality, and cost, and scope is in the middle. We prefer to use the traditional speed/cost/quality project triangle and define scope as the size of the triangle. This conception is represented in Figure 19.1, where the legs are speed, cost, and quality, and the area of the triangle is the scope of the project. If the triangle is small, the scope of the project is small and



Figure 19.1 Project Triangle

thus the cost, time, and quality elements are proportional. The representation is less important than the reminder that a balance among these four factors is necessary to develop products.

Ignoring any one of the legs of the triangle will cause you to deliver a poor product. If you ignore the quality of the product, a feature may lack the desired or required characteristics and functionality or the product may be so buggy as to render it unusable. If you ignore the speed element, your competitors are likely to beat you to market and you will lose the “first mover” advantage and be perceived as a follower rather than an innovator. The larger the scope of the project, the higher the cost, the slower the speed to market, and the more effort required to achieve a quality standard. Any of these scenarios should be worrisome enough for you to seriously consider how you and your organization actively balance these constraints.

To completely understand why these tradeoffs exist and how to manage them, you must first understand their definitions. We will define *cost* as any related expense or capital investment that is utilized by or needed for the project. Costs include such direct charges as the number of engineers working on the project, the number of servers (real or virtual) required to host the new service, and the marketing campaign for the new service. They also include indirect or allocated costs such as an additional database administrator necessary to handle the increased workload caused by another set of databases and the additional bandwidth utilized by customers of the feature. Why include these allocated costs? The answer is that by spending time on performance of the feature, you may find ways to cut hardware costs, network bandwidth requirements, maintenance headcount, and other expenses. Time spent engineering a solution may reduce the cost of supporting or maintaining that solution.

For the definition of *quality*, we focus on the appropriateness of the feature to meet the needs of the marketplace. A feature launched with half of the true market demand for functionality is not likely to generate as much interest (and resulting revenues) from customers as one that fully meets the expectations of potential users. Included in our definition is the notion of defects within a product, as any “buggy” solution will by definition be below the expectations of the target market. Thus, the faster a solution is released, assuming constant cost, the less likely it is to meet market expectations

in terms of capability (feature completeness relative to need) or usability (defects presenting difficult usage or interfaces). Quality can also be affected by how well we test our solutions for appropriateness prior to launch using both automation and manual techniques. Organizations that skimp on tools for testing cannot utilize their testing engineers as efficiently as their more generous counterparts.

For the definition of *speed*, we will use the amount of time that a feature or project takes to move from “ideation” (the identification of a potential need and resulting solution) within the product development life cycle to release in production. We know that the life cycle doesn’t end with the release to production; in fact, it continues through support and eventually depreciation. However, those phases of the feature’s life are typically a result of the decisions made much earlier. For example, rushing a feature through the life cycle without ample time being given to design considerations will significantly increase the cost to support the feature in production.

Scope, then, is just the functional outcome, per Figure 19.1, of time, cost, and quality. An increase in any dimension while holding the others constant increases the area of the triangle. Spending time or money on non-market-facing functionality, such as long-term cost of ownership considerations, increases scope. Modifying underlying hardware or software infrastructure, such as the replacement of databases or persistence technology, modifies scope by increasing time and/or costs while potentially not affecting the overall quality of the product.

We use the project triangle to represent the equality in importance of these constraints. The two diagrams in Figure 19.2 represent different foci for different projects. The project on the left has a clear predilection for faster speed and higher quality at the necessary increase in cost. This project might be something that is critical to block a competitor from gaining market share. Thus, it needs to be launched by the end of the month and be full featured in an attempt to beat a competitor to market with a similar product. The cost of adding more engineers—possibly more senior engineers and more testing engineers—is worth the advantage in the marketplace with your customers.

The project on the right in Figure 19.2 focuses on increased speed to market with a lower cost point, at the expense of reduced quality. This project might be something necessary for compliance where it is essential to meet a deadline to avoid penalties.

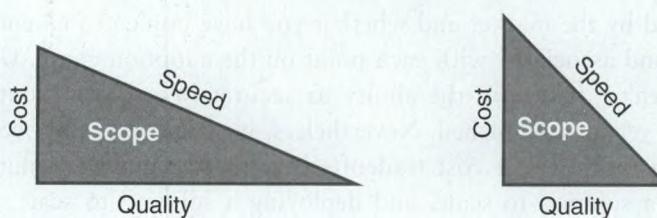


Figure 19.2 Project Triangle Choices

There are likely no revenue-generating benefits for the feature; therefore, it is essential to keep the costs as low as possible. This project might be the modern-day equivalent to a Y2K bug, where the fix does not need to be full functioned but rather simply needs to perform the basic functionality by the specified date with minimal cost.

Success in a technology business is predicated on making tough decisions about time, cost, and quality. Often, these decisions are made via a well-thought-out process that seeks to understand the pros and cons of giving more or less time, money, or people to certain projects. As we will discuss later in this chapter, there are several processes of varying complexity that you can use to help make these decisions.

While each leg of the project triangle is important, the relationships in tradeoffs are not always linear. A perfect example of this is the *Mythical Man Month* example, in which a 5% increase in cost (effort or headcount) is not likely to represent a 5% decrease in time to market. Obviously, scope is increased in this case as effort increases significantly, but without an equal reduction in speed. Understanding that these tradeoffs aren't always proportional helps inform business decisions regarding tradeoffs. Why face significantly higher costs for slightly shorter delivery times? Sometimes such a decision may be wise, but for steady-state businesses it probably does not make sense.

Relation to Scalability

Decisions we make about the non-market-facing capabilities of our solutions are embedded within the notion of scope in the project triangle. We may decide, for instance, to forgo designing and deploying a horizontally scalable solution until we better understand the market adoption forces. Such a decision helps us to reduce the time to market of the solution and the effort (cost) to produce a solution. In doing so, we reduce the scope of the project but take on longer-term liabilities associated with rework of the product to meet longer-term market growth needs. Conversely, if we design and implement a horizontally scalable solution it will increase the time to market and cost of the solution without a beneficial impact to short-term market adoption expectations. Scope is increased overall with such an approach.

The primary issue to be pondered here is how quickly your product or solution will be adopted by the market and whether you have implemented enough scale to meet the demand associated with each point on the adoption curve. Unfortunately, most of us aren't gifted with the ability to accurately forecast adoption of products that have yet to be launched. Nevertheless, we can hedge our bets in this area by recognizing the inherent cost tradeoffs between designing a solution to scale, implementing a solution to scale, and deploying a solution to scale. Here we use the term *design* to refer to the process of describing how a solution should scale through architectural documentation. The cost of such an exercise is high in terms

	Design	Implement	Deploy
Scale Objective	20x to Infinite	3x to 20x	1.5x to 3x
Engineering Cost	Low	High	Medium
Intellectual Cost	High	Medium	Low to Medium
Asset Cost	Low	Low to Medium	High to Very High
Total Cost	Low/Medium	Medium	Medium/High

Figure 19.3 D-I-D Matrix: Design, Implement, Deploy

of intellectual engagement, but low in terms of the effort needed to actually implement or deploy the architecture. As such, the overall cost to describe how we expect a solution to scale is relatively low and is something we should consider for any solution we expect to gain widespread acceptance by a large market. Having this blueprint for implementation in our back pocket will help should our solution be rapidly adopted, as we can quickly schedule the real work (coding) and the equipment purchases or cloud leases (deployment) to be successful.

Figure 19.3 describes these tradeoffs in relative terms as well as the target goals for each. The D-I-D matrix is useful in thinking about the goals for how much headroom you should plan for in each of the design, implement, and deploy phases.

How to Think About the Decision

There are a variety of methods to choose from when you need to determine the appropriate tradeoffs among speed, cost, and quality. You can choose to rely on one of these methods, or you can learn them all so that you use each in the most appropriate manner. Unfortunately, no decision process will ever be able to guarantee that you reach a *correct* decision because often there is no *correct* decision; each approach has different pros and cons.

Our goal here is to arm you with several methodologies so that you can make the *best* decision based on the information that you have today. We recommend one of four methods to our clients.

The first method is the one most closely aligned with Agile principles—namely, to prefer speed and small releases over everything else. At its heart, Agile is a process focused on discovery. It assumes incomplete knowledge about the true requirements of a market and, therefore, espouses end-user interactions as the primary means of discovering the true market need. Smaller and faster is better in Agile. The sooner we can get something into the customers' hands, the sooner we can learn what needs to be augmented, what needs to be fixed, and what works well as is. To be successful with this approach, we must focus on the absolute minimum viable product or feature set necessary for any release. Describing this strategy in terms of the project triangle, the initial triangle is always small. Time is short, cost and effort are small, and, because we don't know the market need, the "quality" leg is similarly small. We then

iteratively expand “quality” by changing our direction to sense the desired functionality for adoption. This has an iterative effect on the cost of the project over time as well as the effort. Our triangle “grows” in size with each release. In some cases, we may replace features in a release that did not gain adoption with new features.

Unfortunately, even Agile methods require tradeoffs. What do we do when we need to cut down stories to size a release for a sprint? Three approaches may be used to facilitate decision making with both Agile methods and other development methodologies. The first approach is essentially the same *gut feel* method described in Chapter 16, Determining Risk. The second method is to make a list of pros and cons for each constraint. The third method is what we call a *decision matrix*. It involves constructing a well-thought-out ranking and analysis of the important short- and long-term factors under consideration and associating them with the speed, cost, and quality tradeoffs. If that last one sounds confusing, don’t worry; we’ll go through it in more detail in a few paragraphs.

Gut feel suffers from a high level of inaccuracies and the inability to scale the solution in large organizations. That said, it has its place in small decisions where the cost of making a decision is proportional to the level of investment in the product itself. Small investments in effort should not be overly burdened by large investments in process.

The second, more formal method of tradeoff analysis is comparison of pros and cons. Using the product triangle, list the pros and cons of each leg modification. As an example of how this works, let’s assume you are trying to making a decision about what to do with a payment feature on your Web site. The following may be some of the tradeoffs:

1. Increase engineers allocation
 - Pros: Faster payment feature development; better feature design
 - Cons: Other features suffer from reallocation; cost allocated to feature increases
2. Speed feature into production
 - Pros: Fulfill business need for no more manual processing
 - Cons: Possibly weaker design; fewer contingencies thought through; increased cost in hardware
3. Reduce quality testing
 - Pros: Meet business timeline
 - Cons: More unidentified and/or unfixed bugs

The next step is to analyze the pros and cons to determine the best approach. One widely used approach is to apply weights (impact) to the pros and cons and

determine a mathematical outcome. A simpler approach is to simply look at the preponderance of information and make an informed decision. Again, effort should be proportional to the value of the decision. Larger decisions likely are worth the extra cost of a methodical weighing of pros and cons.

The third method of tradeoff analysis is a more formal process that builds on the mathematical approach of the pros-and-cons method. In this process, you take the tradeoffs identified and add to them factors that are important in accomplishing the project. At the end of the analysis, you have a score that you can use to judge each tradeoff based on the metrics that you deem most important for your organization. As stated earlier, we cannot guarantee that you will make a correct decision when using this (or any other) approach, because factors that may impact you in the future might not be known at this point. However, this method provides some reassurance that you have made a decision that is based on data and is the best one you can make at this time.

Let us continue with the example of a payments feature on your Web site to see how this analytical approach works. In addition to the pros and cons, we need to specify the factors that are most important to us in accomplishing this feature. Our work results in identification of the following factors:

- Meet the business goal of launching by the end of the month.
- Maintain availability of the entire system at 99.99%.
- The feature should scale to 10x growth.
- The other product releases should not be pushed off track by this effort.
- We want to follow established processes as much as possible.

Now we must rank order these factors to find out which ones are the most important. In Figure 19.4, the tradeoffs are listed down the left column and the factors across the top of the matrix. These factors are sorted and have a *weight* below

Tradeoffs	Weight	Factors					Total
		Meet Business Goal of Launch by EOM	Maintain Availability at 99.99%	Feature Scales to 10x	Keep Other Releases on Track	Follow Established Processes	
Engineers Reallocated	5	9	1	9	-3	-3	67
Speed Feature to Production	9	9	-3	-3	3	-3	27
Reduce Quality Testing	1	1	-3	-1	9	-9	-1

Scale

-9 Highly Unsupportive
-3 Very Unsupportive
-1 Unsupportive
1 Supportive
3 Very Supportive
9 Highly Supportive

Figure 19.4 Decision Matrix

$$\text{Total} = (9 \times 5) + (1 \times 4) + (9 \times 3) + (-3 \times 2) + (-3 \times 1)$$

Figure 19.5 Total Calculation

each factor. For simplicity, we used 1 through 5, as there are five factors. For more elaborate matrices, you can use a variety of scales, such as 1, 3, 9, or allocation out of a 100-value sum, where you have 100 points to allocate among the factors (one may get 25, whereas others may get 3).

After the matrix is created, you need to fill in the middle, which is the strength of support that a tradeoff has on a factor. Here we use a scale from -9 to 9, with increments of 1, 3, -3, and -1. If a tradeoff fully supports a factor, it would receive a score of 9. If it somewhat supports the factor, it gets a 3. If it is unsupportive of the factor, in which case it would cause the opposite of the factor, it gets a negative score; the higher this negative score, the more it is unsupportive. For example, the tradeoff of Reduce Quality Testing for the feature has a -9 score for Follow Established Processes in Figure 19.4, because it clearly does not follow established processes of testing.

After the matrix is filled out, we can perform the calculations on it. The formula is to multiply each score in the body of the matrix by the weight of each factor and then sum these products for each tradeoff, thereby producing the total score. Using the Engineers Reallocated tradeoff, we have a formula as depicted in Figure 19.5.

The total score for this tradeoff in the equation in Figure 19.5 is 67. This formula is calculated for each tradeoff. Armed with the final score, we can analyze each tradeoff individually as well as all the tradeoffs collectively. From this sample analysis, we decide to find a way to allow more time to be spent on quality testing while proceeding with reallocating engineers and expediting the feature into production.

Fast or Right Checklist

- What does your gut tell you about the tradeoff?
- What are the pros and cons of each alternative?
- Is a more formal analysis required because of the risk or magnitude of the decision?
- Is there a benefit from reducing quality and effort to produce a product faster and learn from the market (discovery)?
- If a more formal analysis is required:
 - What are the most important factors? In Six Sigma parlance, these are critical to quality indicators.

- How do these factors rank compared to each other—that is, what is the single most important factor?
- What are the actual tradeoffs being discussed?
- How do these tradeoffs affect the factors?
- Would you feel comfortable standing in front of your board explaining your decision based on the information you have today?

We have given you a reason to prefer speed for the purposes of discovery and three methods of analyzing the tradeoffs from balancing the cost, quality, and speed constraints. It is completely appropriate to use all three of these methods at different times or in increasing order of formality until you believe that you have achieved a sufficiently rigorous decision. Two factors that you might consider when deciding which method to use are the risk of the project and the magnitude of the decision. The risk should be calculated by one of the methods described in Chapter 16. There is not an exact level of risk that corresponds to a particular analysis methodology. Using the traffic light risk assessment method, projects that would be considered green could be analyzed by gut feeling, whereas yellow projects should at least have the pros and cons compared as described in the pros-and-cons comparison process earlier. Examples of these tradeoff rules are shown in Table 19.1. Of course, red projects should be candidates for preparation of a fully rigorous decision matrix. This is another great intersection of processes where a set of rules to work by would be an excellent addition to your documentation.

Conclusion

In this chapter, we tackled the tough and ever-present balancing act between cost, quality, speed, and scope. The project triangle is used to show how each constraint is equally important and worthy of attention. Each project will have a different predilection for satisfying one or more of these constraints. Some projects have a greater

Table 19.1 *Risk and Tradeoff Rules*

Risk: Traffic Light	Risk: FMEA	Tradeoff Analysis Rule
Green	< 100 points	No formal analysis required
Yellow	< 150 points	Compare pros and cons
Red	> 150 points	Fill out decision matrix

need to satisfy the “reduce cost” goal; in others, it is imperative that the quality of the feature be maintained even at the detriment of cost, speed, and scope.

The cost of a feature or project includes both the direct and indirect costs associated with. It can become a fairly exhaustive effort to attempt to allocate all costs with a particular feature, and this exercise is generally not necessary. Rather, it typically suffices to be aware that there are many levels of costs, which occur over both the short and long terms. The definition of quality includes both the amount of bugs in a feature and the amount of full functionality. A feature that does not have all the functions specified is of poorer quality than one that has all the specified features. Speed can be defined as the time to market or the pace at which the feature moves through the product development life cycle into production but not beyond. Post-production support is a special case that is more a cause of cost, quality, and speed tradeoffs, rather than a part of them.

As business leaders and technology managers, we are constantly making tradeoff decisions among these three constraints of cost, quality, and speed. We are well aware of some of these tradeoffs, but others are more subliminal. Some of these decisions occur consciously, whereas others are subconscious analyses that are done in a matter of seconds.

There is a direct relationship when these constraints are applied to infrastructure or scalability projects. In contrast, there is an indirect relationship when decisions made for features affect the overall scalability of the system many months or years later because of predictable—and in some cases unforeseen—factors.

Within Agile environments, speed (time to market) should always trump other concerns, with a focus on limiting the overall scope of a project primarily by limiting quality. In this context, quality addresses market fit. This focus helps speed initiatives to market so that we can collect real customer feedback (in terms of usage) on our products. In so doing, we hope to make better-informed decisions in the future.

Whether you employ Agile methods or not, you will still find yourself needing to make tradeoffs among cost, quality, and speed. To help you make these decisions, we introduced three methods for decision analysis: the gut feel method (first introduced in the discussion of risk), a pros-and-cons comparison, and a rigorous decision matrix that provides formulas to calculate scores for each tradeoff. Although we conceded that there is no *correct* answer possible due to the unknowable factors, there are certainly *best* answers that can be achieved through rigorous analysis and data-driven decisions.

As we consider the actual decisions made on the tradeoffs to balance cost, quality, speed, and scope, as well as the method of analysis used to arrive at those decisions, the fit within your organization at this particular time is the most important aspect in selecting a specific decision-making process. As your organization grows and matures, it may need to modify or augment these processes, make them more

formal, document them further, or add steps that customize them more for the company. For any process to be effective, it must be used, and for it to be used, it needs to be a good fit for your team.

Key Points

- There is a classic balance among cost, quality, and speed in almost all business decisions.
- Technology decisions, especially in the product development life cycle, must balance these three constraints daily.
- Agile methods tend to have a bias toward speed for the purposes of developing market insights (discovery). They do so by reducing scope/quality (quality is defined as market fit here) to gain insights at an earlier stage in the product development life cycle.
- Each project or feature can make different allocations across cost, quality, and speed.
- Cost, quality, and speed are known as the project triangle because a triangle represents the equal importance of all three constraints.
- There are short- and long-term ramifications of the decisions and tradeoffs made during feature development.
- Tradeoffs made on individual features can affect the overall scalability of the entire system.
- Technologists and managers must understand and be able to make the right decisions in the classic tradeoff among speed, quality, and cost.
- There are at least three methods of performing a tradeoff analysis: gut feel, pros-and-cons comparison, and decision matrix.
- The risk of the project should influence which method of tradeoff analysis is performed.
- A set of rules to govern which analysis method should be used in which circumstances would be extremely useful for your organization.

Part III

Architecting Scalable Solutions

Chapter 20: Designing for Any Technology	317
Chapter 21: Creating Fault-Isolative Architectural Structures	327
Chapter 22: Introduction to the AKF Scale Cube	343
Chapter 23: Splitting Applications for Scale	357
Chapter 24: Splitting Databases for Scale	375
Chapter 25: Caching for Performance and Scale	395
Chapter 26: Asynchronous Design for Scale	411

Chapter 20

Designing for Any Technology

Success in warfare is gained by carefully accommodating ourselves to the enemy's purpose.

—Sun Tzu

Have you ever heard someone describe an architecture or design by using the names of the third-party or open source solutions used to implement that platform or architecture? It might sound something like this:

We use Apache Web servers and Tomcat application servers. We use MySQL databases and NetApp storage devices. Our network gear is Cisco routers and switches.

Nice product plugs. If this were a movie, you could sell the product placements. Each of the solution providers mentioned will certainly appreciate this description of an implementation. And yes, we meant to say “implementation,” because the preceding statement is neither a design nor an architecture.

This chapter describes the difference between architecture and implementation. We further make the argument that the best architectures are not accomplished through vendor or open source implementations, but rather through vendor and technology-agnostic designs.

An Implementation Is Not an Architecture

Sometimes it is easiest, albeit indirect, to describe what something *is* by defining that something by what it *is not*. For instance, in attempting to teach a child what a dog is, you might be required from time to time to explain that a cat is not a dog and that a dog is not a cat. This approach is especially useful when two things are often confused in popular speech and literature. An example of such a popular confusion exists within the definition of an implementation of an architecture and the actual system’s architecture.

The best architects of buildings and houses do not describe trusses, beams, and supports using the vendor's name, but rather by their size, load capacity, composition, and so on. This is because the architect realizes that in most cases the items in question are commodities and that the vendor solution will likely be selected based on price, reputation, and quality. Similarly, electrical engineers do not typically reference vendor names when describing a design; instead, they refer to a resistor by its level of resistance (in ohms) and its tolerance (variation to actual resistance). The house architect and the electrical engineer understand that describing something with a vendor's name is an implementation, whereas describing something through specifications and requirements is a design or an architecture.

Implementations represent the choices you have made due to cost considerations, build versus buy decisions, returns on investment, skill sets within your team, and so on. The use of C++ or Java or PHP as a coding language is not indicative of your architecture; rather, these are choices of tools and materials to implement components of your architecture. The choice of a Microsoft database over Sybase, MySQL, or Oracle as a database is not an architecture, but rather an implementation of a database component of your architecture. The decision to go open source versus using a vendor-provided solution is another example of an implementation decision, as is the decision to use a Microsoft operating system rather than some variant of Linux. Referring back to the discussion on architects and electrical engineers, implementation decisions are made by technicians (builders and electricians) and are made to be consistent with the architectural design.

Technology-Agnostic Design

The architecture of a platform describes how something works in generic terms with specific requirements. The implementation describes the specific technologies or vendor components employed. Physical architectures tend to describe the components performing the work, whereas logical architectures tend to define the activities and functions necessary to do the work. We like to discuss architectures from a system's perspective, where the logical architecture is mapped to its physical components such that both can be evaluated in the same view to the extent possible.

The implementation is a snapshot of the architecture. Imagine an architecture with a write database and a read database. All writes go to the write database, and all reads are directed to one or several read databases in a load-balanced fashion. For a very small site, it may make sense for a single database to accomplish all of these things (with an additional failover database for high availability, of course). The implementation in this case would be a single database, whereas the site is potentially architected (and implemented in the software) for more than one database. Now consider the case where the architecture calls for nearly any database to be used with

the application connecting through an abstracted data access layer (DAL) or data access object (DAO). The specific implementation at a point in time might be a database from Microsoft, but with some modifications to the DAL/DAO it could ultimately become an open source database or database from IBM, Oracle, or Sybase. While the implementation (choice of solution provider) may change over time, the architecture stays somewhat static.

We define technology-agnostic design (TAD) and technology-agnostic architecture (TAA) as the design or architecture of a system that is agnostic toward the technology required to implement it. The aim of TAD and TAA is to separate the design and architecture from the technology employed and the specific implementation. This separation decreases both cost and risk while increasing scalability and availability of your product, system, or platform. Some of our clients even incorporate TAD or TAA into their architectural principles.

TAD and Cost

As we mentioned earlier, architects of buildings and electrical engineers rarely describe their work by giving the names of the vendors providing materials. These architects understand that if they design something appropriately, they open up opportunities for negotiations among competing providers of the aforementioned materials. These negotiations, in turn, help drive down the cost of building (or implementing) the house. Each vendor is subject to a competitive bidding process, where price, quality, and reputation all come into play.

Technology solutions, much like building materials, experience the effects of commoditization over time. A good idea or implementation that becomes successful in an industry is bound to attract competitors. The competitors within the solution space initially compete on differences in functionality and service, but over time these differences decrease as all competitors adopt useful feature sets. In an attempt to forestall the effects of commoditization through increased switching costs, providers of systems and software try to produce proprietary solutions or tools that interact specifically and exclusively with their systems.

Avoiding getting trapped by extensive modification of any provider's solution or adoption of tightly integrated provider tools allows you the flexibility of leveraging the effects of commoditization. As competitors within a solution space begin to converge on functionality and compete on price, you remain free to choose the lowest cost of ownership for any given solution. This flexibility results in capital outlay, which minimizes the impact on cash flow and lowers amortized costs, which positively affects profits on a net income basis. The more your architecture allows you to bring in competing providers or partners, the lower your overall cost structure.

Several times during your career, you are likely to find a provider of technology that is far superior in terms of features and functionality to other providers. You

may determine that the cost of implementing this technology is lower than using the other providers' technology because you have to build less to implement the product. In making such a decision, you should feel comfortable that the "lock-in" opportunity cost of choosing such a provider exceeds the option cost of switching to another provider in the future. In other words, recognize that other providers will move quickly to close the functionality gap, and do your best to ensure that the integration of the service in question can be replaced with products and services from other providers down the road. Recognize further than in using proprietary tools today, you reduce your leverage in negotiating lower prices from other providers in the future.

TAD and Risk

In 2009, several American financial institutions suddenly collapsed; only five years earlier, those institutions would have been considered indestructible. Many independent investment banks that led the storied march of American capitalism collapsed in a matter of weeks or were devoured by larger banks. Many people started to question the futures of Citibank and Bank of America as the government moved to prop them up with federal funds. Fannie Mae and Freddie Mac both received government bailouts and became the object of additional government legislation and regulation. Other industries, perennially teetering on the edge of disaster, such as the American automobile industry, struggled to find ways to remake themselves.

Imagine that you have built a wonderful business producing specialty vans for handicapped people from Ford vehicles. One hundred percent of your business is built around the Ford Econoline Van, and you can't easily retool your factory for another van given the degree of specialization required in tooling and skills. What do you do if Ford goes out of business? What happens if Ford stays in business but increases its prices, significantly changes the Econoline Van family, or increases the interest rate on the loans you use to purchase and customize the vans?

Now, apply a similar set of questions to your implementation technologies (again, not an architecture). Maybe you have used a proprietary set of APIs for some specific asynchronous functionality unique to the database in question. Alternatively, maybe you have leveraged a proprietary set of libraries unique to the application server you've chosen. What do you do when one or both of those providers go out of business? What if the provider of either technology finds itself being sued for some portion of its solution? What if the viability and maintenance of the product relies upon the genius of a handful of people within the company of the provider and those people leave? What if the solution suddenly starts to suffer from quality problems that aren't easily fixed?

Technology-agnostic design reduces these risks by increasing your ability to quickly move to other providers. By reducing your switching costs, you have reduced not only your own costs, but also the risk to your customers and shareholders.

TAD and Scalability

TAD aids scalability in two ways. The first way is that it forces your company and organization to create disciplines around scale that do not depend on any single provider or service. This discipline allows you to scale in multiple dimensions through multiple potential partners, the result of which is a more predictable scalable system independent of any single solution provider.

A common misperception is that by implementing a certain solution, you become reliant upon that solution. Just because you use GoldenGate's database replication technology does not mean that you are dependent upon it alone for scale. True, on any given day, you will rely upon the application to work for proper functioning of your site, but that is not the same as saying that the architecture relies upon it for scale. Again, we separate architecture from implementation. Architecture is a design and should not rely upon any given vendor for implementation. Implementation is a point-in-time description of how the architecture works on that day and at that moment. The proper architecture in this replication scenario would call for a replication mechanism with requirements that can be fulfilled by a number of vendors, of which GoldenGate is but one. If you have done a thorough analysis of the provider landscape and know that you can switch either databases or replication technology providers (again, not without some work), you have a scalable solution that is independent of the provider.

You should not get caught in the trap of saying that you must personally build all components to be truly independently scalable. Recall our discussion in Chapter 15, Focus on Core Competencies: Build Versus Buy. Scalable design allows you to drop in commodity solutions to achieve an implementation. Furthermore, nearly all technologies ultimately move toward commoditization or attract open source alternatives. The result is that you rarely need to build most things that you will need outside of those things that truly differentiate your product or platform.

Review of the Four Simple Questions from Chapter 15

These are the four questions from Chapter 15 that we recommend be used to guide any build versus buy decision:

- Does this component create strategic competitive differentiation? Will we have long-term sustainable differentiation as a result of this component in switching costs, barriers to entry, and so on?
- Are we the best owners of this component or asset? Are we the best equipped to build it and why? Should we sell it if we build it?

- What is the competition for this component? How soon until the competition catches up to us and offers similar functionality?
- Can we build this component cost-effectively? Are we reducing our costs and creating additional shareholder value, and are we avoiding lost opportunity in revenue?

Remember that you are always likely to be biased toward building—so do your best to protect against that bias. The odds are against you in your quest to build a better product than those already available. For this reason, you should tune your bias toward continuing to do what you do well today—your primary business.

The second way that TAD supports scalability is actually embedded within the four questions from Chapter 15. Can you see it? Let's ask two questions to get to the answer. First, do you believe there will be a rapid convergence of the functionality in question? Second, do you need to deeply integrate the solution in question to leverage it? If you believe that competitors will rapidly converge on functionality and you do not need deep integration to the selected provider's solution, you should consider using the solution. In doing so, you avoid the cost of building the solution over the long term. The key, as hinted at by the second question, is to avoid deep integration. You should not be building logic deep within your system to benefit from a provider's solution. Building such logic ties you into the solution provider and makes it more difficult for you to benefit from commoditization—that is, to switch vendors in an effort to reduce price.

We argue that deep integration of a third-party provider's solution is almost never critical to scale if you've properly architected your system, platform, or product. In our experience, we have never come across such a situation that absolutely demands that a company be deeply tied with a third-party provider to scale to the company's needs. In most cases, this misconception is fueled by a poor decision made somewhere else in the architecture. You may, however, find yourself in a situation where a pending or existing crisis can be resolved more quickly by leveraging unique functionality provided by a third party. Should you find yourself in that situation, we recommend the following course of action:

1. Abstract the integration into a service so that future changes are limited to the service and not deeply integrated within your systems. Such an abstraction will limit the switching costs after your crisis is over.
2. Make a commitment to resolve the dependency as soon as possible after the crisis.

Resolving Build Versus Buy and TAD Conflicts

Don't be confused with the apparent conflicts between a build versus buy decision and designing agnostically; the two are actually complementary when viewed properly. We've stated that you should own and have core competencies within your team around architecting your platform, service, or product to scale. This does not mean that you need to build each discrete component of your architecture. Architecture and design are completely separate disciplines from the standpoint of development and implementation. Build versus buy is an implementation or development decision, and TAD and TAA are design and architecture decisions.

A proper architecture allows for many choices, and the build versus buy decision should result in buying only if doing so creates a sustainable competitive advantage.

TAD and Availability

TAD and TAA affect availability in a number of ways, but the most obvious is the way they support the ability to switch providers of technology when one provider has significantly greater availability or quality than other providers. Often, this leadership position changes over time between providers of services, and you are best positioned to take advantage of shifts in that leadership by being agnostic as to the provider. Again, agnosticism in design and architecture leads to benefits to customers and shareholders.

The TAD Approach

Now that we've discussed the reasons for TAD and TAA, let's examine how to approach TAD and TAA. Implementing TAD/TAA is fairly simple and straightforward. At its core, it means designing and architecting platforms using concepts rather than solutions. Pieces of the architecture are labeled with their generic system type (database, router, firewall, payment gateway, and so on) and potentially further described with characteristics or specific requirements (gigabit throughput, 5 terabytes of storage, ETL cloud, and so on).

The approach for TAD is straightforward and consists of three easily remembered rules. The first is to think and draw only "boxes," like those that you might find in a wire diagram. For a physical architecture, these boxes depict the type of system employed but never the brand or model. Router, switch, server, storage, database, and so on are all appropriate boxes to be employed in a physical architecture. Logical architectures define the activities, functions, data flow, and processes of the

system and should also avoid the inclusion of vendor names. For this step, it does not matter that your company might have contracts with specific providers or have an approved provider list. This step is just about ensuring that the design is agnostic, and contracts and approved providers are all implementation-related issues.

The second rule is to remove all references to providers, models, or requirements that demand either a provider or model. It's appropriate to indicate requirements in a design, such as a switch that is capable of 10-gigabit throughput, but it is not appropriate to indicate a switch capable of running a Cisco proprietary protocol. Again, interface requirements have to do with implementation. If you have something running in your production environment that requires the same brand of system in other places, you have already locked yourself into a vendor and violated the TAD approach.

The final rule is to describe any requirements specific to your design in agnostic terms. For example, use the number of SQL transactions per second instead of a proprietary vendor database. Use storage in terms of IOPS, bytes, spindles, or speeds rather than anything specific to any given vendor like a proprietary storage replication system. When in doubt, ask yourself whether your design has forced you into a deal with a vendor for any given reason, or whether the description of the design is biased by any given vendor or open source solution.

The TAD Approach: Three Simple Rules

Here are three simple rules to help guide you in TAD designs:

- In the design itself, think in terms of boxes and wire diagrams rather than prose. Leave the detail of the boxes to the next step.
- In defining boxes and flows, use generic terms—application server, Web server, RDBMS, and so on. Don't use vendor names.
- Describe requirements in industry standards. Stay away from proprietary terms specific to a vendor.

Allow your instincts to guide you. If you "feel" as though you are being pulled toward a vendor in a description or design statement, attempt to "loosen" that statement to make it agnostic.

Teams that employ the JAD and ARB processes should ensure they are following TAD during architecture design and review sessions by enforcing these three rules. If a design shows up for an ARB review with the names of vendors written in the boxes, it should be rejected and the team asked to redesign it appropriately.

A very simple test to determine whether you are violating the technology-agnostic design principles is to check whether any component of your architecture is identified with the name of a vendor. Data flows, systems, transfers, and software that are specifically labeled as coming from a specific provider should be questioned. Ideally, even if for some reason a single provider must be used (remember our arguments that this should never be the case), the provider's name should be eliminated. You simply do not want to give any provider a belief that you have already chosen its solution—doing so will handicap you in negotiations.

Technology agnosticism is as much about culture as it is a process or principle during design. Engineers and administrators tend to be very biased toward specific programming languages, operating systems, databases, and networking devices. This bias is very often a result of past experiences and familiarity. An engineer who knows C++ better than Java, for instance, will obviously be biased toward developing something within C++. An engineer who understands and has worked with Cisco networking equipment her entire career will obviously prefer Cisco to a competitor. This bias is simply human nature, and it is difficult to overcome. As such, it is imperative that the engineers and architects understand the reasons for agnosticism. Bright, talented, and motivated individuals who understand the causality between agnosticism and the maximization of flexibility within scalability and the maximization of shareholder wealth will ultimately begin parroting the need for agnosticism.

Conclusion

This chapter made the case for technology-agnostic design and architecture. Technology-agnostic design lowers cost, decreases risk, and increases both scalability and availability. If implemented properly, TAD complements the build versus buy decision process.

TAD is as much a cultural initiative as it is a process or principle. The biggest barrier to implementing TAD will likely be the natural biases of the engineers and architects for or against certain providers. Ensuring that the organization understands the benefits and reasons for TAD will help overcome these biases. Review the section in Chapter 4, Leadership 101, covering the causal roadmap to success.

Key Points

- The most scalable architectures are not implementations and should not be described as implementations. Vendors, brand names, and open source identifications should be avoided in describing architectures, as these are descriptions of implementations.

- TAD and TAA reduce costs by increasing the number of options and competitors within a competitive selection process.
- TAD and TAA reduce risk by lowering switching costs and increasing the speed with which providers or solutions can be replaced in the event of intellectual property issues or issues associated with business viability of your providers.
- TAD and TAA increase scalability through the reduction of cost to scale and the reduction of risk associated with scale. Where technology solutions are likely to converge, TAD and TAA can help you achieve rapid and cost-effective scale by buying rather than building a solution yourself.
- TAD and TAA increase availability by allowing you to select the highest-quality provider and the provider with the best availability at any point in time.
- TAD and TAA are as much cultural initiatives as they are processes or principles. To effectively implement them, you need to overcome the natural human bias for and against technologies with which we are more or less conversant, respectively.

Chapter 21

Creating Fault-Isolative Architectural Structures

The natural formation of the country is the soldier's best ally.

—Sun Tzu

In the days before full-duplex and 10-gigabit Ethernet, when repeaters and hubs were used within CSMA/CD (carrier sense multiple access with collision detection) networks, collisions among transmissions were common. Collisions reduced the speed and effectiveness of the network, as collided packets would likely not be delivered on their first attempt. Although the Ethernet protocol (then an implementation of CSMA/CD) used collision detection and binary exponential back-off to protect against congestion in such networks, network engineers also developed the practice of segmenting networks to allow for fewer collisions and a faster overall network. This segmentation into multiple collision domains also created a fault-isolative infrastructure wherein a bad or congested network segment would not necessarily propagate its problems to each and every other peer or sibling network segment. With this approach, collisions were reduced, overall speed of delivery in most cases was increased, and failures in any given segment would not bring the entire network down.

Examples of fault isolation exist all around us, including the places where you live and work. Circuit breakers in modern electrical infrastructure (and fuses in older infrastructures) exist to isolate faults and protect the underlying electrical circuit and connected components from damage. This general approach can be applied to not just networks and electrical circuits, but every other component of your product architecture.

Fault-Isolative Architecture Terms

In our practice, we often refer to fault-isolative architectures as *swim lanes*. We believe this term paints a vivid picture of what we wish to achieve in fault isolation.

For swimmers, the swim lane represents both a barrier and a guide. The barrier exists to ensure that the waves that a swimmer produces do not cross over into another lane and interfere with another swimmer. In a race, this helps to ensure that no interference happens to unduly influence the probability that any given swimmer will win the race.

Swim lanes in architecture protect your systems in a similar fashion. Operations of a set of systems within a swim lane are meant to stay within the guide ropes of that swim lane and not cross into the operations of other swim lanes. Furthermore, swim lanes provide guides for architects and engineers who are designing new functionality to help them decide which set of functionality should be placed in which type of swim lane to make progress toward the architectural goal of high scalability.

Swim lane, however, is not the only fault-isolative term used within the technical community. Terms like *pod* are often used to define fault-isolative domains representing a group of customers or set of functionality. *Pudding* is the act of splitting some set of data and functionality into several groups for fault isolation purposes. Sometimes pods are used to represent groups of services; at other times they are used to represent separation of data. Thinking back to our definition of fault isolation as applied to either components or systems, the separation of data or services alone would be fault isolation at a component level only. Although this has benefits to the overall system, it is not a complete fault isolation domain from a systems perspective and as such only protects you for the component in question.

Shard is yet another term that is often used within the technical community. Most often, it describes a database structure or storage subsystem (or the underlying data as in “shard that data set”). *Sharding* is the splitting of these systems into failure domains so that the failure of a single shard does not bring the remainder of the system down as a whole. Usually “sharding” is used when considering methods of performing transactions on segmented sets of data in parallel with the purpose of speeding up some computational exercise. This is also sometimes called horizontal partitioning. A storage system consisting of 100 shards may have a single failure that allows the other 99 shards to continue to operate. As with pods, however, this does not mean that the systems addressing those remaining 99 shards will function properly. We will discuss this concept in more detail later in this chapter.

Slivers, chunks, clusters, and pools are also terms with which we have become familiar over time. *Slivers* is often used as a replacement for shards. *Chunks* is often used as a synonym for pods. *Clusters* is sometimes used interchangeably with pools—especially when there is a shared notion of session or state within the pool—but at other times is used to refer to an active–passive high-availability solution. *Pools* most often references a group of servers that perform similar tasks. This is a fault isolation term but not in the same fashion as swim lanes (as we’ll discuss later). Most often, these are application servers or Web servers performing some portion of

functionality for your platform. All of these terms most often represent components of your overall system design, though they can easily be extended to mean the entire system or platform rather than just its subcomponent.

Ultimately, there is no single “right” answer regarding what you should call your fault-isolative architecture. Choose whatever word you like the most or make up your own descriptive word. There is, however, a “right” approach—designing to allow for scale and graceful failure under extremely high demand.

Common Fault Isolation Terms

Common fault isolation terms include the following:

- **Swim lane** is most often used to describe a fault-isolative architecture from a platform or complete system perspective.
- **Pod** is most often used as a replacement for swim lane, especially when fault isolation is performed on a customer or geographic basis.
- **Shard** is a fault isolation term most often used when referencing the splitting of databases or storage subcomponents. It is typically used in reference to activities to speed up processing through parallelism.
- **Sliver** is a synonym for shard, often also used for storage and database subcomponents.
- **Chunk** is a synonym for pods.
- **Pool** is a fault isolation term commonly applied to software services but is not necessarily a swim lane in implementation.

Benefits of Fault Isolation

Fault-isolative architectures offer many benefits within a platform or product. These benefits range from the obvious benefits of increased availability and scalability to the less obvious benefits of decreased time to market and cost of development. Companies find it easier to roll back releases, as we described in Chapter 18, Barrier Conditions and Rollback, and push out new functionality while the site, platform, or product is “live” and serving customers.

Fault Isolation and Availability: Limiting Impact

As the name would seem to imply, fault isolation greatly benefits the availability of your platform or product. Circuit breakers are great examples of fault isolation.

When a breaker trips, only a portion of your house is impacted. Similarly, when a fault isolation domain or swim lane fails at the platform or systems architecture level, you lose only the functionality, geography, or set of customers that the swim lane serves. Of course, this assumes that you have architected your swim lane properly and that other swim lanes are not making calls to the swim lane in question. A poor choice of a swim lane in this case can result in no net benefit for your availability, so the architecting of swim lanes becomes very important. To explain this, let's look at a swim lane architecture that supports high availability and contrast it with a poorly architected swim lane.

Salesforce, a well-known Software as a Service (SaaS) company that is often credited with having coined the term "pods," was one of AKF Partners' first clients. Tom Keeven, one of our firm's managing partners, was a member of the technical advisory board for Salesforce for a number of years. Tom describes the Salesforce architecture as being multitenant, but not all-tenant. Customers (i.e., companies that use the Salesforce service) are segmented into one of many pods of functionality. Each pod comprises a fault-isolated grouping of customers with nearly all of the base functionality and data embedded within that pod to service the customers it supports. Because multiple customers are in a single pod and occupy that pod's database structure, the solution is multitenant. But because not every customer is in a single database structure for the entire Salesforce solution, it is not "all-tenant." Figure 21.1 depicts what this architecture might look like. This graphic is not an

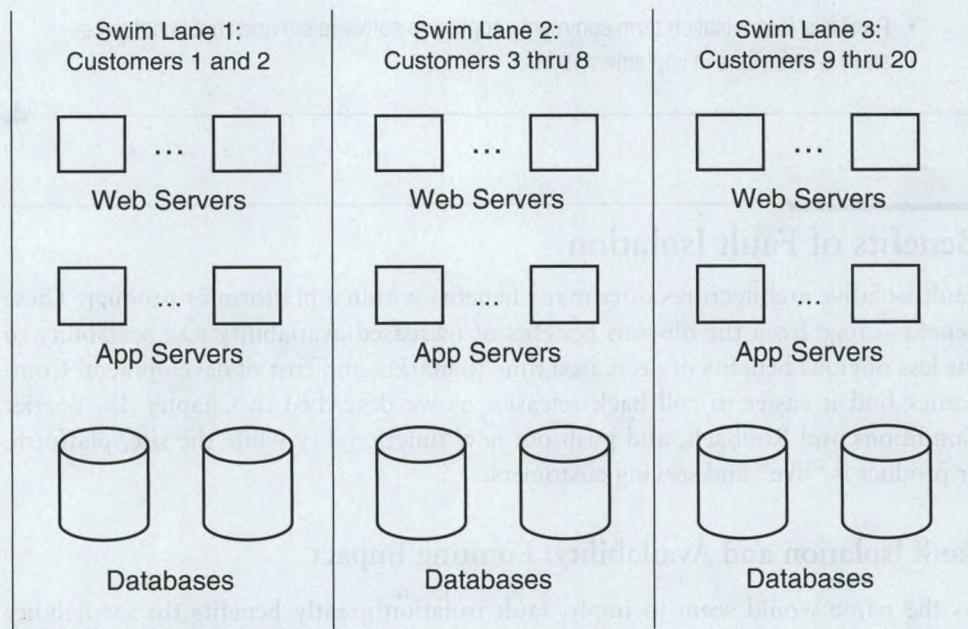


Figure 21.1 *Swim Lane by Customer*

exact replica of the Salesforce.com architecture, but rather is meant to be illustrative of how Web, application, and database servers are dedicated to customer segments.

A fault-isolated architecture like that shown in Figure 21.1 gives companies a great deal of choice as to how they operate their solutions. Want to put a swim lane (or pod) in Europe to meet the expectations of European privacy laws? No problem—the segmentation of data allows this to happen easily. Want to spread swim lanes throughout the United States to have faster response time from a U.S. end user's perspective? Also no problem—each swim lane again is capable of operating independently. Want to create an architecture that allows for easy, risk-free migration to an Infrastructure as a Service (IaaS) offering such as Amazon Web Services (AWS)? The fault-isolated swim lane approach allows you to move one or multiple swim lanes anywhere you would like, with each running on a different underlying infrastructure architecture. An interesting side benefit of these strategies is that should you ever experience a geographically localized event, such as an earthquake, terrorist attack, or data center fire, only the portion of your customers in the pods within that region will be impacted.

Contrast the preceding customer segmentation approach with an approach where fault isolation domains are created on a services level. The Patient Protection and Affordable Care Act (PPACA), often referred to as just the Affordable Care Act (ACA), was signed into law in the United States on March 23, 2010. Shortly thereafter, the U.S. government embarked upon an initiative to create a healthcare exchange through which people in the United States could purchase health insurance. The initial architecture created an interrelated mesh of services behind a single portal (Web site) that would serve the needs of most Americans. Shortly after the launch of this site, however, both the government and consumers were surprised with the lack of responsiveness and general nonavailability of the system. AKF Partners was asked to contribute to the troubleshooting to help get this initiative back on track, and as good, nonpartisan Americans we agreed to do so on a pro bono basis.

The *Time* magazine article “Obama’s Trauma Team” does a great job of laying out all the reasons for the “failure to launch” in the initial go-live of Healthcare.gov. In many respects, it serves as a reason for why we wrote the first edition of this book: The failures were legion. The government initially failed to integrate and manage multiple independent contractors; no single person or entity seemingly was in charge of the entire solution. The government also failed to put into place the processes and management that would oversee the successful operation of the solution post launch. Finally, and most informative for this chapter’s discussion, the government never thought to bring in appropriately experienced architects to design a fault-isolated and scalable solution.¹ Healthcare.gov was made up of a number of services, each of which

1. Brill, Steven. Obama’s trauma team. *Time*, February 27, 2014. <http://time.com/10228/obamas-trauma-team/>.

was interrelated and relied upon the others for some number of transactions. Data services were separated from interaction services. The functioning of the system relied on several third-party systems outside the control of the Healthcare.gov team, such as interactions with Veterans Administration services, Social Security services, and credit validation services. Loosely speaking, when one service calls another service and then hangs waiting for a synchronous response, it creates a serial synchronous chain. Such services act similarly to old-fashioned Christmas tree light strings: When one bulb blows, the whole string goes “lights out.” The more bulbs you have in a series, the higher the probability that one of them will blow and cause a complete system failure. This effect is called the “multiplicative effect of failure of systems in series.”

This is a common fault with many service-oriented designs in general, and with the initial implementation of Healthcare.gov in particular. If services rely on each synchronously, faults will propagate along the synchronous chain. This isn’t an indictment of service-oriented architectures, but rather a warning about how one should think about architecting them. Interacting synchronous services are the opposite of fault isolation—they will, by definition, lower your overall availability.

The Healthcare.gov design team experienced a type of failure common in many engineering organizations: They didn’t ask the question, “How will this solution fail?” Healthcare.gov, upon launch, experienced multiple sources of slowness throughout its environment. Each one ultimately could slow down the entire solution given how the solution components were interrelated.

How could the team have gotten around this problem? The simplest solution would be to create fault isolation zones along state boundaries. The ACA initiative allowed for states to run their own independent exchanges. Connecticut, for example, was one of the first states to successfully implement such an exchange.² Had the federal system implemented exchanges for each of the states that decided not to implement its own independent exchange, it could have significantly reduced the impact of any failure or slowness of the Healthcare.gov site. Furthermore, it could have put the state exchanges closest to the states they served to significantly reduce customer response times. With this approach, data would have been partitioned to further reduce query times, and overall response times per interaction would have decreased significantly. Scalability would have increased. The overall complexity of the system and difficulty in troubleshooting would have been reduced, leading to improved mean time to restore service and defect resolution times.

We have not offered an example of the Healthcare.gov architecture because of our concerns that doing so may aid people of nefarious intent. As indicated, we do not

2. Cohen, Jeff, and Diane Webber. Healthcare.gov recruits leader of successful Connecticut effort. *NPR*, August 8, 2014. <http://www.npr.org/blogs/health/2014/08/26/343431515/healthcare-gov-recruits-leader-of-successful-connecticut-effort>.

use Healthcare.gov here to demonstrate that service-oriented isolation approaches should never be used. Quite the contrary—these approaches are an excellent way to isolate code bases, speed time to market through this isolation, and reduce the scalability requirements for caching for action-specific services. But putting such services in series and having them rely on synchronous responses is a recipe for disaster. You can either ensure that the first-order service (the first one to be called before any other service can be used, such as a login) is so highly available and redundant as to minimize the risk, or you can perform multiple splits to further isolate failures.

The first approach—making the service even more highly available—can be accomplished by adding significantly greater capacity than is typically needed. In addition, the incorporation of markdown functionality (see the sidebar “Markdown Logic Revisited” or Chapter 18 for a review) on a per-company basis might help us isolate certain problems.

Markdown Logic Revisited

Recall from Chapter 18 that we provided an implementation of the architectural principle “design to be disabled” in what we called *markdown functionality*. Markdown functionality enables certain features within a product to be turned off without affecting other features. The typical reason companies invest in markdown functionality is to limit the negative impact of new feature releases on either availability or scalability.

Proper markdown functionality allows a new release to remain in a production environment while the offending code or system is fixed and without rolling back the entire release. The offending code or system is simply taken offline, typically through a software “toggle,” and is brought back online after the cause of unintended behavior is rectified.

The second approach—performing multiple splits to isolate failures—is our preferred method of addressing both scalability and availability. If it used this method, Healthcare.gov could have combined the splits of services and splits of customers on a state basis. Using geo-location services, the government could have routed customers to the appropriate state swim lane (the swim lane in which the individual’s data lived). In the event that a customer moved to another state, his or her data could be moved as well. Remember that such an issue exists anyway, as some states decided to implement their own separate exchanges.

We will discuss these types of splits in greater detail in Chapter 22, Introduction to the AKF Scale Cube; Chapter 23, Splitting Applications for Scale; and Chapter 24, Splitting Databases for Scale. In these chapters, we introduce the AKF Scale Cube and explain how to apply it to services, databases, and storage structures.

Fault Isolation and Availability: Incident Detection and Resolution

Fault isolation also increases availability by making incidents easier to detect, identify, and resolve. If you have several swim lanes, each dedicated to a group of customers, and only a single swim lane goes down, you know quite a bit about the failure immediately—it is limited to the swim lane servicing a set of customers. As a result, your questions to resolve the incident are nearly immediately narrowed. More than likely, the issue is a result of systems or services that are servicing that set of customers alone. Maybe it's a database unique to that customer swim lane. You might ask, "Did we just roll code out to that swim lane or pod?" or more broadly, "What were the most recent changes to that swim lane or pod?"

As the name implies, fault isolation has incredible benefits for incident detection and resolution. Not only does it prevent the incident from propagating throughout your platform, but it also focuses your incident resolution process like a laser and shaves critical time off the restoration of service.

Fault Isolation and Scalability

This is a book on scalability and it should come as no surprise, given that we've included it as a topic, that fault isolation somehow benefits your scalability initiatives. The precise means by which fault isolation affects scalability has to do with how you split your services, as we'll discuss in Chapters 22 through 24, and relates to the architectural principle of scaling out rather than up. The most important thing to remember is that a swim lane must not communicate synchronously with any other swim lane. It can make asynchronous calls with the appropriate timeouts and discard mechanisms to other swim lanes, but you cannot have connection-oriented communication to any other service outside of the swim lane. We'll discuss how to construct and test swim lanes later in this chapter.

Fault Isolation and Time to Market

Creating architectures that allow you to isolate code into service-oriented or resource-oriented systems gives you the flexibility of focus and the ability to dedicate engineers to those services. For a small company, this approach probably doesn't make much sense. As the company grows, however, the lines of code, number of servers, and overall complexity of its system will grow. To handle this growth in complexity, you will need to focus your engineering staff. Failing to specialize and focus your staff will result in too many engineers having too little information on the entire system to be effective.

If you run an ecommerce site, you might have code, objects, methods, modules, servers, and databases focused on checkout, finding, comparing, browsing, shipping,

inventory management, and so on. By dedicating teams to these areas, each team will become an expert on a code base that is itself complex, challenging, and growing. The resulting specialization will allow for faster new-feature development and a faster time to resolve known or current incidents and problems. Collectively, this increase in speed to delivery may result in a faster time to market for bug fixes, incident resolution, and new-feature development.

Additionally, this isolation of development—and ideally isolation of systems or services—will reduce the merge conflicts that would happen within monolithic systems development. Here, we use the term “monolithic systems development” to identify a source that is shared across all set of functions, objects, procedures, and methods within a given product. Duplicate checkouts for a complex system across many engineers will result in an increase in merge conflicts and errors. Specializing the code and the engineering teams will reduce these conflicts.

This is not to say that code reuse should not be a focus for the organization; it absolutely should be a focus. Develop shared libraries, and consider creating a dedicated team that is responsible for governing their development and usage. These libraries can be implemented as services to services, as shared dynamically loadable libraries, or compiled and/or linked during the build of the product. Our preferred approach is to dedicate shared libraries to a team. In the case where a non-shared-library team develops a useful and potentially sharable component, that component should be moved to the shared-library team. This approach is somewhat analogous to that applied in an open source project, where you use the open source code and share it back with the project and owning organization.

Recognizing that engineers like to continue to be challenged, you might be concerned that engineers will not want to spend a great deal of time on a specific area of your site. You can slowly rotate engineers to ensure all of them obtain a better understanding of the entire system and in so doing stretch and develop them over time. Additionally, through this process, you start to develop potential future architects with a breadth of knowledge regarding your system or fast-reaction SWAT team members who can easily dig into and resolve incidents and problems.

Fault Isolation and Cost

In the same ways and for the same reason that fault isolation reduces time to market, it can also reduce cost. In the isolation by services example, as you get greater throughput for each hour and day spent on a per-engineer basis, your per-unit cost goes down. For instance, if it normally took you 5 engineering days to produce the average story or use case in a complex monolithic system, it might now take you 4.5 engineering days to produce the average story or use-case in a disaggregated system with swim lanes. The average per-unit cost of your engineering endeavors was just reduced by 10%!

You can do one of two things with this per-unit cost reduction, both of which impact net income and as a result shareholder wealth. You might decide to reduce your engineering staff by 10% and produce exactly the same amount of product enhancements, changes, and bug fixes at a lower absolute cost than before. This reduction in cost increases net income without any increase in revenue.

Alternatively, you might decide that you will keep your current cost structure and attempt to develop more products at the same cost. The thought here is that you will make great product choices that increase your revenues. If you are successful, you also increase net income and as a result your shareholders will become wealthier.

You may believe that additional sites usually end up costing more capital than running out of a single site and that operational expenses will increase with a greater number of sites. Although this is true, most companies aspire to have products capable of weathering geographically isolated disasters and invest to varying levels in disaster recovery initiatives that help mitigate the effects of such disasters. As we will discuss in Chapter 32, Planning Data Centers, assuming you have an appropriately fault-isolated architecture, the capital and expenses associated with running three or four properly fault-isolated data centers can be significantly less than the costs associated with running two completely redundant data centers.

Another consideration in justifying fault isolation is the effect that it has on revenue. Referring back to Chapter 6, Relationships, Mindset, and the Business Case, you can attempt to calculate the lost opportunity (lost revenue) over some period of time. Typically, this will be the easily measured loss of a number of transactions on your system added to the future loss of a higher than expected customer departure rate and the resulting reduction in revenue. This loss of current and future revenues can be used to determine if the cost of implementing a fault-isolated architecture is warranted. In our experience, some measure of fault isolation is easily justified through the increase in availability and the resulting decrease in lost opportunity.

How to Approach Fault Isolation

The most fault-isolative systems are those that make absolutely no calls and have no interaction with anything outside of their functional or data boundaries. The best way to envision this situation is to think of a group of lead-lined concrete structures, each with a single door. Each door opens into a long isolated hallway that has a single door at each end; one door accesses the lead-lined concrete structure and one door accesses a shared room with an infinite number of desks and people. In each of these concrete structures is a piece of information that any one individual sitting at the many desks might want. To get that information, he has to travel the long hallway dedicated to the room with the information he needs and then walk back to his

desk. After that journey, he may decide to get a second piece of information from the room he just entered, or he might travel down another hallway to another room. It is impossible for a person to cross from one room to the next; he must always make the long journey down the hallway. If too many people get caught up attempting to get to the same room down the same hallway, it will be immediately apparent to everyone in the room; in such a case, they can decide to either travel to another room or simply wait.

In this example, we've not only illustrated how to think about fault-isolative design, but also demonstrated two benefits of such a design. The first benefit is that a failure in capacity of the hallway does not keep anyone from moving on to another room. The second benefit is that everyone knows immediately which room has the capacity problem. Contrast this with an example where each of the rooms is connected to a shared hallway and there is only one entrance to this shared hallway from our rather large room. Although each of the rooms is isolated, should people back up into the hallway, it becomes both difficult to determine which room is at fault and impossible to travel to the other rooms. This example also illustrates our first principle of fault-isolative architecture.

Principle 1: Nothing Is Shared

The first principle of fault-isolative design or architecture is that absolutely nothing is shared. Of course, this is the extreme case and may not be financially feasible in some companies, but it is nevertheless the starting point for fault-isolative design. If you want to ensure that capacity or system failure does not cause problems for multiple systems, you need to isolate system components. This may be very difficult in several areas, like border or gateway routers. That said, and recognizing both the financial and technical barriers in some cases, the more thoroughly you apply this principle, the better your results will be.

One often-overlooked area is URIs/URLs. For instance, consider using different subdomains for different groups. If grouping by customers, consider cust1.allscale.com to custN.allscale.com. If grouping by services, consider view.allscale.com, update.allscale.com, input.allscale.com, and so on. The domain grouping ideally should reference isolated Web and app servers as well as databases and storage unique to that URI/URL. If financing allows and demand is appropriate, dedicated load balancers, DNS, and access switches should be used.

If you identify two swim lanes and have them communicate to a shared database, they are really the same swim lane, not two different ones. You may have two smaller fault isolation zones from a service's perspective (for instance, the application servers), which will help when one application server fails. Should the database fail, however, it will bring down both of these service swim lanes.

Principle 2: Nothing Crosses a Swim Lane Boundary

This is another important principle in designing fault-isolative systems. If you have systems communicating synchronously, they can cause a potential fault. For the purposes of fault isolation, “synchronous” means any transaction that must wait for a response to complete. It is possible, for instance, to have a service use an asynchronous communication method but block waiting for the response to a request. To be asynchronous from a fault isolation perspective, a service must not care about whether it receives a response. The transaction must be “fire and forget” (such as a remote write that can be lost without issue) or “fire and hope.” “Fire and hope” requests follow a pattern of notifying a service that you would like a response and then polling (to some configurable number of times) for a response. The ideal number here is 1 so that you do not create long queue depths of transactions and stall other requests.

Overall, we prefer no communication to happen outside a fault isolation zone and we never allow synchronous communication. Think back to our room analogy: The room and its hallway were the fault isolation zone or domain; the large shared room was the Internet. There was no way to move from one room to the next without traveling back to the area of the desk (our browser) and then starting down another path. As a result, we know exactly where bottlenecks or problems are happening immediately when they occur, and we can figure out how to handle those problems.

Any communication between zones—or paths between rooms in our scenario—can cause problems with our fault isolation. A backup of people in one hallway may be the cause in the hallway connected to that room or any of a series of rooms connected by other hallways. How can we tell easily where the problem lies without a thorough diagnosis? Conversely, a backup in any room may have an unintended effect in some other room; as a result, our room availability goes down.

Principle 3: Transactions Occur Along Swim Lanes

Given the name and the previous principle, this principle should go without saying—but we learned long ago not to assume anything. In technology, assumption is the mother of catastrophe. Have you ever seen swimmers line up facing the length of a pool, but seen the swim lane ropes running widthwise? Of course not, but the competition in the resulting water obstacle course would probably be great fun to watch.

The same is true for technical swim lanes. It is incorrect to say that you’ve created a swim lane of databases, for instance. How would transactions get to the databases? Communication would have to happen across the swim lane, and per Principle 2, that should never happen. In this case, you may well have created a pool, but because transactions cross a line, it is not a swim lane as we define it.

When to Implement Fault Isolation

Fault isolation isn't free and it's not necessarily cheap. Although it has a number of benefits, attempting to design every single function of your platform to be fault isolative would likely be cost prohibitive. Moreover, the shareholder return just wouldn't be there. And that's the response to the preceding heading. After 20 and a half chapters, you probably can sense where we are going.

You should implement just the right amount of fault isolation in your system to generate a positive shareholder return. "OK, thanks, how about telling me how to do that?" you might ask.

The answer, unfortunately, will depend on your particular needs, the rate of growth and unavailability, the causes of unavailability in your system, customer expectations with respect to availability, contractual availability commitments, and a whole host of things that result in a combinatorial explosion, which make it impossible for us to describe for you precisely what you need to do in your environment.

That said, there are some simple rules to apply to increase your scalability and availability. We present some of the most useful here to help you in your fault isolation endeavors.

Approach 1: Swim Lane the Money-Maker

Whatever you do, always make sure that the thing that is most closely related to making money is appropriately isolated from the failures and demand limitations of other systems. If you are operating a commerce site, this might be your purchase flow from the "buy" button and checkout process through the processing of credit cards. If you are operating a content site and you make your money through proprietary advertising, ensure that the advertising system functions separately from everything else. If you are operating a recurring registration fee-based site, ensure that the processes from registration to billing are appropriately fault isolated.

It stands to reason that you might have some subordinate flows that are closely tied to the money-making functions of your site, and you should consider these for swim lanes as well. For instance, in an ecommerce site, the search and browse functionality might need to be in swim lanes. In content sites, the most heavily trafficked areas might need to be in their own swim lanes or several swim lanes to help with demand and capacity projections. Social networking sites may create swim lanes for the most commonly hit profiles or segment profile utilization by class.

Approach 2: Swim Lane the Biggest Sources of Incidents

If in your recurring quarterly incident review (Chapter 8, Managing Incidents and Problems) you identify that certain components of your site are repeatedly causing

other incidents, you should absolutely consider these for future headroom projects (Chapter 11, Determining Headroom for Applications) and isolate these areas. The whole purpose of the quarterly incident review is to learn from past mistakes. Thus, if demand-related issues are causing availability problems on a recurring basis, we should isolate those areas from impacting the rest of our product or platform.

Approach 3: Swim Lane Along Natural Barriers

This is especially useful in multitenant SaaS systems and most often relies upon the *z*-axis of scale discussed later in Chapters 22 to 24. The sites and platforms needing the greatest scalability often have to rely on segmentation along the *z*-axis, which is most often implemented on customer boundaries. Although this split is typically first accomplished along the storage or database tier of architecture, it follows that we should create an entire swim lane from request to data storage or database and back.

Very often, *multitenant* indicates that you are attempting to get cost efficiencies from common utilization. In many cases, this approach means that you can design the system to run one or many “tenants” in a single swim lane. If this is true for your platform, you should make use of it. If a particular tenant is very busy, assign it to a swim lane. A majority of your tenants have very low utilization? Assign them all to a single swim lane. You get the idea.

Fault Isolation Design Checklist

The design principles for fault-isolative architectures are summarized here.

- *Principle 1: Nothing is shared* (also known as “share as little as possible”). The less that is shared within a swim lane, the more fault isolative the swim lane becomes.
- *Principle 2: Nothing crosses a swim lane boundary*. Synchronous (defined by expecting a request—not the transfer protocol) communication never crosses a swim lane boundary; if it does, the boundary is drawn incorrectly.
- *Principle 3: Transactions occur with swim lanes*. You can’t create a swim lane of services because the communication to those services would violate Principle 2.

The approaches for fault-isolative architectures are as follows:

- *Approach 1: Swim lane the money-maker*. Never allow your cash register to be compromised by other systems.

- *Approach 2: Swim lane the biggest sources of incidents.* Identify the recurring causes of pain and isolate them.
- *Approach 3: Swim lane natural barriers.* Customer boundaries make good swim lanes.

Although a number of approaches are possible, these will go a long way toward increasing your scalability while not giving your CFO a heart attack.

How to Test Fault-Isolative Designs

The easiest way to test a fault-isolative design is to draw your platform at a high level on a whiteboard. Add dotted lines for any communication between systems, and solid lines indicating where you believe your swim lanes do exist or should exist. Anywhere a dotted line crosses a solid line indicates a violation of a swim lane. From a purist perspective, it does not matter if that communication is synchronous or asynchronous, although synchronous transactions and communications are a more egregious violation from both a scalability and an availability perspective. This test will identify violations of the first and second principles of fault-isolative designs and architectures.

To test the third principle, simply draw an arrow from the user to the last system on your whiteboard. The arrow should not cross any lines for any swim lane; if it does, you have violated the third principle.

Conclusion

In this chapter, we discussed the need for fault-isolative architectures, principles of implementation, approaches for implementation, and finally a design test. We most commonly use swim lanes to identify a completely fault-isolative component of an architecture, although terms like “pods” and “slivers” are often used to mean the same thing.

Fault-isolative designs increase availability by ensuring that subsets of functionality do not diminish the overall functionality of your entire product or platform. They further aid in increasing availability by allowing for immediate detection of the areas causing problems within the system. They lower both time to market and costs by allowing for dedicated, deeply experienced resources to focus on the swim lanes and by reducing merge conflicts and other barriers and costs to rapid development. Scalability is increased by allowing for scale in multiple dimensions as discussed in Chapters 22 through 24.

The principles of swim lane construction address sharing, swim lane boundaries, and swim lane direction. The fewer things that are shared within a swim lane, the more isolative and beneficial that swim lane becomes to both scalability and availability. Swim lane boundaries should never have lines of communication drawn across them. Swim lanes always move in the direction of communication and customer transactions and never across them.

Always address the transactions making the company money first when considering swim lane implementation. Then, move functions causing repetitive problems into swim lanes. Finally, consider the natural layout or topology of the site for opportunities to swim lane, such as customer boundaries in a multitenant SaaS environment.

Key Points

- A swim lane is a fault-isolative architecture construct in which a failure in the swim lane is not propagated and does not affect other platform functionality.
- “Pod,” “shard,” and “chunk” are often used in place of the term “swim lane,” but they may not represent a “full system” view of functionality and fault isolation.
- Fault isolation increases availability and scalability while decreasing time to market and costs of development.
- The less you share in a swim lane, the greater that swim lane’s benefit to availability and scalability.
- No communication or transaction should ever cross a swim lane boundary.
- Swim lanes go in the direction of—and never across—transaction flow.
- Swim lane the functions that directly impact revenue, followed by those that cause the most problems and any natural boundaries that can be defined for your product.

Chapter 22

Introduction to the AKF Scale Cube

Ponder and deliberate before you make a move.

—Sun Tzu

In Chapter 21, we referred several times to the AKF Scale Cube to highlight methods by which components of our architecture might be split into swim lanes or failure domains. In this chapter, we reintroduce the AKF Scale Cube. We developed the scale cube to help our clients think about how to split services, data, and transactions, and to a lesser degree, teams and processes.

The AKF Scale Cube

Imagine a cube drawn with the aid of a three-dimensional axis for a guide. We will call the point of intersection of our three axes the *initial point*, as referenced by the values $x = 0$, $y = 0$, and $z = 0$. Figure 22.1 shows this cube and these three axes. Each axis will describe a unique method of scaling products, processes, and teams.

The initial point, with coordinates of $(0, 0, 0)$, is the point of least scalability within any system. It consists of a single monolithic solution deployed on a single server. It might scale “up” with larger and faster hardware, but it won’t scale “out.” As such, it will limit your growth to that which can be served by a single unit. In other words, your system will be bound by how fast the server runs the application in question and how well the application is tuned to use the server. We call work on any of the three axes a “split,” as in “an x -axis split” or a “ y -axis split.”

Making modifications to your solution for the purposes of scaling moves you along one of the three axes. Equivalent effort applied to any axis will not always return equivalent results. For instance, a week of work applied to the x -axis might allow transaction growth to scale very well but not materially impact your ability to

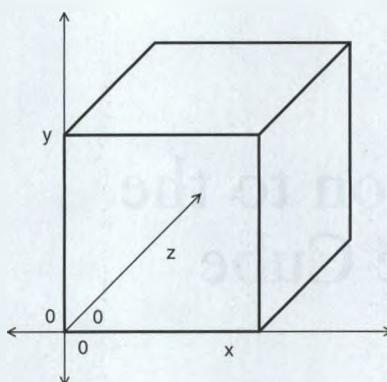


Figure 22.1 AKF Scale Cube with Axes

store, retrieve, and search through product-related information. To address such a storage or memory constraint, you might need to consider a week of work along the y -axis. Such y -axis splits might allow you to split information for the purposes of searching through it faster, but may hinder your efforts when the joining of information is required—for this, you might need to consider z -axis splits.

Choosing one axis does not preclude you from making use of other axes later. Recall from Chapter 19, Fast or Right?, that the cost to design splits is comparatively low relative to the cost of implementing and deploying those splits. As such, we want to think about how we would make use of each of the three axes in our cube in our designs. When we make choices about what to implement, we should select the splits that have the highest return (in terms of scale) for our effort and that meet our needs in terms of delivering scalability in time for customer demand.

In the following sections, we discuss the meaning of each of the axes at a very high level. In Chapter 23, Splitting Applications for Scale, and Chapter 24, Splitting Databases for Scale, we will dig deeper into the most common applications of each of these axes: splitting services and splitting databases.

The x -Axis of the Cube

The x -axis of the AKF Scale Cube represents cloning of services and data with absolutely no bias. Perhaps the easiest way to represent such a split is to think first in terms of people and organizations. Let's first consider the days of yore in which typing pools handled the typing of meeting minutes, letters, internal memos, and so on. Use of the term *pool* dates back as far as 70 or more years as a means of identifying a single logical service (typing) distributed among several entities (in this case

people). Work would be sent to the typing pool largely without a bias as to which individual typist performed the work. This distribution of work among clones is a perfect example of *x*-axis scalability.

Another people example to illustrate the *x*-axis might be found within the accounts receivable or accounts payable portion of your company's finance organization. Initially, for small to medium companies that do not outsource this kind of work, the groups might consist of a few people, each of whom can perform all of the tasks within his or her area. The accounts payable staff can all receive bills and generate checks based on a set of processes and send those checks out or get them countersigned depending on the amount of the check. The accounts receivable staff is capable of generating invoices from data within the system, receiving checks, making appropriate journal entries, and depositing the checks. Each person can do all of the tasks, and it does not matter to whom the work goes.

Both of these examples illustrate the basic concept of the *x*-axis, which is the unbiased distribution of work across clones. Each clone can do the work of the other clones, and there is no bias with respect to where the work travels (other than individual efficiency). Each clone has the tools and resources to get the work done and will perform the work given to it as quickly as possible.

The *x*-axis seems great! When we need to perform more work, we just add more clones. Is the number of memoranda exceeding your current typing capacity? Simply add more typists! Is your business booming and there are too many invoices to create and too many payments coming in? Add more accounts receivable clerks! Why would we ever need any more axes?

Let's return to our typing pool first to answer this question. Assume that to complete some of our memoranda, external letters, and notes, a typist needs to have certain knowledge. Suppose also that as the company grows, the services offered by the typing pool increase. The pool now performs some 100 different types and formats of services, and the work is not evenly distributed across these types of services. External client letters have several different formats that vary by the type of content included within the message, memoranda vary by content and intent, meeting notes vary by the type of meeting, and so on. Now an individual typist may get some work done very fast (the work that is most prevalent throughout the pool) but also be required to spend time looking up the less frequently encountered formatting, which in turn slows down the entire pipeline of work. As the type of work increases for any given service, more time may be spent trying to get work of varying sizes done, and the instruction set to accomplish this work may not be easily kept in any given typist's head.

These are all examples of problems associated with the *x*-axis: It simply does not scale well with an increase in data, either as instruction sets or reference data. The same holds true if the work varies by the sender or receiver. For instance, maybe vice

presidents and senior executives get special formatting or are allowed to send different types of communication than directors of the company. Perhaps the sender uses special letterhead or stock. Maybe the receiver of the message triggers a variation in tone of communication or paper stock. Account delinquent letters, for instance, may require a special tone not referenced within the notes to be typed.

As another example, consider again our accounts receivable group. This group obviously performs a very wide range of tasks, from the invoicing of clients to the receipt of bills, the processing of delinquent accounts, and the deposit of funds into our bank accounts. The processes for each of these tasks grows as the company grows, and our controller will certainly want some specific process controls to exist so that money doesn't errantly find its way out of the accounts receivable group and into an employee's pocket before payday! This is another place where scaling for transaction growth alone is not likely to allow us to scale cost-effectively into a multibillion-dollar company. We will likely need to perform splits based on the services this group performs and/or the clients or types of clients they serve. These splits are addressed by the *y*- and *z*-axes of our cube, respectively.

The *x*-axis split tends to be easy to understand and implement and fairly inexpensive in terms of capital and time. Little additional process or training is necessary, and managers find it easy to distribute the work. Our people analogy holds true for systems as well, which we will see in Chapters 23 and 24. The *x*-axis works well when the distribution of a high volume of transactions or work is all that we need to do.

Summarizing the *x*-Axis

The *x*-axis of the AKF Scale Cube represents the cloning of services or data such that work can easily be distributed across instances with absolutely no bias.

The *x*-axis implementations tend to be easy to conceptualize and typically can be implemented at relatively low cost.

The *x*-axis implementations are limited by growth in the instructions to accomplish tasks and growth in the data necessary to accomplish tasks.

The *y*-Axis of the Cube

The *y*-axis of the AKF Scale Cube represents a separation of work responsibility by either the type of data, the type of work performed for a transaction, or a combination of both. One way to view these splits is as a split by responsibility for an action.

We often refer to these outcomes as *service- or resource-oriented splits*. In a *y-axis* split, the work for any specific action or set of actions, as well as the information and data necessary to perform that action, is split away from other types of actions. This type of split is the first split that addresses the monolithic nature of work and the separation of the same into either pipelined workflows or parallel processing flows. Whereas the *x-axis* is simply the distribution of work among several clones, the *y-axis* represents more of an “industrial revolution” for work: We move from a “job shop” mentality to a system of greater specialization, just as Henry Ford did with his automobile manufacturing assembly line. Rather than having 100 people creating 100 unique automobiles, with each person doing 100% of the tasks, we now have 100 unique individuals performing subtasks such as engine installation, painting, windshield installation, and so on.

Let’s return to our previous example of a typing service pool. In our *x-axis* example, we identified that the total output of our pool might be hampered as the number and diversity of tasks grew. Specialized information might be necessary based on the nature of typing work performed: An internal memorandum might take on a significantly different look than a memo meant for external readers, and meeting notes might vary by the type of meeting. The vast majority of the work might consist of letters to clients of a certain format and typed on a specific type of letterhead and bond. When a typist is presented with one of the 100 or so formats that represent only about 10% to 20% of the total work, he or she might have to stop and look up the appropriate format, grab the appropriate letterhead and/or bond, and so on. One approach to this situation might be to create much smaller pools specializing in some of the more common requests within this 10% to 20% of the total work and a third pool that handles the small minority of the remainder of the common requests. Both of these new service pools could be sized proportionally to the work.

The expected benefit of such an approach would be a significant increase in the throughput of the large pool representing a vast majority of the requests. This pool would no longer “stall” on a per-typist basis when presented with a unique request. Furthermore, for the next largest pool of typists, some specialization would happen for the next most common set of requests, and the output expectations would be the same; for those sets of requests, typists would be familiar with them and capable of handling them much more quickly than before. The remaining set of requests that represent a majority of formats but a minority of request volume would be handled by the third pool; although throughput would suffer comparatively, it would be isolated to a smaller set of people who might also at least have some degree of specialization and knowledge. The overall benefit should be that throughput increases significantly. Notice that in creating these pools, we have also created a measure of fault isolation, as identified in Chapter 21, Creating Fault-Isolative Architectural

Structures. Should one pool stall due to paper issues and such, the entire “typing factory” does not grind to a halt.

It is also easy to see how the separation of responsibilities would be performed within our running example of the accounts receivable department. Each unique action could become its own service. Invoicing might be split off into its own team or pool, as might payment receiving/journaling and deposits. We might further split late payments into its own special group that handles collections and bad debt. Each of these functions is associated with a unique set of tasks that require unique data, experience, and instructions or processes. By splitting them, we reduce the amount of information any specific person needs to perform his or her job, and the resulting specialization should allow us to perform processing faster. The *y*-axis industrial revolution has saved us!

Although the benefits of the *y*-axis are compelling, such splits tend to cost more than the simpler *x*-axis splits. The reason for the increase in cost is that to perform the *y*-axis split, very often some rework or redesign of process, rules, software, and the supporting data models or information delivery system is required. Most of us don’t think about splitting up the responsibilities of our teams or software when we are running a three-person company or operating a Web site running on a single server. Additionally, the splits themselves create some resource underutilization initially that manifests itself as an initial increase in operational cost.

The benefits of the *y*-axis are numerous, however. Although *y*-axis splits help manage the growth in transactions, they also help scale what something needs to know to perform those transactions. The data that is being operated upon as well as the instruction set to operate that data decreases, which means that people and systems can be more specialized, resulting in higher throughput on a per-person or per-system basis. Even a simple *y*-axis split will also give us a first level of fault isolation. In the accounts receivable example, if a member of a bad debts “pool” is sick for the day, only his or her service pool will be impacted.

Summarizing the *y*-Axis

The *y*-axis of the AKF Scale Cube represents separation of work by responsibility, action, or data.

The *y*-axis splits are easy to conceptualize but typically come at a slightly higher cost than the *x*-axis splits.

The *y*-axis splits aid in scaling not only transactions, but also instruction size and data necessary to perform any given transaction.

The z-Axis of the Cube

The *z*-axis of the cube is a split biased most often by the requestor or customer or, alternatively, by something that we know about the requestor or customer. The bias here is focused on data and actions that are unique to the person for whom the request is being performed. While *z*-axis splits may or may not address the monolithic nature of instructions, processes, or code, they very often do address the monolithic nature of the data necessary to perform these instructions, processes, or code.

To perform a *z*-axis split of our typing service pool, we might look at both the people who request work and the people to whom the work is being distributed. In analyzing the request work, we can look at segments or classes of groups that might require unique work or represent exceptional work volume. It's likely the case that executives represent a small portion of our total employee base, yet account for a majority or supermajority of the work for internal distribution. Furthermore, the work for these types of individuals might be somewhat unique in that executives are allowed to request more types of work to be performed. Maybe we will limit internal memoranda to executive requests, or say that personal customer notes might be requested only by an executive. A specialist pool of typists might best serve this unique volume and type of work. We might also dedicate one or more typists to the CEO of the company, who likely has the greatest number and variety of requests. All of these are examples of *z*-axis splits.

In our accounts receivable department, we might decide that some customers require specialized billing, payment terms, and interaction unique to the volume of business they do with us. We might dedicate a group of our best financial account representatives and even a special manager to one or more of these customers to handle their unique demands. In so doing, we would reduce the amount of knowledge necessary to perform the vast majority of our billing functions for the majority of our customers while creating account specialists for our most valuable customers. We would expect these actions to increase the throughput of our standard accounts group, as they need not worry about special terms. The relative throughput for special accounts should also increase, as these individuals specialize in that area and are familiar with the special processes and payment terms.

Although *z*-axis splits are sometimes the most costly for companies to implement, the returns from them (especially from a scalability perspective) are usually phenomenal. Specialized training in the previous examples represents a new cost to the company, and this training is an analog to the specialized set of services one might need to create within a systems platform. Data separation can become costly for some companies, but when performed can be amortized over the life of the platform or the system.

An additional benefit that z-axis splits create is the ability to separate services by geography. Want to locate your accounts receivable group closer to the accounts they support to decrease mail delays? Easy to do! Want your typing pool to be close to the executives and people they support to limit interoffice mail delivery (remember these are the days before email)? Also simple to do!

Summarizing the z-Axis

The z-axis of the AKF Scale Cube represents separation of work by customer or requestor.

Like x- and y-axis splits, z-axis splits are easy to conceptualize, but very often can be difficult and costly to implement for companies.

The z-axis splits aid in scaling transactions and data and may aid in scaling instruction sets and processes if implemented properly.

Putting It All Together

Why would we ever need more than one, or maybe two, axes of scale within our platform or organizations? The answer is that your needs will vary by your company's current size and expected annual growth. If you expect your organization to stay small and grow slowly, you may never need more than one axis of scale. In contrast, if you expect to grow quickly, or if growth is unexpected and violent, you are better off having planned for that growth in advance. Figure 22.2 depicts our cube, the axes of the cube, and the appropriate labels for each of the axes.

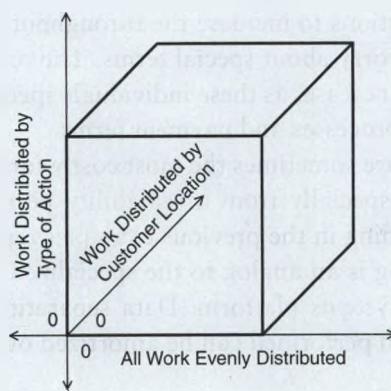


Figure 22.2 AKF Scale Cube

The *x*-axis of scale is very useful and easy to implement, especially if you have stayed away from creating state within your system or team. You simply clone the activity among several participants. Scaling along the *x*-axis starts to fail, however, when you have a lot of different tasks requiring significantly different information from many potential sources. Fast transactions start to run at the speed of slow transactions, and everything starts to work suboptimally.

State Within Applications and the *x*-Axis

You may recall from Chapter 12 that we briefly defined stateful systems as “those in which operations are performed within the context of previous and subsequent operations.” We indicated that state very often drives up the cost of the operations of systems, as most often the state (previous and subsequent calls) is maintained within the application or a database associated with the application. The associated data often increases memory utilization, storage utilization, and potentially database usage and licenses.

Stateless systems may allow us to break the affinity between a single user and a single server. Because subsequent requests can go to any server clone, the *x*-axis becomes even easier to implement. The lack of affinity between customer and server means that we need not design systems specific to any type of customer and so forth. Systems are now free to be more uniform in composition. This topic will be covered in more detail in Chapter 26, Asynchronous Design for Scale.

The *y*-axis helps to solve that problem by isolating transaction type and speed to systems and people specializing in that area of data or service. Slower transactions are now bunched together, but because the data set has been reduced relative to the *x*-axis only example, they run faster than they had previously. Fast transactions also speed up because they are no longer competing for resources with the slower transactions and their data set has been reduced. Monolithic systems are reduced to components that operate more efficiently and can scale for data and transaction needs.

The *z*-axis not only helps us scale transactions and data, but may also help with monolithic system deconstruction. Furthermore, we can now move teams and systems around geographically and start to gain benefits from this geographic dispersion, such as disaster recovery.

Looking at our pool of typists, we can separate the types of work that the typists perform based on the actions involved. We might create a customer-focused team responsible for general customer communication letters, an internal memos team, and a team focused on meeting minutes—all of these are examples of the *y*-axis. Each team is likely to have some duplication to allow for growth in transactions

within that team, which is an example of *x*-axis scale. Finally, we might decide that some members of the team should specialize to handle specific customers or requestors such as an executive group. Although this is a *z*-axis split, these teams may also have specialization by task (*y*-axis) and duplication of team members (*x*-axis). Aha! We've put all three axes together.

Think of how you not only addressed a problem of scale with this example, but also reduced some of the risk in introducing change. If you have a new process you want to try, you can introduce it to a group within a service pool (along the *x*-axis), to an entire functional team (along the *y*-axis), or to a region (along the *z*-axis). You can see how the new process works with a limited group of people, and decide after measuring it whether to roll it out to the other teams, to tweak it before introducing it everywhere, or to roll it back if it isn't working as expected.

For our accounts receivable department, we have split the groups based on the invoicing, receiving, and deposits activities, all of which are *y*-axis splits. Each group has multiple members performing the same task, which is an *x*-axis split. We have created special separation of these teams focused on major accounts and recurring delinquent accounts, and each of these specialized teams (a *z*-axis split) has further splits based on function (*y*-axis) and duplication of individuals (*x*-axis).

AKF Scale Cube Summary

Here is a summary of the three axes of scale:

- The *x*-axis represents the distribution of the same work or mirroring of data across multiple entities.
- The *y*-axis represents the distribution and separation of work responsibilities or data meaning among multiple entities.
- The *z*-axis represents the distribution and segmentation of work by customer, customer need, location, or value.

Hence, *x*-axis splits are mirror images of functions or data, *y*-axis splits separate data based on data type or type of work, and *z*-axis splits separate work by customer, location, or some value-specific identifier (e.g., a hash or modulus).

When and Where to Use the Cube

We will discuss the topic of where and when to use the AKF Scale Cube in Chapters 23, Splitting Applications for Scale, and 24, Splitting Databases for Scale.

That said, the cube is a tool and reference point for nearly any discussion around scalability. You might make a representation of it within your scalability, 10x, or headroom meetings—a process that was discussed in Chapter 11, Determining Headroom for Applications. The AKF Scale Cube should also be presented during Architecture Review Board (ARB) meetings, as discussed in Chapter 13, if you adopt a principle requiring the design of more than one axis of scale for any major architectural effort. It can serve as a basis for nearly any conversation around scale, because it helps to create a common language among the engineers of an organization. Rather than talking about specific approaches, teams can focus on concepts that might evolve into any number of approaches.

You might consider requiring footnotes or light documentation indicating the type of scale for any major design within the joint architecture design (JAD) process introduced in Chapter 13, Joint Architecture Design and Architecture Review Board. The AKF Scale Cube can also come into play during problem resolution and postmortems in identifying how intended approaches to scale did or did not work as expected and suggesting how to fix them in future endeavors.

Agile teams can use the AKF Scale Cube to use a single “scale” language within the team. Use it during Agile designs and have someone ask the question, “How does this solution scale, and along which axis does it scale?”

The AKF Scale Cube is a tool best worn on your tool belt rather than placed in your toolbox. It should be carried at all times, as it is lightweight and can add significant value to you and your team. If referenced repeatedly, it can help to change your culture from one that focuses on specific fixes to one that discusses approaches and concepts to help identify the best potential fix. It can switch an organization from thinking like technicians to acting like engineers.

Conclusion

This chapter reintroduced the concept of the AKF Scale Cube. This cube has three axes, each of which focuses on a different approach toward scalability. Organizational construction may be used as an analogy for systems to help better reinforce the approach of each of the three axes of scale. The cube is constructed such that the initial point ($x = 0, y = 0, z = 0$) is a monolithic system or organization (single person) performing all tasks with no bias based on the task, customer, or requestor.

Growth in people or systems performing the same tasks represents an increase in the x -axis. This axis of scale is easy to implement and typically comes at the lowest cost, but it suffers when the number of types of tasks or data necessary to perform those tasks increases.

A separation of responsibilities based on data or the activity being performed is growth along the y -axis of our cube. This approach tends to come at a slightly

higher cost than *x*-axis growth, but also benefits from a reduction in the data necessary to perform a task. Other benefits of such an approach include some fault isolation and an increase in throughput for each of the new pools based on the reduction of data or instruction set.

A separation of responsibility based on customer or requestor represents growth along the *z*-axis of scale. Such separation may allow for reduction in the instruction set for some pools and almost always reduces the amount of data necessary to perform a task. The result is that both throughput and fault isolation are often increased. The cost of *z*-axis splits tends to be the highest of the three approaches in most organizations, although the return is also huge. The *z*-axis split also allows for geographic dispersion of responsibility.

Not all companies need all three axes of scale to survive. Some companies may do just fine with implementing the *x*-axis. Extremely high-growth companies should plan for at least two axes of scale and potentially all three. Remember that planning (or designing) and implementing are two separate functions.

Ideally the AKF Scale Cube, or a construct of your own design, will become part of your daily tool set. Using such a model helps reduce conflict by focusing on concepts and approaches rather than specific implementations. If added to JAD, ARB, and headroom meetings, it helps focus the conversation and discussion on the important aspects and approaches to growing your technology platform.

Key Points

- The AKF Scale Cube offers a structured approach and concept for discussing and solving scale. The results are often superior to a set of rules or implementation-based tools.
- The *x*-axis of the AKF Scale Cube represents the cloning of entities or data and an equal unbiased distribution of work across them.
- The *x*-axis tends to be the least costly to implement, but suffers from constraints in instruction size and data set.
- The *y*-axis of the AKF Scale Cube represents separation of work biased by activity or data.
- The *y*-axis tends to be more costly than the *x*-axis but solves issues related to instruction size and data set size in addition to creating some fault isolation.
- The *z*-axis of the AKF Scale Cube represents separation of work biased by the requestor or person for whom the work is being performed.
- The *z*-axis of the AKF Scale Cube tends to be the most costly to implement but very often offers the greatest scale. It resolves issues associated with data set size and may or may not solve instruction set issues. It also allows for global distribution of services.

- The AKF Scale Cube can be an everyday tool used to focus scalability-related discussions and processes on concepts. These discussions result in approaches and implementations.
- Designing and implementing are two different functions. The cost to design is relatively much lower than the cost to implement. Therefore, if you design to scale along the three dimensions of the AKF Scale Cube early in the product development life cycle, you can choose to implement or deploy later when it makes sense for your business.
- ARB, JAD, Agile design, and headroom are all process examples where the AKF Scale Cube might be useful.

Chapter 23

Splitting Applications for Scale

Whether to concentrate or to divide your troops must be decided by circumstances.

—Sun Tzu

The previous chapter introduced the model by which we describe splits to allow for nearly infinite scale. Now we'll apply those concepts to our real-world product needs. To do this, we'll separate the product into pieces that address our application and service offerings (covered in this chapter) and the splits necessary to allow our storage and databases to scale (covered in the next chapter). The same model and set of principles hold true for both approaches, but the implementation varies enough that it makes sense for us to address them in two separate chapters.

The AKF Scale Cube for Applications

Whether applied to databases, applications, storage, or even organizations, the underlying meaning of the AKF Scale Cube does not change. However, given that we will now use this tool to accomplish a specific purpose, we will add more specificity to the axes. These additional descriptions remain true to the original but provide greater clarity for the architecting of applications to allow for greater scalability. Let's first start with the AKF Scale Cube from the end of Chapter 22.

In Chapter 22, we defined the *x*-axis of our cube as the cloning of services and data with absolutely no bias. In the *x*-axis approach to scale, the only thing that is different between one system and 100 systems is that the transactions are evenly split between those 100 systems as if each was a single instance capable of handling 100% of the original requests rather than the 1% that they actually do handle. We will rename our *x*-axis as horizontal duplication/cloning of services to make it more obvious how we will apply this to our architecture efforts.

The *y*-axis is represented as a separation of work responsibility by either the type of data, the type of work performed for a transaction, or a combination of both. We

most often describe this as a service-oriented split within an application; as such, we will now label this axis as a split by function or service. Here, “function” and “service” are indicative of the actions performed by your platform, but they can just as easily be resource-oriented splits, such as those based on the object upon which an action is being taken. A function- or service-oriented split should be thought of as occurring along action or “verb” boundaries, whereas a resource-oriented split most often takes place along “noun” boundaries. We’ll describe these splits later in this chapter.

The *z*-axis focuses on data and actions that are unique to the person or system for which the request is being performed. We sometimes refer to the *z*-axis as being a “lookup-oriented” split in applications. The “lookup” term indicates that users or data are subject to a non-action-oriented bias that is represented somewhere else within the system. We store the relationships of users to their appropriate split or service somewhere, or determine an algorithm such as a hash or modulus of a user ID that will reliably and consistently send us to the right location set of systems to get the answers for the set of users in question. Alternatively, we may apply an indiscriminate function to the transaction (e.g., a modulus or hash) to determine where to send the transaction.

The new AKF Scale Cube for applications now looks like Figure 23.1.

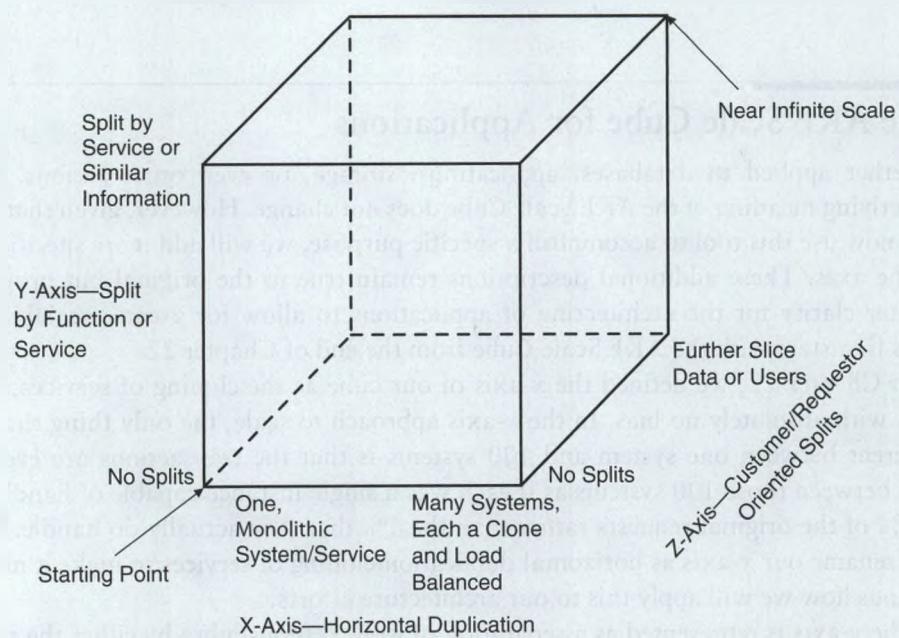


Figure 23.1 AKF Application Scale Cube

The *x*-Axis of the AKF Application Scale Cube

The *x*-axis of the AKF Application Scale Cube represents cloning of services with absolutely no bias. As described previously, if we have a service or platform that is scaled using the *x*-axis alone and consisting of N systems, each of the N systems can respond to any request and will give exactly the same answer as the other ($N - 1$) systems. There is no bias based on service performed, customer, or any other data element. For example, the login functionality exists in the same location and application as the shopping cart, checkout, catalog, and search functionality. Regardless of the request, it is sent to one of the N systems that constitute our *x*-axis split.

The *x*-axis approach is simple to implement in most cases. You simply take exactly the same code that existed in a single instance implementation and put it on multiple servers. If your application is not “stateful,” simply load balance all of the inbound requests to any of the N systems. If you are maintaining data associated with user state or otherwise require persistence from a user to an application or Web server (i.e., the application is “stateful”), the implementation is slightly more difficult. In the cases where persistency or state (or persistency resulting from the need for state) is necessary, a series of transactions from a single user are simply pegged to one of the N instances of the *x*-axis split. This can be accomplished with session cookies from a load balancer. Additionally, as we will discuss in more detail in Chapter 26, Asynchronous Design for Scale, certain methods of centralizing session management can be used to allow any of N systems to respond to an individual user’s request without requiring persistency to that system.

The *x*-axis split has several benefits and drawbacks. Most notably, this split is relatively simple to envision and implement. The *x*-axis also allows for near-infinite scale from a number of transactions perspectives. When your applications or services are hosted, it does not increase the complexity of your hosting environment. Drawbacks of the *x*-axis approach include the inability of this split to address scalability from a data/cache perspective or instruction complexity perspective.

As stated, *x*-axis splits are easy to envision and implement. As such, when you face the prospect of developing a quick solution to any scale initiative, *x*-axis splits should be one of the first options that you consider. Because it is generally easy to clone services, the cost impact in terms of design expense and implementation expense is low. Furthermore, the additional time-to-market cost to release functionality with an *x*-axis split is generally low compared to other implementations, as you are simply cloning the services in question and can easily automate this task.

In addition, *x*-axis splits allow us to easily scale our platforms with the number of inbound transactions or requests. If you have a single user or small number of users who grow from making 10 requests per second to 1,000 requests per second, you

need add only roughly 100 times the number of systems or cloned services to handle the increase in requests.

Finally, the team responsible for managing the services of your platform does not need to worry about a vast number of uniquely configured systems or servers. Every system with an *x*-axis split is roughly equivalent to every other system with the same split. Configuration management of all servers is relatively easy to perform, and new service implementation is as simply as cloning an existing system or generating a machine instance or virtual machine. Configuration files likely do not vary, and the only things the Agile team needs to be concerned about are the total number of systems in an *x*-axis implementation and whether each is getting an appropriate amount of traffic. In IaaS cloud environments, this approach is used for auto-scaling.

Although *x*-axis splits scale well with increased transaction volumes, they do not address the problems incurred with increasing amounts of data. Consider the case where a product must cache a great deal of data to serve client requests. As that data volume grows, the time to serve any given request will likely increase, which is obviously bad for the customer experience. Additionally, the product may become constrained on the server or application itself if the data size becomes too unwieldy. Even if caching isn't required, the need to search through data on other storage or database systems will likely expand as the customer base and/or product catalog increases in size.

In addition, *x*-axis splits don't address the complexity of the software implementing your system, platform, or product. Everything in an *x*-axis split alone is assumed to be monolithic in nature; as a result, applications will likely start to slow down as servers page instruction/execution pages in and out of memory to perform different functions. As a product becomes more feature rich, monolithic applications slow down and become more costly and less easily scaled, either as a result of this instruction complexity or because of the data complexity mentioned earlier. Engineering teams start to lose velocity or throughput as the monolithic code base begins to become more complicated to understand.

Summarizing the Application *x*-Axis

The *x*-axis of the AKF Application Scale Cube represents the cloning of an application or service such that work can easily be distributed across instances with absolutely no bias.

In general, *x*-axis implementations are easy to conceptualize and typically can be implemented at relatively low cost. They are the most cost-effective way of scaling transaction growth. They can be easily cloned within your production environment from existing systems or "jumpstarted" from "golden master" copies of systems. They do not tend to increase the complexity of your operations or production environment.

On the downside, *x*-axis implementations are limited by the growth of a monolithic application, which tends to slow down the processing of transactions. They do not scale well with increases in data or application size. They do not allow engineering teams to scale well, as the code base is monolithic and can become comparatively complex.

The *y*-Axis of the AKF Application Scale Cube

The *y*-axis of the scale cube represents a separation of work responsibility within your application. We most frequently think of it in terms of functions, methods, or services within an application. The *y*-axis split addresses the monolithic nature of an application by separating that application into parallel or pipelined processing flows. A pure *x*-axis split would have *N* instances of the exact same application performing exactly the same work on each instance. Each of the *N* instances would receive $1/N$ th of the work. In a *y*-axis split, we might take a single monolithic application and split it up into *Y* distinct services, such as login, logout, read profile, update profile, search profiles, browse profiles, checkout, display similar items, and so on.

Not surprisingly, then, *y*-axis splits are more complicated to implement than *x*-axis splits. At a very high level, it is often possible to implement a *y*-axis split in production without actually splitting the code base itself, although the benefits derived from this approach are limited. You can do this by cloning a monolithic application and deploying it on multiple physical or virtual servers.

As an example, let's assume that you want to have four unique *y*-axis split servers, each serving one-fourth of the total number of functions within your site. One server might serve login and logout functionality, another read and update profile functionality, another "contact individual" and "receive contacts," and yet another all of the other functions of your platform. You may assign a unique URL or URI to each of these servers, such as login.akfpartners.com and contacts.akfpartners.com, and ensure that any of the functions within the appropriate grouping always get directed to the server (or pool of servers) in question. This is a good first approach to performing a split and helps work out the operational kinks associated with splitting applications. Unfortunately, it doesn't give you all of the benefits of a full *y*-axis split made within the code base itself.

Most commonly, *y*-axis splits are implemented to address the issues associated with a code base and data set that have grown significantly in complexity or size. They also help scale transaction volume, as in performing the splits you must add virtual or physical servers. To get the most benefit from a *y*-axis split, the code base itself needs to be split up from a monolithic structure to a series of individual services that constitute the entire platform.

Operationally, *y*-axis splits help reduce the time necessary to process any given transaction as the data and instruction sets that are being executed or searched are smaller. Architecturally, *y*-axis splits allow you to grow beyond the limitations that systems place on the absolute size of software or data. In addition, *y*-axis splits aid in fault isolation as identified within Chapter 21, Creating Fault-Isolative Architectural Structures; a failure of a given service will not bring down all of the functionality of your platform.

From an engineering perspective, *y*-axis splits allow you to grow your organization more easily by focusing teams on specific services or functions within your product. For example, one team might be dedicated to the search and browse functionality, another team to the development of an advertising platform, yet another team to account functionality, and so on. New engineers get up to speed faster because they are dedicated to a specific section of functionality within your system. More experienced engineers will become experts at a given system and as a result can produce functionality within that system faster. The data elements upon which any *y*-axis split works will likely be a subset of the total data on the site; as such, engineers will better understand the data with which they are working and be more likely to make better choices in creating data models.

Of course, *y*-axis splits also have drawbacks. They tend to be more costly to implement in terms of engineering time than *x*-axis splits because engineers need to rewrite—or at the very least disaggregate—services from the monolithic application. In addition, the operations and infrastructure teams now need to support more than one configuration of server. This, in turn, might mean that the operations environment includes more than one class or size of server so as to utilize the most cost-efficient system for each type of transaction. When caching is involved, data might be cached differently in different systems, although we highly recommend that a standard approach to caching be shared across all of the splits. URL/URI structures will grow, and when referencing other services, engineers will need to understand the current structure and layout of the site or platform to address each of the services.

Summarizing the Application *y*-Axis

The *y*-axis of the AKF Application Scale Cube represents separation of work by service or function within the application. Splits of the *y*-axis are meant to address the issues associated with growth and complexity in the code base and data sets. The intent is to create fault isolation as well as to reduce response times for *y*-axis split transactions.

The *y*-axis splits can scale transactions, data sizes, and code base sizes. They are most effective in scaling the size and complexity of your code base. They tend to cost a bit more than *x*-axis splits because the engineering team needs to rewrite services or, at the very least, disaggregate them from the original monolithic application.

The *z*-Axis of the AKF Application Scale Cube

The *z*-axis of the Application Scale Cube is a split based on a value that is “looked up” or determined at the time of the transaction; most often, this split is based on the requestor or customer of the transaction. The requestor and the customer may be completely different people. The requestor, as the name implies, is the person submitting a request to the product or platform, whereas the customer is the person who will receive the response or benefit of the request. Note that these are the most common implementations of the *z*-axis, but not the only possible implementation. For the *z*-axis split to be valuable, it must help partition not only transactions, but also the data necessary to operate on those transactions. A *y*-axis split helps us scale by reducing instructions and data necessary to perform a service; a *z*-axis split attempts to do the same thing through non-service-oriented segmentation.

To perform a *z*-axis split, we look for similarities among groups of transactions across several services. If a *z*-axis split is performed in isolation from the *x*- and *y*-axes, each split will be a single monolithic instance of a product. The most common implementation of a *z*-axis split involves segmenting the identified solution N ways, where each of the N implementations is the same code base. In some cases, however, deployments may contain a superset of capabilities. As an example, consider the case of the “freemium” business model, where a subset of services is free and perhaps supported by advertising, while a larger set of services requires a license fee for usage. Paying customers may be sent to a separate server or set of servers with a broader set of capabilities.

How do we get benefits with a *z*-axis split if we have the same monolithic code base across all instances? The answer lies in the activities of the individuals interacting with those servers and the data necessary to complete those transactions. Many applications and sites today rely on such extensive caching that it becomes nearly impossible to cache all the necessary data for all potential transactions. Just as the *y*-axis split helped us cache some of this data for unique services, so does the *z*-axis split help us cache data for specific groups or classes of transactions biased by user characteristics.

The benefits of a *z*-axis split are an increase in fault isolation, transactional scalability, and cache-ability of objects necessary to complete our transactions. You might offer different levels of service to different customers, though to do so you might need to layer a *y*-axis split within a *z*-axis split. The end results we would expect from these splits are higher availability, greater scalability, and faster transaction processing times.

The *z*-axis, however, does not help us as much with code complexity, nor does it improve time to market. Furthermore, we add some operational complexity to

our production environment; we now need to monitor several different systems with similar code bases performing similar functions for different clients. Configuration files may differ as a result, and systems may not be easily moved once configured depending on your implementation.

Because we are leveraging characteristics unique to a group of transactions, we can also improve our disaster recovery plans by geographically dispersing our services. We can, for instance, locate services closer to the clients using or requesting those services. With a sales lead system, we could put several small companies in one geographic area on a server close to those companies; for a large company with several sales offices, we might split that company into several sales office systems spread across the company and placed near the offices in question.

Another example of a *z*-axis split would be separating products by SKU (stock keeping unit) or product number. These are typically numeric IDs such as “0194532” or alphanumeric labels such as “SVN-JDF-045.” For search transactions on a typical ecommerce site, for instance, we could divide our search transactions into three groups, each serviced by a different search engine. The three groups might consist of SKUs starting with numbers 0–3, 4–6, and 7–9, respectively. Alternatively, we could separate our searches by product category—for example, cookware in one group, books in another group, and jewelry in a third search group.

The *z*-axis also helps to reduce risk. Whether within a continuous or phased delivery model, deployment of new solutions to a segment of users limits the impacts of new changes on the entire population of users.

Summarizing the Application *z*-Axis

The *z*-axis of the AKF Application Scale Cube represents separation of work based on attributes that are looked up or determined at the time of the transaction. Most often, these are implemented as splits by requestor, customer, or client.

Of the three types of splits, *z*-axis splits tend to be the most costly to implement. Although software does not necessarily need to be disaggregated into services, it does need to be written such that unique pods can be implemented. Very often, a lookup service or deterministic algorithm will need to be written for these types of splits.

The *z*-axis splits aid in scaling transaction growth, scaling instruction sets, and decreasing processing time (the last by limiting the data necessary to perform any transaction). The *z*-axis is most effective at scaling growth in customers or clients. It can aid with disaster recovery efforts, and limit the impact of incidents to only a specific segment of customers.

Putting It All Together

The observant reader has probably figured out that we are about to explain why you need multiple axes of scale and not just single-axis splits. We will work backward through the axes and explain the problems with implementing them in isolation.

A *z*-axis only implementation has several problems when applied in isolation. To better understand these problems, let's assume the previous case where you make N splits of your customer base in a sales lead tracking system. Because we are implementing only the *z*-axis here, each instance is a single virtual or physical server. If it fails for hardware or software reasons, the services for that customer or set of customers become completely unavailable. That availability problem alone is reason enough for us to implement an *x*-axis split for each of our *z*-axis splits. If we split our customer base N ways along the *z*-axis, with each of the N splits having at least $1/N$ th of our customers initially, we would put at least two “cloned” or *x*-axis servers in each of the N splits. This ensures that if a server fails, we can still service the customers in that pod. Reference Figure 23.2 as we discuss this implementation further.

It is likely more costly for us to perform continued customer-oriented splits to scale our transactions than it is to simply add servers within one of our customer-oriented splits. Operationally, it should be relatively simple to add a cloned system to our service for any given customer, assuming that we do not have a great deal of state enabled. Therefore, in an effort to reduce the overall cost of scale, we will probably implement a *z*-axis split with an *x*-axis split in each *z*-axis segment. We can also now scale horizontally via *x*-axis replication within each of our N number of *z*-axis pods. If a customer grows significantly in terms of the volume of its transactions, we can perform a cost-effective *x*-axis split (the addition of more cloned servers or virtual machines) within that customer's pod.

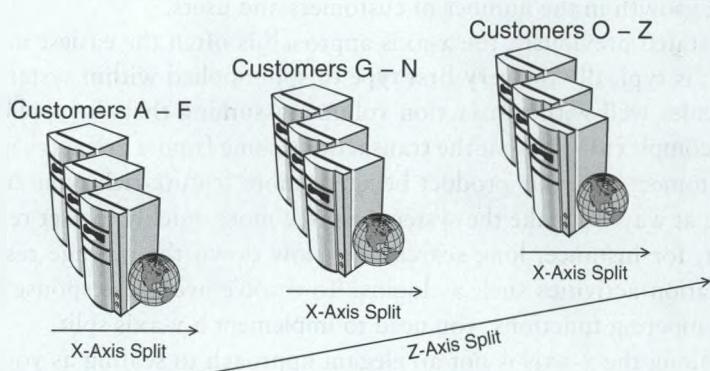


Figure 23.2 Example: *z*- and *x*-Axes Split

Of course, as we have previously mentioned, the *z*-axis split really does not help us with code complexity. As our functionality increases and the size of our application grows, performing *x*- and *z*-axis splits alone will not allow us to focus and gain experience on specific features or services. Our time to market will likely suffer as the monolithic application grows in complexity. We may also find that the large monolithic *z*- and *x*-axis splits will not help us enough given all of the functions that need cached data. A single, very active customer, focused on many of its own clients within our application, may find that a monolithic application is just too slow. This scenario would force us to focus on *y*-axis splits as well.

The *y*-axis split has its own set of problems when implemented in isolation. The first is similar to the problem of the *x*-axis split, in that a single server focused on a subset of functionality results in the functionality being unavailable when the server fails. As with the *z*-axis split, we will want to increase our availability by adding another cloned or *x*-axis server for each of our functions. We can save money by adding servers in an *x*-axis fashion for each of our *y*-axis splits versus continuing to split along the *y*-axis. Rather than modifying the code and further deconstructing it, we simply add servers into each of our *y*-axis splits and bypass the cost of further code modification.

The *y*-axis split also does not scale as well with customer growth as the *z*-axis split does. The *y*-axis splits focus more on the cache-ability of similar functions and work well when we have an application growing in size and complexity. Imagine, however, that you have decided to perform a *y*-axis split of your login functionality and that many of your client logins happen between 6 a.m. and 9 a.m. Pacific Time. Assuming that you need to cache data to allow for efficient logins, you will likely find that you need to perform a *z*-axis split of the login process to gain a higher cache hit ratio. As stated earlier, *y*-axis splits help most when you face a scenario of growth in the application and functionality, *x*-axis splits are most cost-effective when you must deal with transaction growth, and *z*-axis splits aid most when your organization is experiencing growth in the number of customers and users.

As we've stated previously, the *x*-axis approach is often the easiest to implement and, as such, is typically the very first type of split applied within systems or applications. It scales well with transaction volume, assuming that the application does not grow in complexity and that the transactions come from a defined base of slowly growing customers. As your product becomes more feature rich, you are forced to start looking at ways to make the system respond more quickly to user requests. You do not want, for instance, long searches to slow down the average response time of short-duration activities such as logins. To resolve average response time issues caused by competing functions, you need to implement a *y*-axis split.

Splitting along the *x*-axis is not an elegant approach to scaling as your customer base grows. As the number of your customers increases and as the data elements

necessary to support them within an application increases, you need to find ways to segment these data elements to allow for maximum cost-effective scale, such as with y- or z-axis splits.

AKF Application Scale Cube Summary

Here is a summary of the three axes of scale:

- The x-axis represents the distribution of the same work or mirroring of an application across multiple entities. It is useful for scaling transaction volume cost-effectively, but does not scale well with data volume growth.
- The y-axis represents the distribution and separation of work responsibilities by "verb" or action across multiple entities. The y-axis can improve development time, as services are now implemented separately. It also helps with transaction growth and fault isolation. It helps to scale data specific to features and functions, but does not greatly benefit an organization that is experiencing customer data growth.
- The z-axis represents distribution and segmentation of work by customer, customer need, location, or value. It can create fault isolation and scale along customer boundaries. It does not aid in the scenario of growth of data specific to features or functions, nor does it aid in reducing time to market.

Hence, x-axis splits are mirror images of functions, y-axis splits separate applications based on the work performed, and z-axis splits separate work by customer, location, or some value-specific identifier (e.g., a hash or modulus).

Practical Use of the Application Cube

If you know anything about airline reservations, you most likely learned about it in a systems class that discussed the SABRE (Semi-automated Business Research Environment) reservation system. American Airlines implemented the SABRE system to automate reservations. IBM developed SABRE in the late 1950s and ran it on two IBM 7090 mainframes. This type of mainframe system was the prototypical monolithic system that relied on very large compute infrastructure to process high volumes of transactions and maintain a very high availability with built-in hardware redundancy. Fast-forward to today, and the airline reservation and pricing systems look very different.

Today's airline reservation systems have to handle incredibly high volumes of transactions. Web interfaces have allowed consumers to shop rapidly for multiple

connection paths, travel times, and prices. The ratio known as “look-to-book” (how many flights a consumer looks at before booking one) is, on average, 100 to 1. Instead of relying on large compute platforms such as mainframes, some airlines have implemented software that uses all three axes of scale to provide super processing capability and high availability. One such software system is produced by PROS Holdings, Inc. (NYSE: PRO). PROS is a big data software company that helps its customers use big data to sell more seats effectively.

Before we explain how the PROS system is implemented, we need to delve into the very complex and sophisticated world of airline reservations and pricing. Our discussion here is by no means a thorough or exact explanation, but rather a simplification of the process to help you understand how the PROS system works.

Airline pricing is determined by a combination of available inventory and an origin/destination (O/D) model. The O/D model is used to understand air travelers’ true origins and destinations for any specific airport. For example, on a flight from Chicago O’Hare International Airport (ORD) to Los Angeles International Airport (LAX), there will be numerous passengers with many different origins and destinations. One passenger may only be traveling directly from Chicago to Los Angeles, another might be connecting in Los Angeles to Honolulu (HNL), while a third may have started in Newark, New Jersey (EWR), and is connecting on to San Francisco (SFO). In this case, the ORD → LAX flight serves the demand for at least three different O/Ds: ORD–LAX, ORD–HNL, and EWR–SFO. With an average seating capacity exceeding 150 passengers, there may be as many as 150 or more O/Ds. Measuring only the volume of passengers traveling the ORD–LAX route would overstate this demand while not reflecting the demand for the other routes. An O/D model estimates the true travel volume by airport pair based on the passengers’ entire journey and allows airlines to better understand and price flights for their origin and destination markets.

One of the PROS software systems provides this type of O/D model for airlines. Another PROS system uses this O/D model as input, along with an airline’s available inventory, to provide real-time dynamic pricing (RTDP). RTDP systems allow airlines to provide prices based on real-time demand and inventory to customers. This is not done directly; rather, customers make requests and receive responses through global distribution systems (GDS). These GDS are used by online travel agents, including an airline’s own Web site, to aggregate pricing data. Examples of GDS include Sabre (which owns and powers Travelocity), Amadeus, and Travelport.

Now that we know a little bit about airline pricing and reservations, we can look at how PROS engineers architected the PROS product for high availability and scalability. Figure 23.3 shows a typical implementation.

As shown in Figure 23.3, the O/D model is provided by a service completely separated from the real-time dynamic pricing service and distributed asynchronously by

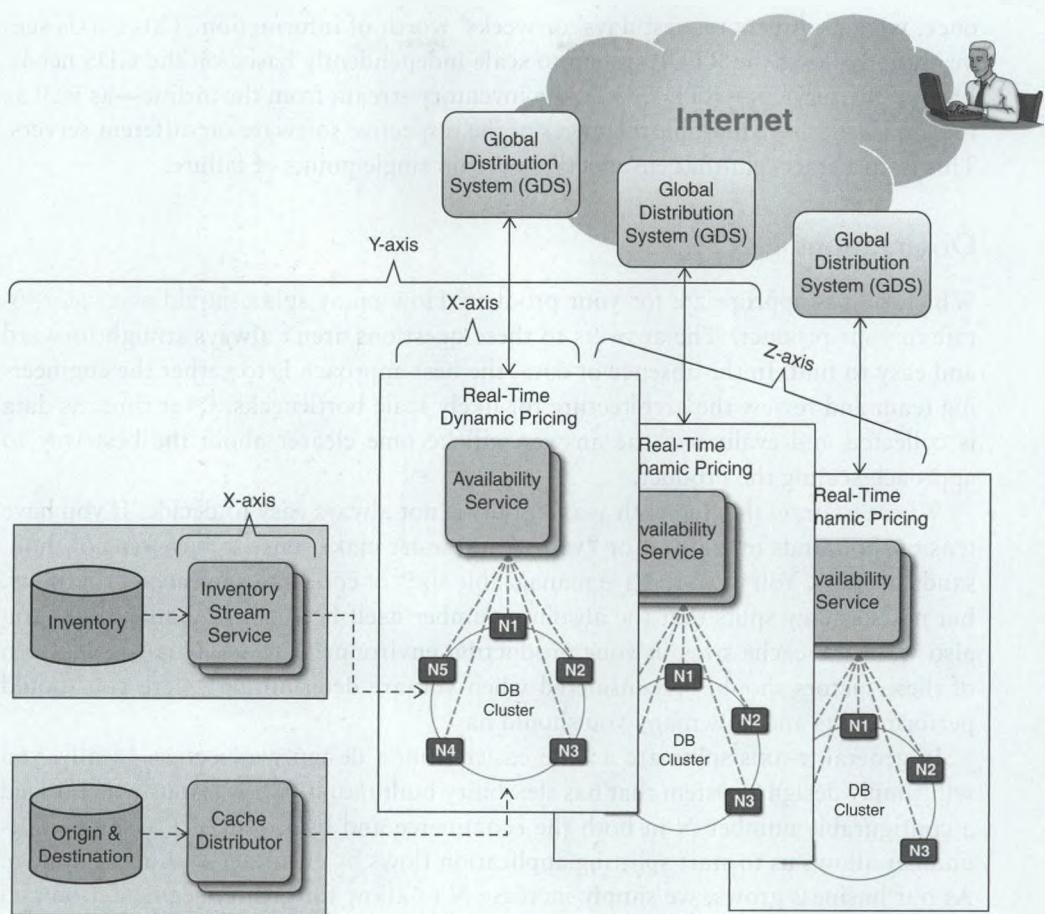


Figure 23.3 PROS Implementation

the cache distributor. This is a y-axis separation that provides fault isolation. Should the O/D model service fail, the RTDP service can continue to provide pricing, albeit with a slightly out-of-date demand model.

In our depiction of the system, there are three implementations of the RTDP service, each providing dynamic pricing to a different GDS. Thus each GDS and RTDP pair together form an isolated “swim lane” of functionality. This again allows for fault isolation. While highly unlikely, it is possible that a GDS will make such a high volume of requests that it will slow down the RTDP service. Should this occur, segmentation of the GDS and pairing of a GDS with a single RTDP keeps other GDS instances from being affected. Additionally, the volumes of pricing requests vary greatly depending on the GDS: Some provide several months’ worth of options at

once, whereas others request days' or weeks' worth of information. This *z*-axis segmentation allows the RTDP system to scale independently based on the GDS needs. Each of the services—RTDP, O/D, the inventory stream from the airline—as well as the DB cluster have multiple instances of the respective software on different servers. This is an *x*-axis split that ensures there are no single points of failure.

Observations

Which split is appropriate for your product? How many splits should you incorporate in your product? The answers to these questions aren't always straightforward and easy to find. In the absence of data, the best approach is to gather the engineering team and review the architecture for likely scale bottlenecks. Over time, as data is collected and evaluated, the answer will become clearer about the best way to approach scaling the product.

Where to draw the line with *y*-axis splits is not always easy to decide. If you have tens of thousands of features or “verbs,” it doesn’t make sense to have tens of thousands of splits. You want to have manageable sizes of code bases in each of the splits, but not so many splits that the absolute number itself becomes unmanageable. You also want the cache sizes in your production environment to be manageable. Both of these factors should be considered when you are determining where you should perform splits and how many you should have.

In general, *z*-axis splits are a little easier from a design perspective. Ideally, you will simply design a system that has flexibility built into it. We previously mentioned a configurable number *N* in both the ecommerce and back-office IT systems. This number allows us to start splitting application flows by customer within the system. As our business grows, we simply increase *N* to allow for greater segmentation and to help smooth the load across our production systems. Of course, potentially some work must be done in data storage (where those customers live), as we will discuss in Chapter 24, but we expect that you can develop tools to help manage that work. With the *y*-axis, unfortunately, it is not so easy to design flexibility into the system.

As always, the *x*-axis is relatively easy to split and handle because it is always just a duplicate of its peers. In all of our previous cases, the *x*-axis was always subordinate to the *y*- and *z*-axes. This is almost always the case when you perform *y*- and *z*-axis splits. To the point, the *x*-axis becomes relevant *within* either a *y*- or *z*-axis split. Sometimes, the *y*- or *z*-axis, as was the case in more than one of the examples, is subordinate to the other, but in nearly all cases, the *x*-axis is subordinate to either the *y*- or *z*-axis whenever the *y*- or *z*-axis or both are employed.

What do you do if and when your business contracts? If you've split to allow for aggressive hyper-growth and the economy presents your business with a downward cycle not largely under your control, what do you do? The *x*-axis splits are easy

to unwind: You simply remove the systems you do not need. If those systems are fully depreciated, you can simply power them off for future use when your business rebounds. The *y*-axis splits might be hosted on a smaller number of systems, potentially leveraging virtual machine software to carve a set of physical servers into multiple servers. The *z*-axis splits should also be capable of being collapsed onto similar systems either through the use of virtual machine software or just by changing the boundaries that indicate which customers reside on which systems.

Conclusion

This chapter discussed the employment of the AKF Scale Cube to applications within a product, service, or platform. We modified the AKF Scale Cube slightly, narrowing the scope and definition of each of the axes so that it became more meaningful to application and systems architecture and the production deployment of applications.

Our *x*-axis still addresses the growth in transactions or work performed by any platform or system. Although the *x*-axis handles growth in transaction volume well, it suffers when application complexity increases significantly (as measured through the growth in functions and features) or when the number of customers with cacheable data needs grows significantly.

The *y*-axis addresses application complexity and growth. As we grow our product to become more feature rich, it requires more resources. Furthermore, transactions that would otherwise complete quickly start to slow down as demand-laden systems mix both fast and slow transactions. In such a scenario, our ability to cache data for all features starts to drop as we run into system constraints. The *y*-axis helps address all of these conditions while simultaneously benefiting our production teams. Engineering teams can focus on smaller portions of our more complex code base. As a result, defect rates decrease, new engineers get up to speed faster, and expert engineers can develop software faster. Because all axes address transaction scale as well, the *y*-axis also benefits us as we grow the transactions against our system, but it is not as easily scaled in this dimension as the *x*-axis.

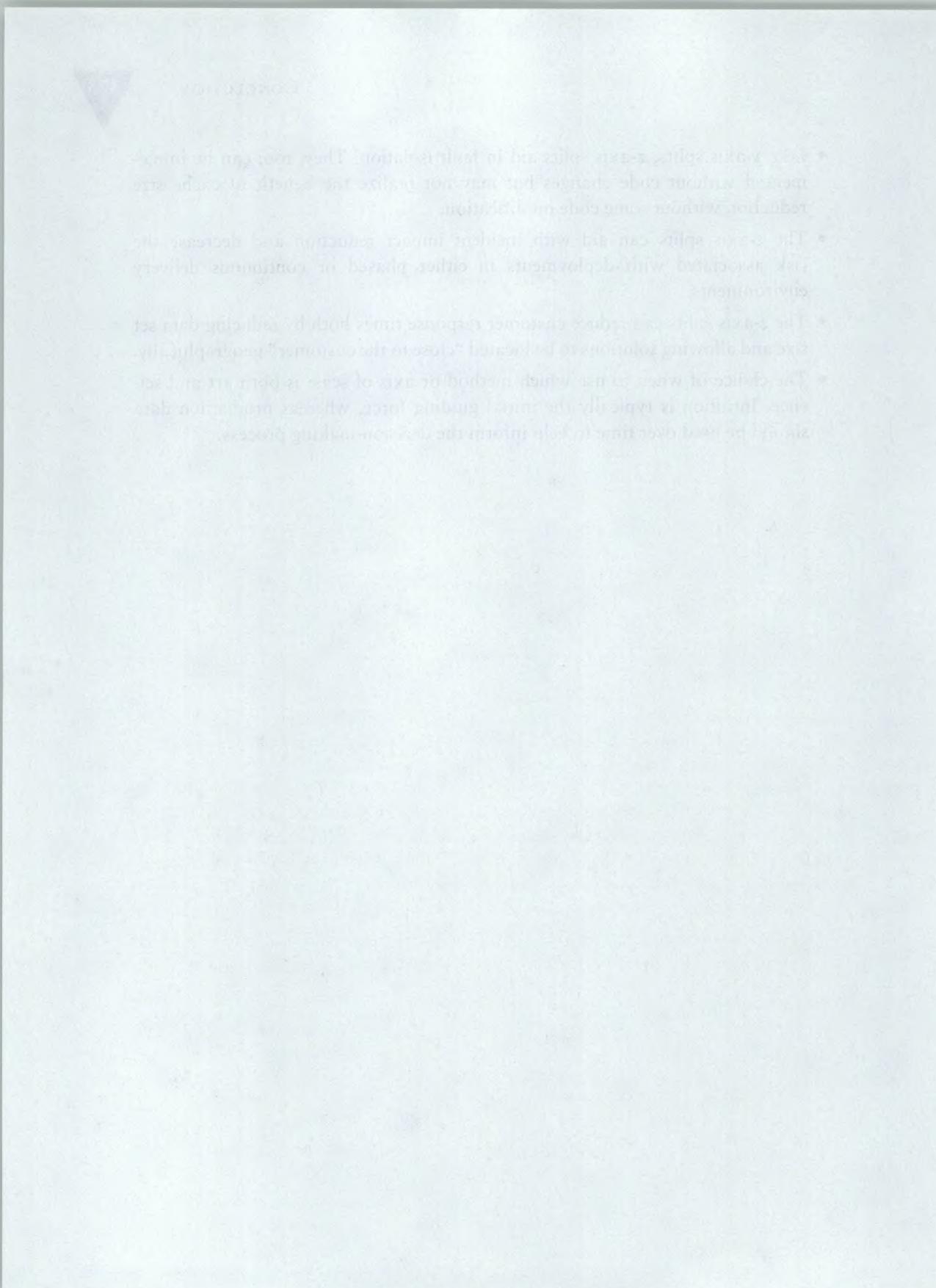
The *z*-axis addresses growth in customer base. As we will see in Chapter 24, it can also help us address growth in other data elements, such as product catalogs. As transactions and customers grow, and potentially as transactions *per* customer grow, we might find ourselves in a position where we might need to address the specific needs of a class of customer. This need might arise solely because each customer has an equal need for some small cache space, but it might be the case that the elements you cache by customer are distinct based on some predefined customer class. Either way, segmenting by requestor, customer, or client helps solve that problem. It also helps us scale along the transaction growth path, albeit not as easily as with the *x*-axis.

As indicated in Chapter 22, not all companies need all three axes of scale to survive. When more than one axis is employed, the x -axis is almost always subordinate to the other axes. You might, for instance, have multiple x -axis splits, each occurring within a y - or z -axis split. When employing y - and z -axis splits together (typically with an x -axis split), either split can become the “primary” means of splitting. If you split first by customer, you can still make y -axis functionality implementations within each of your z -axis splits. These would be clones of each other such that the login service in z -axis customer split 1 looks exactly like the login service for z -axis customer split N . The same is true for a y -axis primary split: The z -axis implementations within each functionality split would be similar or clones of each other.

Key Points

- The x -axis application splits scale linearly with transaction growth. They do not help with the growth in code complexity, customers, or data, however. The x -axis splits are “clones” of each other.
- The x -axis tends to be the least costly to implement, but suffers from constraints in instruction size and data set size.
- The y -axis application splits help scale code complexity as well as transaction growth. They are mostly meant for code scale, as they are not as efficient as x -axis splits for handling transaction growth.
- The y -axis application splits also aid in reducing cache sizes where cache sizes scale with function growth.
- In general, y -axis splits tend to be more costly to implement than x -axis splits as a result of the more extensive engineering time needed to separate monolithic code bases.
- The y -axis splits aid in fault isolation.
- Although y -axis splits can be performed without code modification, you might not get the benefit of cache size reduction and you will not get the benefit of decreasing code complexity.
- The y -axis splits can help scale organizations by reducing monolithic code complexity.
- The z -axis application splits help scale customer growth, some elements of data growth (as we will see in Chapter 24), and transaction growth.
- The z -axis application splits can help reduce cache sizes where caches scale in relation to the growth in users or other data elements.

- Like y -axis splits, z -axis splits aid in fault isolation. They, too, can be implemented without code changes but may not realize the benefit of cache size reduction without some code modification.
- The z -axis splits can aid with incident impact reduction and decrease the risk associated with deployments in either phased or continuous delivery environments.
- The z -axis splits can reduce customer response times both by reducing data set size and allowing solutions to be located “close to the customer” geographically.
- The choice of when to use which method or axis of scale is both art and science. Intuition is typically the initial guiding force, whereas production data should be used over time to help inform the decision-making process.



Chapter 24

Splitting Databases for Scale

So in war, the way is to avoid what is strong and to strike at what is weak.

—Sun Tzu

This chapter focuses on the AKF Scale Cube’s application to databases or, more broadly, “persistence engines”—those solutions that store data (e.g., relational databases, multi-initiated storage solutions, NoSQL solutions). Armed with the information in Chapters 21 through 24, you should be able to create a fault-isolative architecture capable of nearly infinite scale, thereby increasing customer satisfaction and shareholder returns.

Applying the AKF Scale Cube to Databases

The *x*-axis, as you will recall from Chapter 22, focuses on the cloning of services and data without bias. Each *x*-axis implementation requires the duplication of an entire data set. To more directly apply the *x*-axis to persistence engines, here we rename it the “horizontal duplication/cloning of data.” (As in the previous chapter regarding application splits, the name change doesn’t deviate from the original intent of the *x*-axis.)

The *y*-axis was described as a separation of work responsibility based on either the type of data, the work performed for a transaction, or a combination of both. When applied to data, *y*-axis splits are biased either toward the type of data split or the type of work performed on that data. This latter definition is a services-oriented split similar to those discussed in Chapter 23.

The *z*-axis of our database cube continues to have a customer or requestor bias. When applied to data, the *z*-axis very often requires the implementation of a lookup service. Users are routed by the lookup service to their appropriate split within a service or database. New users’ data can be assigned to a split automatically by a modulus function or manually based upon resources. (Stateless applications—that

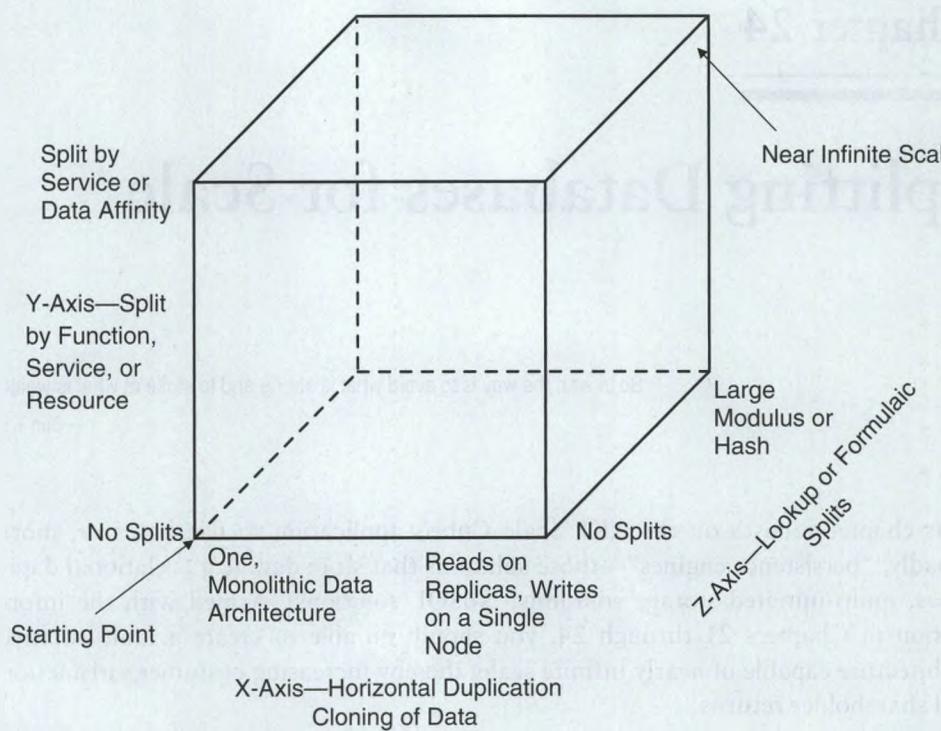


Figure 24.1 AKF Database Scale Cube

is, those with state limited to a session's duration—can choose the appropriate routing based on geo-location or through an algorithm such as a hash, modulus, or other deterministic means.)

The new AKF Scale Cube for databases now looks like Figure 24.1.

The *x*-Axis of the AKF Database Scale Cube

The *x*-axis of the AKF Database Scale Cube represents cloning of data with absolutely no bias. This means for a data tier scaled along the *x*-axis alone, each of N databases will have exactly the same data as the $N - 1$ systems. The databases will be “eventually consistent,” meaning that replication technology will ensure that a database’s state will be replicated exactly on all the other databases after a short time interval. There is no bias to the transactions performed on this data, or to the customer or any other data element; the customer metadata and account information exist alongside the product catalog data, inventory information, contact information, and so on. Therefore a request for data can be served from any of the N databases or storage implementations.

As with the application cube, the *x*-axis approach is simple to implement in most cases. Typically, you will implement some sort of replication system that allows for the near-real-time replication of data within the database or storage systems across some scalable and configurable number N replicated databases or persistent storage systems. Your database likely has a native replication capability built into it, but many third-party tools are also available that enable you to perform the replication.

Replication Delay Concerns

Many companies with which we work initially display some level of concern when we bring up the topic of *replication delay*. The most common concern is the perceived need to be able to immediately access the most current data element for any write. In the most extreme case, we've had clients immediately read a piece of datum from a database simply to validate it was correctly written.

In most cases, we are able to identify a large portion of our client's data that needs only to be current within a single-digit number of seconds. Native database and third-party replication tools within a single data center—even under high volume—can typically keep replicated copies of databases in synch with a master (and sometimes a second master) in less than 5 seconds. Geographically distant data centers can often be synchronized in less than 10 seconds.

We recommend asking the following questions to determine if replicated data is sufficient:

- *How often is this particular data element updated?* If the ratio of views to updates is high, replication delay is probably acceptable. If updates are frequent and views are infrequent, there is little benefit in replication.
- *Will the element that is read be used in a calculation for a future write?* If so, replication delays may be unacceptable.
- *What is the difference in value for decision-making purposes?* If, for instance, the newest update changes a value insignificantly, will that really make a difference in a resulting decision on the part of the person viewing the data?

When considering database replication, look first for native database functionality. It almost never makes sense to build replication functionality yourself, and given that the majority of databases have built-in replication, there is seldom a need to purchase third-party software.

Writes typically happen to a single node within the replicated *x*-axis data tier. By writing to a single node, we reduce the read and write conflicts across all nodes and force a single node to do the work of ensuring the ACID (atomicity, consistency, isolation, and durability) properties of the database. Additionally, this approach ensures that the storage subsystems can be optimized for writes or reads only. Many

times, the “write” copy of the storage tier is used only to write, but sometimes a small number of reads is scheduled to that node if the time sensitivity of the read in question will not allow for the minor delay inherent in replication.

We often teach that distributed object caches and other database-related caches—at least those intended to reduce load on a database—can be examples of *x*-axis splits. Some might argue that if the data is represented in a format that is intended to be more easily consumed by services, it is an example of a *y*-axis split. For a broader discussion of caches, see Chapter 25, Caching for Performance and Scale.

Readers familiar with NoSQL implementations such as MongoDB, Cassandra, and HBase will recognize that they often have multiple copies of replicated data. Often the number of copies of data is configurable within the solution, and many solutions allow you to have “local” and “remote” copies of data. This separation between local and remote allows you to maintain primary copies within one implementation or site and hold other copies in one or multiple other instances for the purposes of higher availability and disaster recovery. Many times clients argue these distributed systems meet the definition and criteria for *x*-axis scaling.

From the perspective of scale, these clients are right: The replication does allow for the distribution of reads across multiple nodes, hence engendering higher levels of transaction throughput. But these are physical separations—not logical separations—of the data. The difference is that one instance, or solution, of the coordination software is being used, which in itself is a scalability concern. Furthermore, from a fault isolation perspective, a NoSQL logical instance can still fail, which may cause widespread outages. As such, we encourage our clients to think about implementing multiple instances of NoSQL solutions to scale both logically (instances) and physically (nodes).

Brewer's Theorem

The astute reader may have recognized that we are “talking around” a concern raised by Eric Brewer of the University of California–Berkeley in the fall of 1998. Brewer posited that it is impossible to guarantee consistency, availability, and partition tolerance within a distributed computer system. *Consistency* refers to all nodes having exactly the same data at the same time. *Availability* refers to each transaction or request getting a success or failure response. *Partition tolerance* means the system continues to operate correctly (is available) when any node or part of the system dies. Brewer's theorem (also known as the CAP theorem—for consistency, availability and partition tolerance) is often cited as a reason for accepting “eventual consistency” as a design limitation of multinode databases. By relaxing the consistency constraint, high availability and partition tolerance become achievable.

An *x*-axis data split has several benefits and drawbacks. Consistent with Chapters 22 and 23, this split is easy to envision and implement. Many databases have native replication technologies that allow for “write and read only” copies or “master and slave” copies of a database. These native replication engines usually support multiple read or “slave” copies of the database. Another *x*-axis implementation is clustering, a capability found in most open source and licensed relational database management systems. By “clustering,” we mean two or more physically separated databases appear to the application as a single instance.

Should a storage system without a database be the target of this technique, many logical and physical replication systems can be applied in both open source and third-party supported systems. This approach allows for linear scale with transactions, although most replication processes place limits on the number of targets or read-only nodes allowed. While this approach supports linear transaction growth, it does not address data growth, the impact of that data growth on request processing time, or the impact of that data growth on addressable storage within any given storage subsystem.

Because *x*-axis splits are easy to envision and implement, they are a good first choice to scale any system when the number of transactions is the primary driver of growth and the size of data (and associated growth) is not a concern. The engineering effort required to support these splits is relatively low. However, the capital needs can be large, as you will need to purchase or partition additional database servers or nodes. Implementation time is generally relatively small, although you will need time to set up and validate the replication.

The *x*-axis splits also allow us to easily scale our data with the number of inbound transactions or requests. As request growth increases, we simply add more read nodes. Furthermore, capacity planning is easy because each of our nodes, if served from similar hardware, can handle a similar number of requests. However, there is often a limit to the number of systems that can be employed, driving us to other means of scale as transaction growth continues to increase. Sometimes we cannot even achieve the vendor- or system-supported limit due to the increase in replication time delays because additional read-only nodes are deployed. Usually, each node has some small impact on the amount of time that it takes to replicate data from the write node. In some implementations, this impact may not manifest itself as a delay to all nodes, but instead introduce a consistently unacceptable delay to a single node within the cluster as we start to reach our maximum target. Therefore, we cannot simply rely on the *x*-axis for scale even in systems characterized by relatively low data growth if transaction growth accelerates over time.

One final benefit of the *x*-axis is that the team managing the infrastructure of your platform does not need to worry about a vast number of uniquely configured

schemas or storage systems. Every system performing an *x*-axis split is exactly equivalent to every other system performing the same split, with the exception of a single system dedicated to “writing.” Configuration management of all nodes is relatively easy to perform, and new service implementation is as easy as replicating the data within an existing system. Your application, when architected to read from a “read service” and write to a “write service,” should scale without further involvement from your engineering team. Ideally, the multiple read systems will be addressed through a third-party load balancing system rather than by having your team write even a small routine to “load balance” or evenly apportion the reads.

Two primary drivers will move us away from scaling along the *x*-axis alone. The first was discussed while addressing the limitations of existing replication technology. The second driver is that *x*-axis scale techniques do not address the scale limits inherent in an increase in the size or amount of data. Similar to the caching concern described in Chapter 23, when you increase the size of data within a database, the response times of that database increase. While indices help significantly reduce this increase in response time, a 10x increase in table sizes can still result in a 1.5x increase in response time. This increase in response time may ultimately drive you to other splits.

Another drawback of *x*-axis replication is the cost of replicating large amounts of data. Typically, *x*-axis implementations are complete clones of a primary database, which in turn means that we might be moving lots of data that is seldom read relative to data within the same replication set that is read frequently. One solution to this concern is to select for replication only the data that has a high volume of reads associated with it. Many of the replication technologies for databases allow such a selection to occur on a table-by-table basis, but few (if any) allow columns within a table to be selected.

Other drawbacks of *x*-axis replication include data currency and consistency concerns, the need for increased experience with replication, and the reliance on third parties to scale. We addressed data currency in the sidebar “Replication Delay Concerns.” To rely on database replication, the team must have the skill sets needed to implement, monitor, and maintain the replication solution. The native or third-party product that you choose to perform your replication typically manages consistency. This functionality seldom creates a problem even in the products having the highest request volume. More often, we see that the consistency manager stops replication due to some concern, which in turn creates a larger data currency issue. These issues are usually resolved in a short amount of time. If you are using a third-party replication tool, as long as your solution does not depend on the features offered by a particular vendor, you can always switch an underperforming partner out for another commodity solution.

Summarizing the Database x-Axis

The *x*-axis of the AKF Database Scale Cube represents the replication of data such that work can easily be distributed across nodes with absolutely no bias.

The *x*-axis implementations tend to be easy to conceptualize and typically can be implemented at relatively low cost. They are the most cost-effective way of scaling transaction growth, although they are usually limited in the number of nodes that can be employed. They can be easily created from a monolithic database or storage system, albeit with an upfront cost in most cases. Furthermore, they do not significantly increase the complexity of your operations or production environment.

On the downside, *x*-axis implementations are limited by the aforementioned replication technology limitations and the size of data that is being replicated. In general, *x*-axis implementations do not scale well with data size and growth.

The *y*-Axis of the AKF Database Scale Cube

The *y*-axis of the AKF Database Scale Cube represents a separation of data by meaning, function, or usage. The *y*-axis split addresses the monolithic nature of the data architecture by partitioning the data into schemas that have distinct meanings and purposes for the applications that access them. In a *y*-axis split, we might split the data into the same “chunks” as we split our application in Chapter 23. These might be exactly the data necessary to perform such separate functions as login, logout, read profile, update profile, search profiles, browse profiles, checkout, display similar items, and so on. Of course, there may be overlap in the data, such as customer-specific data being present in the login/logout functionality as well as the update profile functionality, and we’ll address ways to handle this overlap.

Whereas the splits described previously are delineated by actions, services, or “verbs,” you can also consider splitting by resources or “nouns.” As an example, we might put customer data in one spot, product data in another, user-generated content in a third, and so on. This approach has the benefit of leveraging the affinity that data elements often have with one another and leveraging the talents of the database architects who are familiar with entity relationships within relational databases. The drawback of this approach is that we must either change our approach for splitting applications to also be resource based (that is, “all services that interact with customer data in one application”) or suffer the consequences of not having a swim lane-based and fault-isolated product. Furthermore, if we split our data along

resource meaningful boundaries and split our application along service meaningful boundaries, then we will almost certainly encounter the situation in which services talk to several resources. As we discussed in Chapter 21, this arrangement will lower product availability. For this reason, you should choose resource-oriented or service-oriented splits for both your application and your data.

Consistent with our past explanations of split complexities in Chapters 22 and 23, *y*-axis data splits are often more complex than *x*-axis data splits. To reduce this complexity, you may decide to first implement the splits within the current persistence engines rather than moving data to separate physical hosts. In a database, this can be implemented by moving tables and data to a different schema or database instance within the same physical hardware. Such a move saves the initial capital expense of purchasing additional equipment, but unfortunately it does not eliminate the engineering cost of changing your code to address different storage implementations or databases. For the physical split, you can, on a temporary basis, use tools that link the separate physical databases together; then, if your engineers missed any tables being moved, they will have an opportunity to fix the code before breaking the application. After you are sure that the application is properly accessing the moved tables, you should remove these links. If you leave them in place, they may cause a chain reaction of degraded performance in some instances.

The *y*-axis splits are most commonly implemented to address the issues associated with a data set that has grown significantly in complexity or size and that is likely to continue to grow. These splits also help scale transaction volumes by moving requests to multiple physical or logical systems, thereby decreasing logical and physical contention for the data. For companies facing extreme or hyper-growth scenarios, we recommend splitting the system into separate physical instances. Maintaining logical splits on shared physical instances can overload shared network, compute, and memory structures.

Operationally, *y*-axis splits reduce transaction times, as requests are smaller and tailored to the service performing the transaction. Conceptually, such splits allow you to better understand vast amounts of data by thematically bundling items, rather than lumping everything into the same “storage” container. In addition, *y*-axis splits also aid in fault isolation as identified within Chapter 21; that is, a failure of a given data element will not bring down the functionality of your entire platform (assuming that you have properly implemented the swim lane concept).

When bundled with similar splits at an application level, *y*-axis splits allow teams to grow easily by focusing them on specific services or functions within the product and the data relevant to those functions and services. As discussed in Chapter 23, you can dedicate one team to searching and browsing, a second team to the development of an advertising platform, a third team to account functionality, and so on. All of the engineering benefits, including “on boarding” and expertise in a subset of the code, are realized when both the data and the services acting on that data are split together.

Of course, *y*-axis splits also have some drawbacks. They are absolutely more costly to implement in terms of engineering time than *x*-axis splits. Not only do services need to be rewritten to address different data storage systems and databases, but the actual data also likely needs to be moved if your product has already launched. As a result, operations and infrastructure teams will need to support more than one schema, which in turn may create a need for more than one class or size of server in the operations environment for reasons of cost-efficiency.

Summarizing the Database *y*-Axis

The *y*-axis of the AKF Database Scale Cube represents separation of data meaning, often by service, resource, or data affinity.

The *y*-axis splits are meant to address the issues associated with growth and greater complexity in data size and their impact on requests or operations on that data. If implemented and architected to be consistent with an application *y*-axis split, this technique can create fault isolation as described in Chapter 21.

The *y*-axis splits can scale with growth in both the number of transactions and the size of data. They tend to cost more than *x*-axis splits, as the engineering team needs to rewrite services and determine how to move data between separate schemas and systems.

The *z*-Axis of the AKF Database Scale Cube

As with the Application Scale Cube, the *z*-axis of the AKF Database Scale Cube consists of splits based on values that are “looked up” or determined at the time of the transaction. This split is most commonly performed by looking up or determining the location of data based on a reference to the customer or requestor. However, it can also be applied to any split of data within a resource or service where that split is done without a bias toward affinity or theme. Examples would be a modulus or hash of product number if that modulus or hash were not indicative of the product type.

To illustrate the *z*-axis, let’s take a look at two different ways to split up information in a product catalog. Dividing jewelry into groups such as watches, rings, bracelets, and necklaces is an example of *y*-axis splits. These splits are performed along boundaries characterized by the jewelry itself. Alternatively, *z*-axis splits of this same product catalog could be performed by using an indiscriminate function such as a modulus of the stock keeping unit (SKU). In this approach, the last digit (or several digits) of the SKU determines where the item resides, rather than attributes of the item itself as in the previously described *y*-axis split based on jewelry type.

We often advise clients to perform a *z*-axis split along customer boundaries, such as the location from which the customer makes a request. These types of *z*-axis splits offer many advantages, including the ability to put micro-sites (one or more instances of the *z*-axis split) close to a customer for the purpose of reducing response times. This tactic increases product availability, as geographically localized events will impact only a portion of the customer base. The choice of customer location, while not completely indiscriminate (as in a hash or modulus), fits our “lookup” criteria and, therefore, is classified as a *z*-axis split.

As with the Application Scale Cube, for the *z*-axis split to be valuable, it must help scale both the transactions and the data upon which those transactions are performed. A *z*-axis split attempts to accomplish the data scalability benefits of a *y*-axis split without a bias toward the action (service) or resource itself. In doing so, it also offers more balanced demand across all of the data than a *y*-axis split in isolation. If you assume that each datum has a relatively equal opportunity to be in high demand, average demand, or low demand, and if you apply a deterministic and unbiased algorithm to store and locate such data, you are statistically likely to observe a relatively equal distribution of demand across all of your databases or storage systems. This is not true for the *y*-axis split, which may cause certain systems to have unique demand spikes based on their data content.

Because we are splitting our data and as a result our transactions across multiple systems, we can achieve a transactional scale similar to that within the *x*-axis. Furthermore, we aren’t inhibited by the replication constraints of the *x*-axis because in a *z*-axis split we are not replicating data. Unfortunately, as with the *y*-axis, our operational complexity increases to some extent, as we now have many unique databases or data storage systems; the schema or setup of these databases is similar, but the data in each is unique. Unlike with the *y*-axis, we aren’t likely to realize any of the benefits of splitting up our architecture in a service- or resource-oriented fashion. Schemas or setups are monolithic in *z*-axis only implementations, although the data is segmented as with a *y*-axis split. Finally, *z*-axis splits are associated with some software costs because the code must recognize that requests are not all equivalent. As with application splits, an algorithm or lookup service is created to determine to which system or pod a request should be sent.

Benefits of a *z*-axis split include increased fault isolation (if done through the application tier as well), increased transactional scalability, increased data scalability, and increased ability to adequately predict demand across multiple databases (as the load will likely be evenly distributed). The outcome of these improvements is higher availability, greater scalability, faster transaction processing times, and a better capacity planning function within our organization.

The *z*-axis, however, requires implementation costs and adds operational complexity to our production environment; specifically, the presence of several different

systems with similar code bases will increase monitoring requirements. Configuration files may differ as a result, and systems may not be easily moved when configured depending on your implementation. Because we are leveraging characteristics unique to a group of transactions, we can also improve our disaster recovery plans by geographically dispersing our services.

Summarizing the Database z-Axis

The z-axis of the AKF Database Scale Cube represents separation of work based on attributes that are looked up or determined at the time of the transaction. Most often, these are implemented as splits by requestor, customer, or client, though they can also be splits within a product catalog and determined by product ID or any other characteristic that is determined and looked up at the time of the request.

Of the three types of splits, z-axis splits are often the most costly implementation. Software needs to be modified to determine where to find, operate on, and store information. Very often, a lookup service or deterministic algorithm will need to be written for these types of splits.

The z-axis splits aid in scaling transaction growth, decreasing processing time (by limiting the data necessary to perform any transaction), and capacity planning (by more evenly distributing demand across systems). The z-axis is most effective at evenly scaling growth in customers, clients, requesters, or other data elements that can be evenly distributed.

Putting It All Together

As you've seen in Chapters 22 and 23, and thus far in Chapter 24, the AKF Scale Cube is a very powerful and flexible tool. Of all the tools we've used in our consulting practice, our clients have found it to be the most useful in figuring out how to scale their systems, their databases, and even their organizations. Because it represents a common framework and language, little energy is wasted in defining what is meant by different approaches. This allows groups to argue over the relative merits of an approach rather than spending time trying to understand how something is being split. Furthermore, teams can easily and quickly start applying the concepts within any of their meetings rather than struggling with the options on how to scale something. As in Chapter 23, in this section we will discuss how the cube can be applied to create near-infinite scalability within your databases and storage systems.

A z-axis only implementation of the AKF Database Scale Cube has several problems when implemented in isolation. Let's assume the previous case where you make N splits of your customer base in a jewelry ecommerce platform based on geography

of the customer. Because we are implementing only the *z*-axis here, each instance is a single virtual or physical database server. If it fails for hardware or software reasons, the services for that customer or set of customers become completely unavailable. That availability problem alone is reason enough for us to implement an *x*-axis split for each of our *z*-axis splits. At the very least, we should have one additional database that we can use in the event that our primary database for any given set of customers fails. The same holds true if the *z*-axis is used to split our product catalog. If we split our customer base or product catalog N ways along the *z*-axis, with each of the N splits having at least $1/N$ th of our customers or product initially, we would put at least one additional “cloned” or *x*-axis server in each of the N *z*-axis splits. With this scheme, if a server fails, we can still service the customers in that pod.

The cost of performing each successive split is greater than just the capital cost of new servers. Each time we perform a split, we need to update our code to recognize where the split information is or, at the very least, update a configuration file giving the new modulus or hash values. Additionally, we need to create programs or scripts to move the data to the expected positions within the newly split database or storage infrastructure. Mixing the *x*- and *z*-axis splits helps reduce these costs. Rather than face the cost of iterative *z*-axis splits, we can use *x*-axis scalability in each of our *z*-axis swim lanes to scale transaction growth. Once we have enough pent-up demand for additional *z*-axis splits or wish to significantly increase our system’s availability, we can create several *z*-axis splits at one time; in doing so, we can reduce the related software cost by “batching” them together in multiple simultaneous splits.

Using *y*-axis splits in conjunction with *z*-axis splits can help us create fault isolation. If we led an architectural split by splitting customers first, we could then create fault-isolative swim lanes by function or resource within each of the *z*-axis splits. For example, we might have product information in each of the *z*-axis splits separate from customer account information and so on.

The *y*-axis split has its own set of problems when implemented in isolation. The first is similar to the problem of the *z*-axis split, in that a single database focused on a subset of functionality results in the functionality being unavailable when the server fails. As with the *z*-axis split, we will want to increase our availability by adding at least another cloned or *x*-axis server for each of our functions. We can also save money by adding servers in an *x*-axis fashion for each of our *y*-axis splits versus continuing to split along the *y*-axis. As with our *z*-axis split, it costs us engineering time to continue to split off functionality or resources, and we likely want to spend as much of that time on new product functionality as possible. Rather than modifying the code and further deconstructing our databases, we simply add replicated databases into each of our *y*-axis splits and bypass the cost of further code modification. Of course, this approach assumes that we’ve already written the code to write to a single database and read from multiple databases.

The *y*-axis split also does not scale as well with customer growth, product growth, or some other data elements as the *z*-axis split. Although *y*-axis splits in databases help us disaggregate data, only a finite number of splits is possible, with the precise number depending on the affinity of the data and your application architecture. Suppose, for example, that you split all product information off from the rest of your data. Your entire product catalog is now separated from everything else within your data architecture. You may be able to perform several *y*-axis splits in this area, similar to those described in the jewelry example of splitting watches from rings, necklaces, and so on. But what happens when the number of rings available grows to a point that it becomes difficult for you to further split them by categories? What if the demand on a subset of rings is such that you need to be careful about which hardware serves this data set? A *z*-axis split can allow the rings to exist across several databases without regard to the type of ring. As previously indicated, the load will likely also be uniformly distributed.

The *x*-axis approach is often the easiest to implement and, as such, is very often the very first type of split within data architectures. It scales well with transactions, but very often places a limit on the number of nodes to which you can scale. As transaction volume and the amount of data that you serve grow, you will need to implement another axis of scale. The *x*-axis is very often the first axis of scale implemented by most companies, but as the product and transaction base grows, it typically becomes subordinate to either the *y*- or *z*-axis.

Ideally, as we indicated in Chapter 12, Establishing Architectural Principles, you should plan for at least two axes of scale even if you implement only a single axis. Planning for a *y*-axis or *z*-axis split in addition to initially implementing an *x*-axis replication is a good approach. If you find yourself in a hyper-growth situation, you will want to plan for all three. In this situation, you should determine a primary implementation (say, a *z*-axis by customer), a secondary implementation (a *y*-axis by functionality), and an *x*-axis for redundancy and transaction growth. Then, apply a fault-isolative swim lane per Chapter 21 and even a “swim lane within a swim lane” concept. You may swim lane your customers in a *z*-axis, and then swim lane each of the functions within each *z*-axis in a *y*-axis fashion. The *x*-axis then exists for redundancy and transaction scale. Voila! Your system is both highly available and highly scalable.

AKF Database Scale Cube Summary

Here is a summary of the three axes of scale:

- The *x*-axis represents the distribution of the same data or mirroring of data across multiple entities. It typically relies on replication and places a limit on how many nodes can be employed.

- The *y*-axis represents the distribution and separation of the meaning of data by service, resource, or data affinity.
- The *z*-axis represents the distribution and segmentation of data by attributes that are looked up or determined at the time of request processing.

Hence, *x*-axis splits are mirror images of data, *y*-axis splits separate data thematically, and *z*-axis splits separate data by a lookup or modulus. Often, *z*-axis splits happen by customer, but they may also happen by product ID or some other value.

Practical Use of the Database Cube

Let's examine the practical use of our database cube. Here we present approaches for a couple of commonly used business models (ecommerce and SaaS) and for one common use case within most products (search).

Ecommerce Implementation

eBay remains, to this day, one of the largest transaction processing sites on the Internet. Its growth dynamics are unique compared to many sites. Because users can list nearly anything they would like as a unique item, data growth occurs at a rate faster than is experienced by many other SKU-based commerce sites. Because eBay does business in multiple countries and has an ever-increasing user base, user data growth is also a concern. Similar to many other sites, eBay experiences significant transaction volumes across a number of functions. All of these factors mean that eBay must be able to scale both data and transactions while providing a highly available service for its customers. Many of the principles we teach, we "learned" (through trial and error) during our time at eBay and PayPal during the first tech boom. It isn't difficult to imagine that the *x*-, *y*- and *z*-axes of scaling exist in varying forms within this ecommerce giant.

Some of the first database splits at eBay were functional in nature. As early as 2000, we began to split off product-related items into their own databases. The very first such split from what was then a monolithic database (one database for absolutely everything on the site—a Sun E10K server running Oracle) was a small category of items known as "weird stuff." This was where folks would list things that simply didn't fit easily into other category structures. People sold all sorts of strange things in this category, including, believe it or not, ear wax. (Don't ask us why!) The split made sense, though: "Weird stuff" was not only a small category, but it also represented a small transaction volume. Risk to the business was therefore low had the split failed.

After the “weird stuff” split was deemed successful, Mike Wilson (eBay’s first CTO) and his architecture team continued to define y -axis splits for what eBay started to call the “Items” databases. Think of these as being equivalent to product- or SKU-based splits that we discussed earlier. The infrastructure team would set up unique physical database hosts, and the architecture and development team would split items into these new hosts. The application server code was updated to understand where an item would exist based on its category hierarchy. For example, jewelry would exist on one host, art on another host, and so on.

The team continued to define y -axis or functional/data-oriented splits throughout the site. The eBay feedback system got its own set of hosts, as did user and account information. The load on the primary monolithic database, called the “Market” database in those days, dropped with each successive split.

But eBay continued to grow and some of these new y -axis systems eventually began to experience their own problems with transaction rates. The “Users” database, for instance, was becoming congested and experiencing problems in terms of load, CPU utilization, and memory contention. Our solution was to split users and create multiple user databases along the z -axis of scale. The team worked to split out users based on a modulus of the user ID and place the new groups on their own databases.

To engender higher levels of transaction scale and to provide for higher levels of availability, many of these y - and z -axes splits also relied on database replication to create x -axis copies of the data. The code was instrumented to understand the difference between transactions that intended to “write” (create, update, or delete within “CRUD” operations) and those that wanted to read (the “R” in “CRUD”). The team employed hardware load balancers to direct reads to any of a number of read instances, while directing writes to the primary or “master” database.

It’s been many years since we left eBay, and several generations of engineers and executives have come and gone since our time at the company. While our experience since eBay has helped us fine-tune the approaches we learned there, and while we would do some things differently, it’s interesting to note that much of the architecture the combined team developed and deployed continues to run today.¹

Search Implementation

Most products in nearly any industry have some sort of search capability. Search is a somewhat unique use case, as it requires the product to both traverse a large (and sometimes growing) set of data and return results quickly regardless of data size or

1. eBay scalability best practices. *Highscalability.com*. <http://www.infoq.com/articles/ebay-scalability-best-practices>.

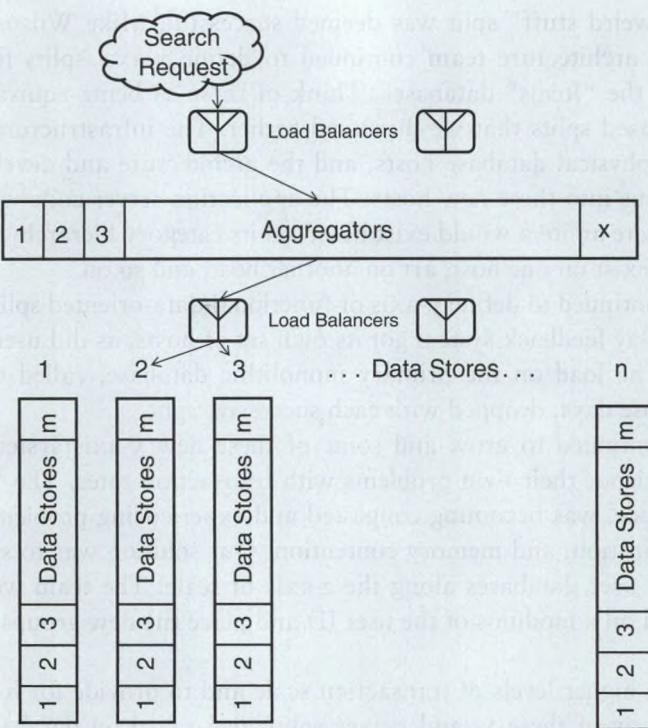


Figure 24.2 Fast Read or Search Subsystem

rates of transactions. While these requirements sound daunting, the solution is actually quite simple once you apply the AKF Scale Cube. One word of caution here: You likely do not need to build search capabilities yourself or replicate the discussion presented in this section. A number of open source and vendor-provided solutions have already created search capabilities similar to those that we describe here. In fact, some very successful products, including Netflix, Apple iTunes, and Wikipedia, use solutions such as Apache Lucene and Solr to implement search functionality within their products.²

For this discussion, refer to Figure 24.2. Splitting search itself off into its own service and supporting data structures represents a y-axis split. Given that one of our requirements is that we want fast response time regardless of data size, we will borrow a technique from MapReduce (see Chapter 27) and shard or partition our entire data set N ways. N ideally is a modifiable number, allowing us to increase the number of partitions over time as our data set grows. Expect a 100-fold increase

2. Next-generation search and analytics with Apache Lucene and Solr 4. IBM Developerworks. <http://www.ibm.com/developerworks/java/library/j-solr-lucene/index.html?ca=drs->

in data over time from where you are today? Set N to 100, and response times for any shard will be roughly equivalent to today's response time. Aggregators then make requests to each shard of data (1 through N) pursuant to any user request. The resulting responses from each shard (each will return at roughly the same response time given an indiscriminate distribution of data) is then combined and returned to a user request. We've now implemented a z -axis split of our data across the shards or partitions.

But what about high availability and the scale of transaction rates on a per-shard basis? The x -axis of the scale cube helps us here. Each shard of data (N ways) can have multiple copies of that data in a cloned or replicated fashion. This ensures that if one node within a shard is busy or unavailable, we can continue to answer the request. If transaction rates increase significantly but data volume does not, we simply add more x -axis clones to each shard. If data volume increases significantly but transaction rates remain constant, we add more z -axis shards. If both increase, we may use the z -axis or both the z - and x -axes to satisfy user demand. The aggregator service must also be cloned or scaled on the x -axis.

Now we simply need to implement load balancers to get the requests to the right aggregators, shards, and clones and finally back to the end user. Load balancers help level the load across the aggregators. Each aggregator makes N requests per the preceding discussion, and a second load-balancing agent helps ensure that we balance load across the M instances of x -axis clones for any shard. Voila! We've now implemented a roughly constant response time search (or fast read) system that scales well for both data and transaction growth.

Business-to-Business SaaS Solution

Many or business-to-business (B2B) SaaS solutions sell their products based on license or “per-seat” (per-employee) fees to other companies. In a great many of these companies to which the SaaS products are sold, value is created by ensuring that the employees can interact with one another and using common data as if that data were hosted on the customer's premises.

From the perspective of the provider of the service, the product needs to be multitenant to allow for economies of scale, while not being subject to the limitations of scale imposed by being “all-tenant.” As a reminder, all-tenant solutions are monolithic in nature and force us to scale vertically. Vertical scale, in turn, can place significant constraints on business growth once we reach the limitation of the biggest solution available for our needs. Multitenant, in contrast, means that we can collocate customers so as to efficiently use systems but does *not* require that we put all customers and data on a single system.

Referring back to Chapter 21, the approach of many B2B companies should become apparent. Salesforce uses the z -axis to split customers across pods (swim

lanes) such that each pod is multitenant but no pod is all-tenant. Salesforce further decomposes some services into unique instances (*y*-axis) within these pods and scales their databases horizontally using *x*-axis splits (refer to the figures and references in Chapter 21 for more detail). Moreover, Salesforce isn't the only B2B player that does this—ServiceNow has a similar architecture and approach,³ as do a number of our other B2B clients.

In fact, this pattern of *z*-axis first splits along customer boundaries is now common among many of our clients, as we've found it to be one of the most effective ways to scale within high-transaction systems. Industries to which we've applied this approach include financial services (banking and money-transmitters), tax preparation solutions, global commerce solutions, logistics solutions, video distribution systems, and many, many more.

Observations

We've twice now discussed when to use which axis of scale. We discussed this topic first in the "Observations" section of Chapter 23 and again earlier in this chapter after we explored each of the axes. The next obvious question you are probably asking is, "When do I decide to allow the application considerations to lead my architectural decisions and when do I decide to allow the data concerns to drive my decisions?"

The answer to that question is not an easy one, and again we refer you back to the "art" portion of our book's title. In some situations, the decision is easier to make, such as the decision within data warehousing discussions. Data warehouses are most often split by data concerns, though this is not always the case. The real question to ask is, "Which portion of my architecture most limits my scale?"

You might have a low-transaction, low-data-growth environment, but an application that is very complex. An example might be an encryption or cipher-breaking system. In such a case, the application likely needs to be broken up into services (*y*-axis) to allow specialists to develop the systems in question effectively. Alternatively, you might have a system such as a content site where the data itself (the content) drives most scalability limits; as such, you should design your architecture around these concerns.

If you are operating a site with transaction growth, complexity growth, and growth in data, you will probably switch between the "application leading design" and "database leading design" approaches to meet your needs. You choose the one that is most limiting for any given area and allow it to lead your architectural efforts. The most mature teams see them as one holistic system.

3. Advanced high availability. *ServiceNow*. <http://www.servicenow.com/content/dam/servicenow/documents/whitepapers/wp-advanced-high-availability-20130411.pdf>.

Timeline Considerations

One of the most common questions we get asked is, “When do I perform an *x*-axis split, and when should I consider *y*- and *z*-axis splits?” Put simply, this question really addresses whether there is an engineered maturity to the process of the AKF Scale Cube. In theory, there is no general timeline for these splits, but in implementation, most companies follow a similar path.

Ideally, a technology or architecture team would select the right axes of scale for its data and transaction growth needs and implement them in a cost-effective manner. Systems with high transaction rates, low data needs, and a high read-to-write ratio are probably most cost-effectively addressed with an *x*-axis split. Such a system or component may never need more than simple replication in both the data tier and the systems tier. Where customer data growth, complex functionality, and high transaction growth intersect, however, a company may need to perform all three axes.

In practice, what typically happens is that a technology team finds itself in a bind and needs to do something quickly. Most often, *x*-axis splits are easiest to implement in terms of time and overall cost. The team may rush to this implementation, then start to look for other paths. The *y*- and *z*-axis splits tend to follow, with *y*-axis implementations tending to be more common as a second step than *z*-axis implementations, due to the conceptual ease of splitting functions within an application.

Our recommendation is to design your systems with all threes axes in mind. At the very least, make sure that you are not architecting solutions that preclude you from easily splitting customers or functions in the future. Attempt to implement your product with *x*-axis splits for both the application and the database, and have designs available to split the application and data by both functions and customers. In this fashion, you can rapidly scale, should demand take off, without struggling to keep up with the needs of your end users.

Conclusion

This chapter discussed the application of the AKF Scale Cube to databases and data architectures within a product, service, or platform. We modified the AKF Scale Cube slightly, narrowing the scope and definition of each of the axes so that it became more meaningful in the context of databases and data architecture.

Our *x*-axis still addresses the growth in transactions or work performed by any platform or system. Although the *x*-axis handles the growth in transaction volume well, it suffers from the limitations of replication technology and does not handle data growth well.

The *y*-axis addresses data growth as well as transaction growth. Unfortunately, it does not distribute demand well across databases, because it focuses on data

affinity. As such, it often produces irregular demand characteristics, which might make capacity modeling difficult and likely will require *x*- or *z*-axis splits.

The *z*-axis addresses growth in data and is most often related to customer growth or inventory element (product) growth. The *z*-axis splits have the ability to more evenly distribute demand (or load) across a group of systems.

Not all companies need all three axes of scale to survive. When more than one axis is employed, the *x*-axis is almost always subordinate to the other axes. You might, for instance, have multiple *x*-axis splits, each occurring within a *y*- or *z*-axis split. Ideally, all such splits will occur in relationship to application splits, with either the application or the data being the reason for making a split.

Key Points

- Although *x*-axis database splits scale linearly with transaction growth, they usually have predetermined limits as to the number of splits allowed. They do not help with the growth in customers or data. The various *x*-axis splits are mirrors of each other.
- The *x*-axis tends to be the least costly to implement.
- The *y*-axis database splits help scale data as transaction growth. They are mostly meant for data scale because in isolation they are not as effective as the *x*-axis in transaction growth.
- The *y*-axis splits tend to be more costly to implement than the *x*-axis splits because of the engineering time needed to separate monolithic databases.
- The *y*-axis splits aid in fault isolation.
- The *z*-axis application splits help scale transaction and data growth.
- The *z*-axis splits allow for more even demand or load distribution than most *y*-axis splits.
- Like *y*-axis splits, *z*-axis splits aid in fault isolation.
- The choice of when to use which method or axis of scale involves both art and science, as does the decision of when to use an “application leading architecture” split or a “data leading architecture” split.

Chapter 25

Caching for Performance and Scale

What the ancients called a clever fighter is one who not only wins, but excels in winning with ease.

—Sun Tzu

What is the best way to handle large volumes of traffic? This is, of course, a trick question and at this point in the book we hope you answered something like “Establish the right organization, implement the right processes, and follow the right architectural principles to ensure the system can scale.” That’s a great answer. An even better answer is to say, “Not to handle it at all.” While this may sound too good to be true, it can be accomplished. The guideline of “Don’t handle the traffic if you can avoid it” should be a mantra of your architects and potentially even an architectural principle. The key to achieving this goal is pervasive use of caches.

In this chapter, we will cover caching and explore how it can be one of the best tools in your scalability toolbox. Caching is built into most of the tools we employ, from CPU caches to DNS caches to Web browser caches. Understanding cache approaches will allow you to build better, faster, and more scalable products.

This chapter covers the three levels of caching that are most under your control from an architectural perspective. We will start with a simple primer on caching and then discuss object caches, application caches, and content delivery networks (CDNs), considering how to leverage each for your product.

Caching Defined

Cache is an allocation of memory by a device or application for the temporary storage of data that is likely to be used again. This term was first used in 1967 in the publication *IBM Systems Journal* to label a memory improvement described as a

high-speed buffer.¹ Don't be confused by this point: Caches and buffers have similar functionality but differ in purpose. Both buffers and caches are allocations of memory and have similar structures. A buffer is memory that is used temporarily for access requirements, such as when data from disk must be moved into memory for processor instructions to utilize it. Buffers can also be used for performance, such as when reordering of data is required before writing to disk. A cache, in contrast, is used for the temporary storage of data that is likely to be accessed again, such as when the same data is read over and over without the data changing.

The structure of a cache is similar to the array data structure implemented with key-value pairs. In a cache, these tuples or entries are called *tags* and *datum*. The tag specifies the identity of the datum, and the datum is the actual data being stored. The data stored in the datum is an exact copy of the data stored in either a persistent storage device, such as a database, or as calculated by an executable application. The tag is the identifier that allows the requesting application or user to find the datum or determine that it is not present in the cache. In Table 25.1, the cache has three items cached from the database—items with tags 3, 4, and 0. The cache can have its own index that could be based on recent usage or other indexing mechanism to speed up the reading of data.

Table 25.1 Cache Structure

(a) Database Index	Data
0	\$3.99
1	\$5.25
2	\$7.49
3	\$1.15
4	\$4.45
5	\$9.99

(b) Cache Index	Tag	Datum
0	3	\$1.15
1	4	\$4.45
2	0	\$3.99

1. According to the caching article in Wikipedia: <http://en.wikipedia.org/wiki/Cache>.

A *cache hit* occurs when the requesting application finds the data for which it is asking in the cache. When the data is not present in the cache (a *cache miss*), the application must go to the primary source to retrieve the data. The ratio of cache hits to requests is called the *cache ratio* or *hit ratio*. This ratio describes the effectiveness of the cache in removing requests from the primary source. Low cache ratios indicate poorly performing caches that may negatively impact performance due to the duplicative requests for data.

A few different methods are available for updating or refreshing data in a cache. The first is an offline process that periodically reads data from the primary source and completely updates the datum in the cache. There are a variety of uses for such a refresh method. One of the most common uses is to populate an empty cache for initial use, such as on startup of a system or product. Another is to recalculate and distribute large segments of data. This latter case may arise with solutions such as yield optimization or retargeting information for advertising engines.

Batch Cache Refresh

Once upon a time there was a batch job called `price_recalc`. This batch job calculated the new price of a product based on input from third-party vendors, which sometimes changed their prices daily or weekly as they ran specials on their products. Instead of running a pricing calculation on demand, our company has determined that based on the business rules, it is sufficient to calculate the price every 20 minutes.

Although we have saved a lot of resources by not dynamically calculating the price, we still do not want the other services to request it continuously from the primary data source, the database. Instead, we need a cache that stores the most frequently used items and prices. In this case, it does not make sense to dynamically refresh the cache because `price_recalc` runs every 20 minutes. It makes much more sense to refresh the cache on the same schedule that the batch job runs.

Another method of updating or refreshing data in a cache is to do so when a cache miss occurs. Here, the application or service requesting the data retrieves it from the primary data source and then stores it in the cache. Assuming the cache is filled, meaning that all memory allocated for the cache is full of data, storing the newly retrieved data requires some other piece of data to be ejected from the cache. Decisions about which piece of data to eject are the subject of an entire field of study. The algorithms that are used to make this determination are known as caching algorithms. One of the algorithms most commonly used in caching is the least recently used (LRU) heuristic, which removes the data that has been accessed furthest in the past.

In Figure 25.1, the service has requested Item #2 (step 1), which is not present in the cache and results in a cache miss. The request is reiterated to the primary source, the database (step 2), where it is retrieved (step 3). The application then must update the cache (step 4); it does so, creating the new cache by ejecting the least recently used item (index 2 tag 0 datum \$3.99). This is a sample of a cache miss with update based on the least recently used algorithm.

Another algorithm is the exact opposite of LRU—that is, the most recently used (MRU). LRU is relatively commonsensical, in that it generally makes sense that something not being used should be expunged to make room for something needed right now. The MRU at first take seems nonsensical, but in fact it has a use. If the likelihood that a piece of data will be accessed again is most remote when it first has been read, MRU works best. Let's return to our `price_recalc` batch job example. This time, we don't have room in our cache to store all the item prices, and the application accessing the price cache is a search engine bot. After the search engine bot has accessed the page to retrieve the price, it marks it off the list and is not likely to revisit this page or price again until all others have been accessed. In this situation, the MRU algorithm might be the most appropriate choice.

As we mentioned earlier, there is an entire field of study dedicated to caching algorithms. Some very sophisticated algorithms take a variety of factors, such as

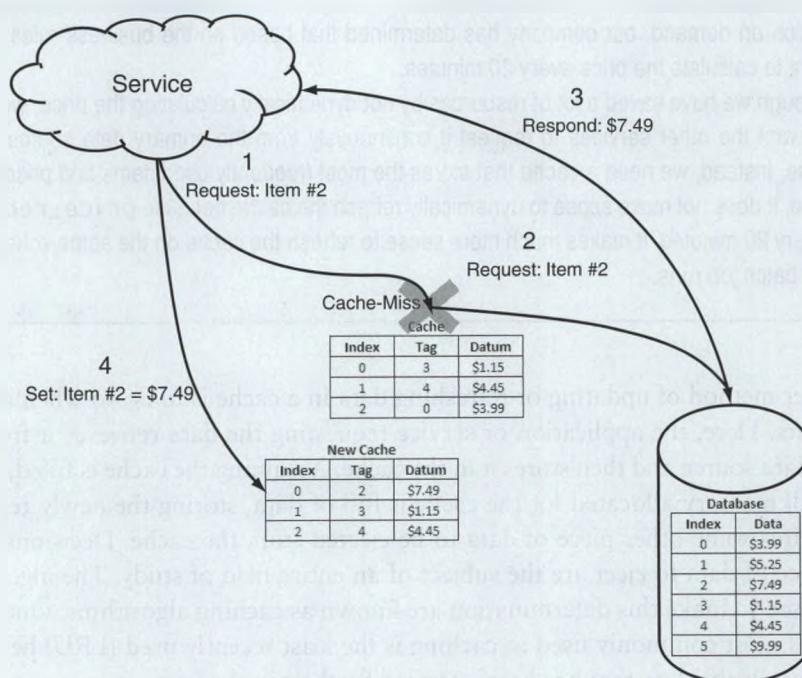


Figure 25.1 Cache Miss LRU

differences in retrieval time, size of data, and user intent, into account when determining which data stays and which data goes.

Thus far, we've focused on reading the data from the cache and assumed that only reads were being performed on the data. What happens when that data is manipulated and must be updated to ensure that if it is accessed again it is correct? In this case, we need to write data into the cache and ultimately get the data in the original data store updated as well. There are a variety of ways to achieve this. One of the most popular methods is a write-through policy, in which the application manipulating the data writes it into the cache and into the data store. With this technique, the application has responsibility for ensuring integrity between the stores.

In an alternative method, known as write-back, the cache stores the updated data until a certain point in the future. In this case, the data is marked as *dirty* so that it can be identified and understood that it has changed from the primary data source. Often, the future event that causes the write-back is the data being ejected from the cache. With this technique, the data is retrieved by a service and changed. This changed data is placed back into the cache and marked as dirty. When there is no longer room in the cache for this piece of data, it is expelled from the cache and written to the primary data store. Obviously, the write-back method eliminates the burden of writing to two locations from the service, but—as you can imagine—it also increases the complexity of many situations, such as when shutting down or restoring the cache.

In this brief overview of caching, we have covered the tag-datum structure of caches; the concepts of cache hit, cache miss, and hit ratio; the cache refreshing methodologies of batch and upon cache miss; caching algorithms such as LRU and MRU; and write-through versus write-back methods of manipulating the data stored in cache. Armed with the information from this brief tutorial, we are now ready to begin our discussion of three types of caches: object, application, and CDN.

Object Caches

Object caches are used to store objects for the application to be reused. These objects usually either come from a database or are generated by calculations or manipulations of the application. The objects are almost always serialized objects, which are marshalled or deflated into a serialized format that minimizes the memory footprint. When retrieved, the objects must be converted into their original data type (a process typically called unmarshalling). *Marshalling* is the process of transforming the memory representation of an object into a byte-stream or sequence of bytes so that it can be stored or transmitted. *Unmarshalling* is the process of decoding from the byte representation into the original object form. For object caches to be used, the application must be aware of them and have implemented methods to manipulate the cache.

The basic methods of manipulation of a cache include a way to add data into the cache, a way to retrieve it, and a way to update the data. These methods are typically called *set* for adding data, *get* for retrieving data, and *replace* for updating data. Depending on the particular cache that is chosen, many programming languages already have built-in support for the most popular caches. For example, Memcached is one of the most popular caches in use today. It is a “high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic Web applications by alleviating database load.”² This particular cache is very fast, using nonblocking network input/output (I/O) and its own slab allocator to prevent memory fragmentation, and thereby guaranteeing allocations to be $O(1)$ or able to be computed in constant time and thus not bound by the size of the data.³

As indicated in the description of Memcached, this cache is primarily designed to speed up Web applications by alleviating requests to an underlying database. This makes sense because the database is almost always the slowest retrieval device in the application tiers. The overhead of implementing ACID (atomicity, consistency, isolation, and durability) properties in a relational database management system can be relatively large, especially when data has to be written and read from disk. However, it is completely normal and advisable in some cases to use an object caching layer between other tiers of the system.

An object cache fits into the typical two- or three-tier architecture by being placed in front of the database tier. As indicated earlier, this arrangement is used because the database is usually the slowest overall performing tier and often the most expensive tier to expand. Figure 25.2 depicts a typical three-tier system stack consisting of a Web server tier, an application server tier, and a database tier. Instead of just one object cache, there are two. One cache is found in between the application servers and the database, and one is found between the Web servers and the application servers. This scheme makes sense if the application server is performing a great deal of calculations or manipulations that are cacheable. It prevents the application servers from having to constantly recalculate the same data, by allowing data to be cached and thereby relieving the load on the application servers. Just as with the database, this caching layer can help scale the tier without additional hardware. It is very likely that the objects being cached are a subset of the total data set from either the database or the application servers. For example, the application code on the Web servers might make use of the cache for user permission objects but not for transaction amounts, because user permissions are rarely changed and are accessed frequently; whereas a transaction amount is likely to be different with each transaction and accessed only once.

-
2. The description of Memcached comes from the Web site <http://www.danga.com/memcached/>.
 3. See the “Big O notation” entry in *Wikipedia*: http://en.wikipedia.org/wiki/Big_O_notation.

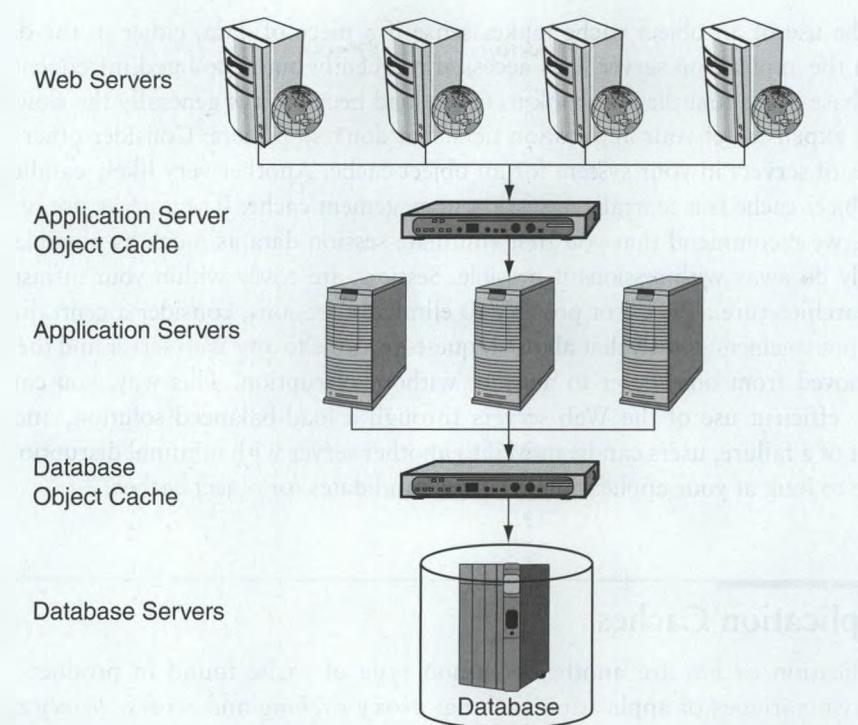


Figure 25.2 Object Cache Architecture

ACID Properties of a Database

Atomicity, consistency, isolation, and durability (ACID) are properties that a database management system employs to ensure that transactions are considered completely reliable.

Atomicity is a property of a database management system that guarantees all tasks of a transaction are completely performed or the entire transaction is rolled back. Failures of hardware or software will not result in half-completed transactions.

Consistency is the property that ensures the database remains in a steady state before and after a transaction. If a transaction is successful, it moves the database from one state to another that is “consistent” with the rules.

Isolation is the property that prevents another transaction from accessing a piece of data while another transaction is acting on it. Most database management systems use locks to ensure isolation.

Durability is the property that after the system has marked the transaction as successful, it will remain completed and not be rolled back. All consistency checks must be completed prior to the transaction being considered complete.

The use of an object cache makes sense if a piece of data, either in the database or in the application server, gets accessed frequently but is updated infrequently. The database is the first place to look to offset load because it is generally the slowest and most expensive of your application tiers. But don't stop there: Consider other tiers or pools of servers in your system for an object cache. Another very likely candidate for an object cache is a centralized session management cache. If you make use of session data, we recommend that you first eliminate session data as much as possible. Completely do away with sessions if possible. Sessions are costly within your infrastructure and architecture. If it is not possible to eliminate sessions, consider a centralized session management system that allows requests to come to any Web server and the session be moved from one server to another without disruption. This way, you can make more efficient use of the Web servers through a load-balanced solution, and in the event of a failure, users can be moved to another server with minimal disruption. Continue to look at your application for more candidates for object caches.

Application Caches

Application caches are another common type of cache found in products. There are two varieties of application caching: *proxy caching* and *reverse proxy caching*. Before we dive into the details of each, let's cover the concepts behind application caching in general. The basic premise is that you want to either speed up perceived performance or minimize the resources used. What do we mean by speeding up perceived performance? End users care only about how quickly the product responds and that they achieve their desired utility from a product. End users do not care which device answers their request, or how many requests a server can handle, or how lightly utilized a device may be.

Proxy Caches

How do you speed up response time and minimize resource utilization? The way to achieve this goal is not to have the application or Web servers handle the requests. Let's start out by looking at *proxy caches*. Internet service providers (ISPs), universities, schools, or corporations usually implement proxy caches. The terms *forward proxy cache* and *proxy server* are sometimes used to be more descriptive. The idea is that instead of the ISP having to transmit an end user's request through its network to a peer network to a server for the URL requested, the ISP can proxy these requests and return them from a cache without ever going to the URL's actual servers. Of course, this saves a lot of resources on the ISP's network from being used as well as speeds up the processing. The caching is done without the end user knowing that it has occurred; to her, the return page looks and behaves as if the actual site had returned her request.

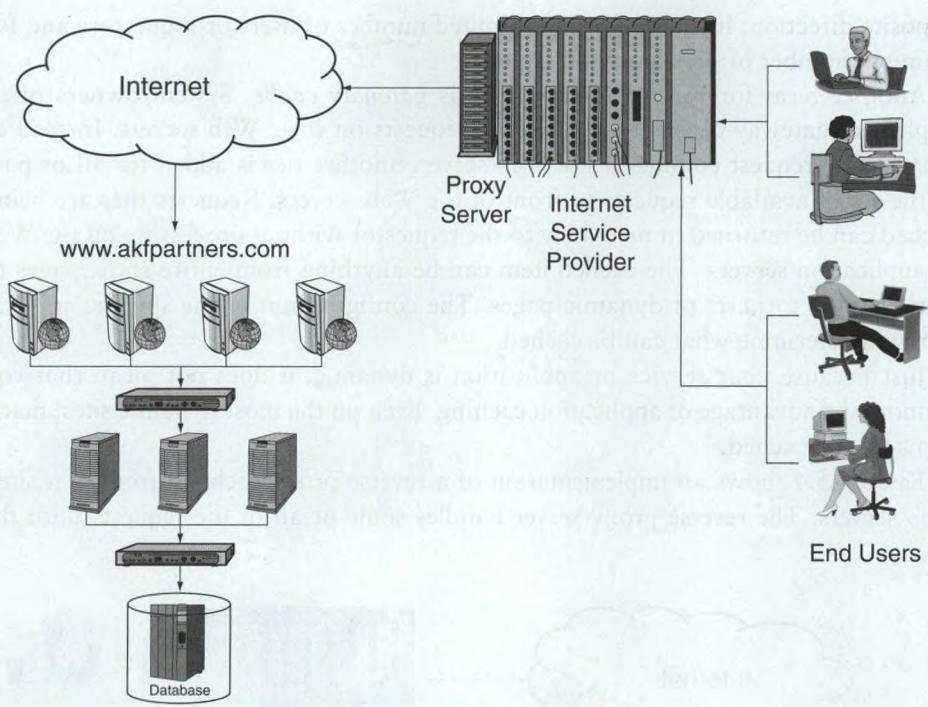


Figure 25.3 *Proxy Server Implementation*

Figure 25.3 shows an implementation of a proxy cache at an ISP. The ISP has implemented a proxy cache that handles requests from a limited set of users for an unlimited number of applications or Web sites. The limited number of users can be the entire subscriber population or—more likely—subsets of subscribers who are geographically colocated. All of these grouped users make requests through the cache. If the data is present, it is returned automatically; if the data is not present, the authoritative site is requested and the data is potentially stored in cache in case some other subscriber requests it. The caching algorithm that determines whether a page or piece of data gets updated/replaced can be customized for the subset of users who are using the cache. It may make sense for caching to occur only if a minimum number of requests for that piece of data is seen over a period of time. This way, the most requested data is cached and sporadic requests for unique data do not replace the most viewed data.

Reverse Proxy Caches

The other type of application caching is the reverse proxy cache. With proxy caching, the cache handles the requests from a limited number of users for a potentially unlimited number of sites or applications. A reverse proxy cache works in the

opposite direction: It caches for an unlimited number of users or requestors and for a limited number of sites or applications.

Another term for reverse proxy cache is *gateway cache*. System owners often implement gateway caches to offload the requests on their Web servers. Instead of every single request coming to the Web server, another tier is added for all or part of the set of available requests in front of the Web servers. Requests that are being cached can be returned immediately to the requestor without processing on the Web or application servers. The cached item can be anything from entire static pages to static images to parts of dynamic pages. The configuration of the specific application will determine what can be cached.

Just because your service or application is dynamic, it does not mean that you cannot take advantage of application caching. Even on the most dynamic sites, many items can be cached.

Figure 25.4 shows an implementation of a reverse proxy cache in front of a site's Web servers. The reverse proxy server handles some or all of the requests until the

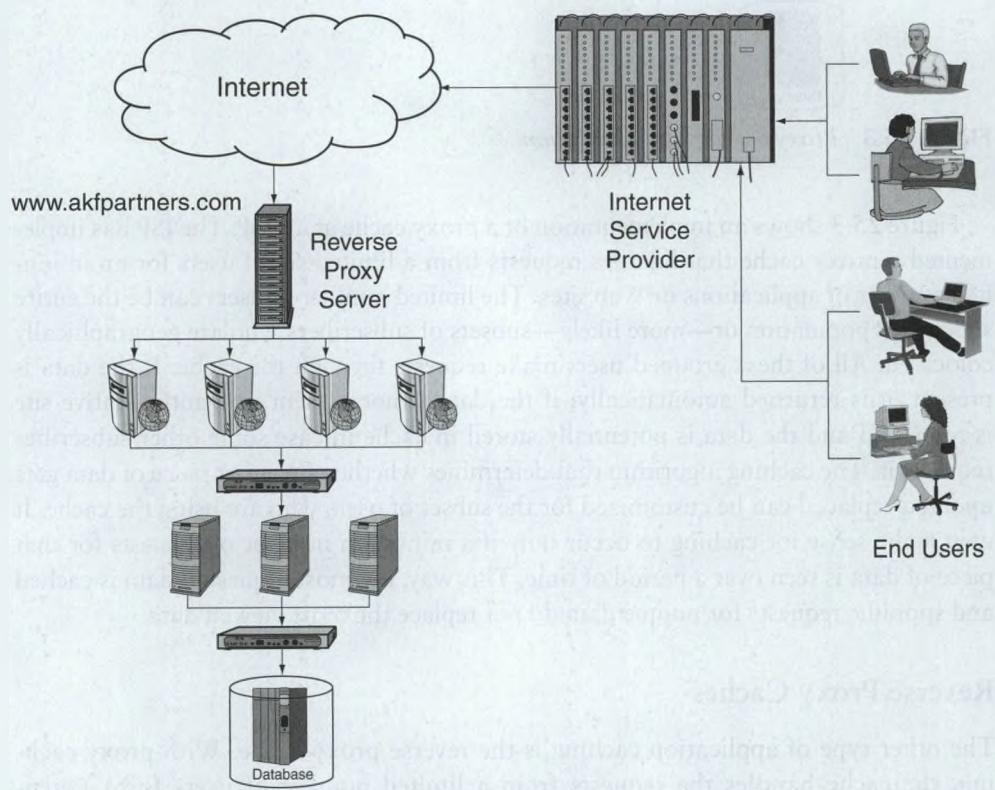


Figure 25.4 Reverse Proxy Server Implementation

pages or data that is stored in them is out of date or until the server receives a request for which it does not have data (a cache miss). When this occurs, the request is passed through to a Web server to fulfill the request and to refresh the cache. Any users that have access to make the requests for the application can be serviced by the cache. This is why a reverse proxy cache is considered the opposite of the proxy cache—a reverse proxy cache handles any number of users for a limited number of sites.

HTML Headers and Meta Tags

Many developers believe that they can control the caching of a page by placing meta tags, such as `Pragma: no-cache`, in the `<HEAD>` element of the page. This is only partially true. Meta tags in the HTML can be used to recommend how a page should be treated by browser cache, but most browsers do not honor these tags. Because proxy caches rarely even inspect the HTML, they do not abide by these tags.

HTTP headers, by comparison, give much more control over caching, especially with regard to proxy caches. These headers cannot be seen in the HTML and are generated by the Web server. You can control them by configurations on the server. A typical HTTP response header could look like this:

```
HTTP/1.1 200 OK
Date: Tue, 24 Feb 2009 19:52:41 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Mon, 26 Jan 2009 23:03:35 GMT
Etag: "1189c0-249a-bf8d9fc0"
Accept-Ranges: bytes
Content-Length: 9370
p3p: policyref="/w3c/p3p.xml",
CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM NAV"
Connection: close
Cache-Control: no-cache
Content-Type: image/gif
```

Notice the `Cache-Control` header identifying `no-cache`. In accordance with the Request for Comments (RFC) 2616 Section 14 defining the HTTP 1.1 protocol, this header *must* be obeyed by all caching mechanisms along the request/response chain.

Another header that is useful in managing caching consists of the `Etag` and `Last-Modified` tags. These are used to validate the freshness of the page by the caching mechanisms.

Understanding the request and response HTTP headers will allow you to more fully control the caching of your pages. This is an exercise that is well worth doing to ensure your pages are being handled properly by all the caching layers between your site and the end users.

Caching Software

Adequately covering even a portion of the caching software that is available both from vendors and from the open source communities is beyond the scope of this chapter. However, some points will be covered to guide you in your search for the right caching software for your company's needs.

The first point is that you should thoroughly understand your application and user demands. Running a site with multiple gigabytes per second of traffic requires a much more robust and enterprise-class caching solution than does a small site serving 10MB per second of traffic. Are you projecting a doubling of requests or users or traffic every month? Are you introducing a brand-new video product line that will completely change the type of and need for caching? These are the types of questions you need to ask yourself before you start shopping the Web for a solution, or you could easily fall into the trap of making your problem fit the solution.

The second point addresses the difference between add-on features and purpose-built solutions and is applicable to both hardware and software solutions. To understand the difference, let's discuss the life cycle of a typical technology product. A product usually starts out as a unique technology that garners sales and gains traction, or is adopted in the case of open source, as a result of its innovation and benefit within its target market. Over time, this product becomes less unique and eventually commoditized, meaning everyone sells essentially the same product with the primary differentiation being price. High-tech companies generally don't like selling commodity products because the profit margins continue to get squeezed each year. Open source communities are usually passionate about their software and want to see it continue to serve a purpose. The way to prevent the margin squeeze or the move into the history books is to add features to the product. The more "value" the vendor adds, the more likely it is that the vendor can keep the price high. The problem with this strategy is that these add-on features are almost always inferior to purpose-built products designed to solve this one specific problem.

An example of this can be seen in comparing the performance of `mod_cache` in Apache as an add-on feature with that of the purpose-built product Memcached. This is not to belittle or take away anything from Apache, which is a very common open source Web server that is developed and maintained by an open community of developers known as the Apache Software Foundation. The application is available for a wide variety of operating systems and has been the most popular Web server on the World Wide Web since 1996. The Apache module, `mod_cache`, implements an HTTP content cache that can be used to cache either local or proxied content. This module is one of hundreds available for Apache, and it absolutely serves a purpose—but when you need an object cache that is distributed and fault tolerant, better solutions are available, such as Memcached.

Application caches are extensive in their types, implementations, and configurations. You should first become familiar with the current and future requirements of your application. Then, you should make sure you understand the differences between add-on features and purpose-built solutions. With these two pieces of knowledge, you will be ready to make a good decision when it comes to the ideal caching solution for your application.

Content Delivery Networks

The last type of caching that we will cover in this chapter is the content delivery network. This level of caching is used to push any of your content that is cacheable closer to the end user. The benefits of this approach include faster response time and fewer requests on your servers. The implementation of a CDN varies, but most generically can be thought of as a network of gateway caches located in many different geographic areas and residing on many different Internet peering networks. Many CDNs use the Internet as their backbone and offer their servers to host your content. Others, to provide higher availability and differentiate themselves, have built their own point-to-point networks between their hosting locations.

The advantages of CDNs are that they speed up response time, offload requests from your application's origin servers, and possibly lower delivery cost, although this is not always the case. The concept is that the total capacity of the CDN's strategically placed servers can yield a higher capacity and availability than the network backbone. The reason for this is that if there is a network constraint or bottleneck, the total throughput is limited. When these obstacles are eliminated by placing CDN servers on the edge of the network, the total capacity increases and overall availability increases as well.

With this strategy, you use the CDN's domain as an alias for your server by including a canonical name (CNAME) in your DNS entry. A sample entry might look like this:

ads.akfpartners.com	CNAME	ads.akfpartners.akfdn.net
---------------------	-------	---------------------------

Here, we have our CDN, akfdn.net, as an alias for our subdomain ads.akfpartners.com. The CDN alias could then be requested by the application. As long as the cache was valid, it would be served from the CDN and not our origin servers for our system. The CDN gateway servers would periodically make requests to our application origin servers to ensure that the data, content, or Web pages that they have in cache is up-to-date. If the cache is out-of-date, the new content is distributed through the CDN to their edge servers.

Today, CDNs offer a wide variety of services in addition to the primary service of caching your content closer to the end user. These services include DNS replacement; geo-load balancing, which is serving content to users based on their geographic locations; and even application monitoring. All of these services are becoming more commoditized as more providers enter into the market. In addition to commercial CDNs, more peer-to-peer (P2P) services are utilized for content delivery to end users to minimize the bandwidth and server utilization from providers.

Conclusion

The best way to handle large amounts of traffic is to avoid handling this traffic in the first place—ideally by utilizing caching. In this manner, caching can be one of the most valuable tools in your toolbox for ensuring scalability. Numerous forms of caching exist, ranging from CPU cache to DNS cache to Web browser caches. However, three levels of caching are most under your control from an architectural perspective: caching at the object, application, and content delivery network levels.

In this chapter, we offered a primer on caching in general and covered the tag-datum structure of caches and how they are similar to buffers. We also covered the terminology of cache hit, cache miss, and hit ratio. We discussed the various refreshing methodologies of batch and upon cache miss as well as caching algorithms such as LRU and MRU. We finished the introductory section with a comparison of write-through versus write-back methods of manipulating the data stored in cache.

Object caches are used to store objects for the application to be reused. Objects stored within the cache usually either come from a database or are generated by the application. These objects are serialized to be placed into the cache. For object caches to be used, the application must be aware of them and have implemented methods to manipulate the cache. The database is the first place to look to offset load through the use of an object cache, because it is generally the slowest and most expensive of your application tiers; however, the application tier is often a target as well.

Two varieties of application caching exist: proxy caching and reverse proxy caching. The basic premise of application caching is that you desire to speed up performance or minimize resources used. Proxy caching is used for a limited number of users requesting an unlimited number of Web pages. Internet service providers or local area networks such as in schools and corporations often employ this type of caching. A reverse proxy cache is used for an unlimited number of users or requestors and for a limited number of sites or applications. System owners most often implement reverse proxy caches to offload the requests on their application origin servers.

The general principle underlying content delivery networks is to push content that is cacheable closer to the end user. Benefits of this type of caching include faster

response time and fewer requests on the origin servers. CDNs are implemented as a network of gateway caches in different geographic areas utilizing different ISPs.

No matter which type of service or application you provide, it is important to understand the various methods of caching so that you can choose the right type of cache. There is almost always a caching type or level that makes sense with SaaS systems.

Key Points

- The most easily scalable traffic is the type that never touches the application because it is serviced by caching.
- There are many layers at which to consider adding caching, each with pros and cons.
- Buffers are similar to caches and can be used to boost performance, such as when reordering of data is required before writing to disk.
- The structure of a cache is very similar to that of data structures, such as arrays with key-value pairs. In a cache, these tuples or entries are called tags and datum.
- A cache is used for the temporary storage of data that is likely to be accessed again, such as when the same data is read over and over without the data changing.
- When the requesting application or user finds the data that it is asking for in the cache, the situation is called a cache hit.
- When the data is not present in the cache, the application must go to the primary source to retrieve the data. Not finding the data in the cache is called a cache miss.
- The number of hits to requests is called a cache ratio or hit ratio.
- The use of an object cache makes sense if you have a piece of data either in the database or in the application server that is accessed frequently but updated infrequently.
- The database is the first place to look to offset load because it is generally the slowest and most expensive application tier.
- A reverse proxy cache—also called a gateway cache—caches for an unlimited number of users or requestors and for a limited number of sites or applications.
- Reverse proxy caches are most often implemented by system owners to offload the requests on their Web servers.
- Many content delivery networks use the Internet as their backbone and offer their servers to host your content.

- Other content delivery networks, to provide higher availability and differentiate themselves, have built their own point-to-point networks between their hosting locations.
- The advantages of CDNs are that they lower delivery cost, speed up response time, and offload requests from your application's origin servers.

Chapter 26

Asynchronous Design for Scale

In all fighting, the direct method may be used for joining battle,
but indirect methods will be needed in order to secure victory.

—Sun Tzu

This chapter addresses two common problems for many products: the reliance upon state and the use of synchronous communications. We explain the impact of synchronicity on both availability and customer satisfaction. We suggest approaches to mitigate these effects and provide a rationale for moving most transactions to asynchronous implementations. We evaluate common reasons for implementing state, suggest alternative approaches, and arm the reader with mitigation strategies when state is required.

Synching Up on Synchronization

The process of synchronization refers to the use and coordination of simultaneously executed threads or processes that are part of an overall task. These processes must run in the correct order to avoid a race condition or erroneous results. Stated another way, synchronization occurs when two or more pieces of work must be completed in a specific order to accomplish a task. The login task provides an excellent example: First, the user's password must be encrypted; then it must be compared against the encrypted version in the database; then the session data must be updated, marking the user as authenticated; then the welcome page must be generated and presented. If any of those processes is conducted out of sequence, the task of logging the user in fails.

One example of synchronization is mutual exclusion (abbreviated mutex). Mutex refers to how global resources are protected from concurrently running processes to ensure only one process updates or accesses the resource at a time. This is often accomplished through semaphores. Semaphores are variables or data types that flag a resource as being either in use or free.

Another classic synchronization method is thread join, in which a process is blocked from executing until a particular thread terminates. After the thread terminates, the waiting process is free to continue. For example, a parent process, such as a “lookup,” would start executing. The parent process kicks off a child process to retrieve the location of the data that it will look up, and this child thread is “joined”—that is, the parent process pauses further execution until the child process terminates.

Dining Philosophers Problem

The dining philosophers analogy is credited to Sir Charles Anthony Richard Hoare (also known as Tony Hoare), the inventor of the Quicksort algorithm. This analogy is used as an illustrative example of resource contention and deadlock.

The story goes as follows: Five philosophers were sitting around a table with a bowl of spaghetti in the middle. Each philosopher had a fork to his left, and therefore each had a fork to his right. The philosophers could either think or eat, but not both. Additionally, to serve and eat the spaghetti, each philosopher required the use of two forks. Without any coordination, it is possible that all the philosophers would pick up their forks simultaneously, so that no one would have two forks with which to serve or eat.

The analogy demonstrates that without synchronization the five philosophers could remain stalled indefinitely and starve, just as five computer processes waiting for a resource could all enter into a deadlocked state. Fortunately, there are many ways to solve such a dilemma. One is to have a rule that each philosopher, upon reaching a deadlock state, will place his fork down, thereby freeing up a resource, and think for a random amount of time. This solution may sound familiar, because it forms the basis for Transmission Control Protocol (TCP) retransmission processes. When no acknowledgment for data is received, a timer is started to wait for a retry. The amount of time is adjusted by the smoothed round trip time algorithm and doubled after each unsuccessful retry.

The preceding is by no means an exhaustive list of synchronization methods. We present it only to illustrate the use and value of synchronization. The complete elimination of synchronization is neither possible nor advisable. That said, many implementations run counter to our desires to create highly scalable, highly available solutions. Let’s discuss those uses and the alternatives.

Synchronous Versus Asynchronous Calls

Now that we have a basic definition and some examples of synchronization, we can move on to a broader discussion of synchronous versus asynchronous calls within a

product. Synchronous calls occur when one process calls another process, and then stalls its execution awaiting a response. As an example of a synchronous method, let's look at a method that we'll call `query_exec`. This method is used to build and execute a dynamic database query. One step in the `query_exec` method is to establish a database connection. The `query_exec` method does not continue executing without explicit acknowledgment of successful completion of this database connection task. If the database is unavailable, the application will stall. In synchronous calls, the originating call is halted and not allowed to complete until the invoked process returns.

A nontechnical example of synchronicity is communication between two individuals either in a face-to-face fashion or over a phone line. If both individuals are engaged in meaningful conversation, it is unlikely that any other action is going on. One individual cannot easily start another conversation with a second partner without first stopping the conversation with the first partner. Phone lines are held open until one or both callers terminate the call.

Now compare this synchronous method to an asynchronous one. An asynchronous method call launches a new thread on execution, and then immediately returns control back to the thread from which it originated. In other words, the *initiating* thread does not wait for the *initiated* thread to complete (or answer) before continuing to execute. The design pattern that describes the asynchronous method call is known as asynchronous design, or asynchronous method invocation (AMI). The asynchronous call continues to execute in another thread and terminates either successfully (or with error) without further interaction with the initiating thread.

Let's return to our example with the `query_exec` method. After making a synchronous database connection call, this method needs to prepare and execute the query. Perhaps there is a monitoring framework that allows the service to note the duration and success of all queries by asynchronously calling a method for `start_query_time` and `end_query_time`. These methods store a system time in memory and wait for the end call to be placed to calculate duration. The duration is then stored in a monitoring database that can be queried to understand how well the system is performing in terms of query run time. Monitoring the query performance is important—but not as important as actually servicing the users' requests. Therefore, the calls to the monitoring methods (i.e., `start_query_time` and `end_query_time`) are done asynchronously. If they succeed and return, great—but if the monitoring calls fail or are delayed for 20 seconds waiting on the monitoring database connection, nobody cares. The user query continues on without interference from asynchronous calls.

Returning to our communication example, email is a great example of asynchronous communication. You write an email and send it, then immediately move on to another task, which may be another email, a round of golf, or something else. A response may come in from the person to whom you sent the email. You are free to read it at any time and then respond at your leisure. Neither you nor the second party is blocked from doing other things while you await a response.

Scaling Synchronously or Asynchronously

Why does understanding the differences between synchronous and asynchronous calls matter? Synchronous calls, if used excessively or incorrectly, can place an undue burden on the system and prevent it from scaling. To see how, let's return to our `query_exec` example.

Why wouldn't it make sense to implement the monitoring calls synchronously? After all, monitoring is important, the methods will execute quickly, and the impact on overall execution time will be small for the short duration we block waiting for start and end times to be populated. As we stated earlier, monitoring is important, but not more important than returning a user's query. The monitoring methods might work quickly when the monitoring database is operational, but what happens when a hardware failure occurs and the database is inaccessible? The monitoring queries will back up waiting to time out. The users' queries that launched these monitoring queries will, in turn, become backed up. These blocked user queries maintain their connection to the user (nonmonitoring) database and consume memory on the application server trying to execute this thread. As the number of database connections grows, the user database might run out of available connections, thereby preventing every other query from executing. Threads on the app servers, in turn, get tied up waiting for database responses. Memory swapping begins on the app servers as blocked threads are written to disk. This swapping slows down all processing and may result in the TCP stack of an app server reaching its limit and refusing end-user connections. Ultimately, new user requests are not processed and users sit waiting for browser or application timeouts. Your application or platform is essentially "down."

As you see, this ugly chain of events can quite easily occur because of a simple oversight on whether a call should be synchronous or asynchronous. The worst thing about this scenario is that the root cause can be elusive. Stepping through the chain of events reveals that it is relatively easy to follow, but when the only symptom associated with an incident is that your product's Web pages are loading slowly, diagnosing the problem can be time consuming. Ideally, you will have sufficient monitoring in place to help you diagnose these types of problems. Nevertheless, such extended chains of events can be daunting to unravel when your site is down and you are frantically scrambling to get it back into service.

Why do we continue to use synchronous calls when they can have such a disastrous effect on availability? The answer is that synchronous calls are simpler than asynchronous calls. "But wait!" you say. "Yes, they are simpler, but oftentimes our methods require that the other methods invoked successfully complete, so we can't put a bunch of asynchronous calls in our system." This is a valid point. Many times you do need an invoked method to complete, and you need to know the status of that call to continue along your thread. Not all synchronous calls are bad; in fact,

many are necessary and make the developer's life far less complicated. At many other times, however, asynchronous calls can and should be used in place of synchronous calls, even when a dependency exists (as described earlier).

If the main thread is oblivious to whether the invoked thread finishes—such as with monitoring calls—a simple asynchronous call is all that is required. If, however, you require information from the invoked thread, but you don't want to stop the primary thread from executing, callbacks can be implemented to retrieve this information. An example of callback functionality is the interrupt handlers in operating systems that report on hardware conditions.

Asynchronous Coordination

Asynchronous coordination and communication between the original method and the invoked method requires a mechanism by which the original method determines when or if a called method has completed executing. Callbacks are methods passed as an argument to other methods and allow for the decoupling of different layers in the code.

In C/C++, this is done through function pointers. In Java, it is done through object references.

Many design patterns use callbacks, such as the delegate design pattern and the observer design pattern. The higher-level process acts as a client of the lower-level process and calls the lower-level method by passing it by reference. For example, a callback method might be invoked for an asynchronous event like file system changes.

In the .NET Framework, the asynchronous communication is leveraged by the keywords `Async` and `Await`. By using these two keywords, a developer can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method.

Different languages offer different solutions to the asynchronous communication and coordination problem. Understand what your language and frameworks offer so that you can implement them when needed.

We mentioned that synchronous calls are simpler than asynchronous calls and, therefore, are more frequently implemented. This is only one reason why synchronous calls are so common. Another reason is that developers typically see only a small portion of the product. Few people in the organization have the advantage of viewing the application in its totality. Architects (as well as someone on the management team) should be looking at the application from this higher-level perspective. These are the people you'll need to rely on to help identify and explain how synchronization may result in scaling issues.

Example Asynchronous Systems

In this section, we employ a fictitious system to explore how you can either convert an existing system to asynchronous calls or design a new system to use them. To help you fully understand how synchronous calls can cause scaling issues and how you can either design from the start or convert a system in place to use asynchronous calls, we'll invoke an example system that we can explore. The system we discuss is taken from an actual client implementation, but is obfuscated to protect privacy.

This system, which we'll call MailScale, allowed subscribed users to email groups of other users with special notices, newsletters, and coupons (see Figure 26.1). The volume of emails sent in a single campaign could be very large, as many as several hundred thousand recipients. These jobs were obviously done asynchronously from the main site. When a subscribed user was finished creating or uploading the email notice, he or she submitted the email job to be processed. Because processing tens of thousands of emails can take several minutes, it really would be ridiculous to hold up the user's completed page with a synchronous call while the job actually processes.

The problem was that behind the main site were schedulers that queued the email jobs and parsed them out to available email servers when they became available. These schedulers were the service that received the email job from the main site when submitted by the user. This was done synchronously: A user clicked Send, the call was placed to the scheduler to receive the email job, and a confirmation was returned that the job was received and queued. This made sense, as the submission

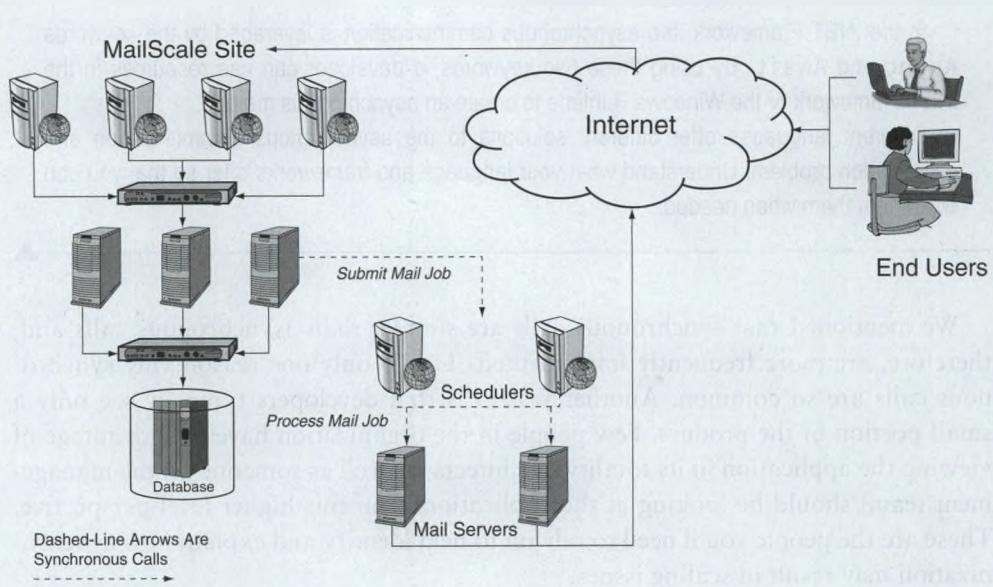


Figure 26.1 *MailScale Example*

task should not fail without notifying the user. Because the call took only a few hundred milliseconds on average, a simple synchronous method seemed appropriate. Unfortunately, the engineer who made this decision was unaware that the schedulers were, in turn, placing synchronous calls to the mail servers.

When a scheduler received a job, it queued it up until a mail server became available. Once a mail server was available, the scheduler would establish a synchronous stream of communication between itself and the mail server to pass all the information about the job and monitor the job while it completed.

When all the mail servers were running under maximum capacity, and the proper number of schedulers was available for the number of mail servers, everything worked fine. When mail slowed down because of an excessive number of bounce-back emails or an ISP mail server that was slow in receiving the outbound emails, the MailScale email servers would slow down and get backed up. This, in turn, backed up the schedulers because they relied on a synchronous communication channel for monitoring the status of the jobs. When the schedulers slowed down and became unresponsive, user requests to submit mail jobs would fail. Application servers would slow down or quit accepting requests. Web servers then failed to accept end-user requests. The entire site became slow and unresponsive, all because of a chain of synchronous calls—the complete picture of which no single person was aware. The solution was to break the synchronous communication into asynchronous calls, preferably at both the app-to-scheduler and scheduler-to-email servers.

There are a few lessons to be learned here. The first and most important is that synchronous calls can cause problems in your system in unexpected places. One call can lead to another call and another, which can get very complicated with all the interactions and multitude of independent code paths through most systems.

The next lesson is that engineers usually do not have a good perspective on the overall architecture, which can cause them to make decisions that daisy-chain processes together. For this reason, architects and managers provide critical input to designs.

The final lesson we take from this example is the complexity of debugging problems of this nature. Depending on the monitoring system, it is likely that the first alert will come from the slowdown of the site and not from the mail servers. When such a failure occurs, naturally everyone starts looking at why the site is slowing down the mail servers, instead of the other way around. These problems can take a while to unravel and decipher.

Another reason to analyze and remove synchronous calls is the multiplicative effect of failure. We touched upon this issue briefly in Chapter 21, Creating Fault-Isolative Architectural Structures, and discuss it in more detail here. If you are old enough, you might remember the old-style Christmas tree lights. With these strings of lights, if a single bulb went out, it caused every other bulb in the entire string of lights to go out. These lights were wired in series, so should any single light fail, the

entire string would fail. As a result, the “availability” of the string of lights was the product of the availability (1—the probability of failure) of all the lights. If any light had a 99.999% availability or a 0.001% chance of failure and there were 100 lights in the string, the theoretical availability of the string of lights was 0.99999100 or 0.999, reduced from 5-nine availability to 3-nine availability (“nine availability” counts the number of 9s in the percent availability). In a year’s time, a 5-nine availability, 99.999%, has just over five minutes of downtime (i.e., bulbs out), whereas a 3-nine availability, 99.9%, has over 500 minutes of downtime. This equates to increasing the chance of failure from 0.001% to 0.1%. No wonder our parents hated putting up those lights!

Systems that rely upon each other for information in a series and in synchronous fashion are subject to the same rates of failure as the Christmas tree lights of yore. If a component fails, it will cause problems within the line of communication back to the end customer. The more calls we make, the higher the probability of failure. The higher the probability of failure, the more likely it is that we will hold open connections and refuse future customer requests. The easiest fix in this case is to make these calls asynchronous and ensure that they have a chance to recover gracefully with timeouts should they not receive responses in a timely fashion. If you’ve waited two seconds and a response hasn’t come back, simply discard the request and return a friendly error message to the customer.

Synchronous and asynchronous calls represent a necessary, but often missed, topic that must be discussed, debated, and taught to organizations. Ignoring these issues creates problems down the road when loads start to grow, servers start reaching maximum capacity, or services get added. Adopting principles, standards, and coding practices regarding synchronous and asynchronous communications now will save a lot of downtime and wasted resources when tracking down and fixing these problems in the future.

Defining State

Another often-ignored engineering topic is stateful versus stateless applications. An application that uses state is called *stateful*. This means it relies on the current condition of execution as a determinant of the next action to be performed. An application or protocol that doesn’t use state is referred to as *stateless*. Hyper-Text Transfer Protocol (HTTP) is a stateless protocol because it doesn’t need any information about the previous request to know everything necessary to fulfill the next request. An example of the use of state would be in a monitoring program that first identifies that a query was requested instead of a cache request and then, based on that information, calculates a duration time for the query. In a stateless implementation of

the same program, it would receive all the information that it required to calculate the duration at the time of request. If it were a duration calculation for a query, this information would be passed to the program upon invocation.

You may recall from a computer science computational theory class the description of Mealy and Moore machines, which are known as state machines or finite state machines. A state machine is an abstract model of states and actions that is used to model behavior. It can be implemented in the real world in either hardware or software. There are other ways to model or describe behavior of an application, but the state machine is one of the most popular.

Mealy and Moore Machines

A *Mealy machine* is a finite state machine that generates output based on the input and the current state of the machine. A *Moore machine* is a finite state machine that generates output based solely on the current state.

A very simple example of a Moore machine is a turn signal that alternates between on and off. The output—the light being turned on or off—is completely determined by the current state. If it is on, the light gets turned off. If it is off, the light gets turned on.

A very simple example of a Mealy machine is a traffic signal. Assume that the traffic signal has a switch to determine whether a car is present. The output is the traffic light's color—red, yellow, or green. The input is a car at the intersection waiting on the light. The output is determined by the current state of the light as well as the input. If a car is waiting and the current state is red, the signal gets turned to green.

These Moore and Mealy machines are both simple examples, but the point is that there are different ways of modeling behavior using states, inputs, outputs, and actions.

When you are running an application as a single instance on a single server, the state of the machine is known and easy to manage. All users run on the same server, so knowing that a particular user has logged in allows the application to use this state of being logged in and whatever input arrives, such as clicking a link, to determine the resulting output. Complexity comes about when we begin to scale our application along the *x*-axis by adding servers. If a user arrives on one server for this request and on another server for the next request, how would each machine know the current state of the user? If your application is split along the *y*-axis and the login service is running in a completely different pool than the report service, how does each of these services know the state of the other? These types of questions arise when trying to scale applications that require state. They are not insurmountable, but they do require

some thought to resolve—ideally before you are in a bind with your current capacity and have to rush out a new server or split the services.

One of the most common implementations of state is the user session. Just because an application is stateful does not mean that it must have user sessions. The opposite is also true. An application or service that implements a session may do so in a stateless manner—consider the stateless session beans in Enterprise JavaBeans. A user session is an established communication link between the client, typically the user's browser, and the server; this link is maintained during the life of the session for that user. Although developers store lots of things in user sessions, perhaps the most commonly held data is the fact that the user is logged in and has certain privileges. This information obviously is important unless you want to continue validating the user's authentication on each page request. Other items typically stored in a user session include account attributes such as preferences for first-seen reports, layout, or default settings. Again, retrieving these items once from the database and then keeping them with the user during the session can be the most economical thing to do.

Sometimes you may want to store something in a user's session, but realize storing this information can be problematic in terms of increased complexity. It makes sense to not have to constantly communicate with the database to retrieve a user's preferences as the user bounces around your site, but this improved performance creates difficulties when a pool of servers is handling user requests. Another complexity of maintaining session state is that if you are not careful, the amount of information stored in the user session will become unwieldy. Sometimes (albeit rarely), an individual user's session data reaches or exceeds hundreds of kilobytes. Of course, this case is excessive, but we've seen clients fail to manage their session data, with the result being a Frankenstein's monster in terms of both size and complexity. Every engineer wants his information to be quickly and easily available, so he sticks his data in the session. When you step back and look at the size of the data and the obvious problems of keeping all these user sessions in memory or transmitting them back and forth between the user's browser and the server, it becomes obvious this situation needs to be remedied quickly.

If you have managed to keep the user sessions to a reasonable size, which methods are available for saving state or keeping sessions in environments with multiple servers? There are three basic approaches: avoid, centralize, and decentralize.

Similar to our approach with caching, the best way to solve a user session scaling issue is to avoid having the issue at all. You can achieve this by either removing session data from your application or making it stateless. The other way to achieve avoidance is to make sure each user is placed on only a single server. This way, the session data can remain in memory on the server because that user will always come

back to that server for requests; other users will go to other servers in the pool. You can accomplish this manually in the code by performing a *z-axis split* (modulus or lookup)—for example, by putting all users with usernames A–M on one server and all users with usernames N–Z on another server. If DNS pushes a user with username “jackal” to the second server, that server just redirects her to the first server to process her request. Another potential solution is to use session cookies on the load balancer. These cookies assign all users to a particular server for the duration of the session. This way, every request that comes through from a particular user will land on the same server. Almost all load balancer solutions offer some sort of session cookie that provides this functionality.

Let’s assume that for some reason none of these solutions works. The next method of solving the complexities of keeping session state on a myriad of servers when scaling is decentralization of session storage. This can be accomplished by storing session data in a cookie on the user’s browser. Many implementations of this approach are possible, such as serializing the session data and then storing all of it in a cookie. This session data must be transferred back and forth, marshalled/unmarshalled, and manipulated by the application—processing that can require a significant amount of time. Recall that marshalling and unmarshalling are processes whereby the object is transformed into a data format suitable for transmitting or storing and converted back again. Another twist on this approach is to store a very little amount of information in the session cookie and then use it as a reference index to a list of objects in a session database or file that contains the full set of session data for each user. With this strategy, both transmission and marshalling costs are minimized.

The third method of solving the session problem with scaling systems is centralization. With this technique, all user session data is stored centrally in a cache system and all Web and app servers can access this data. Then, if a user lands first on Web Server 1 for the login and then on Web Server 3 for a report, both servers can access the central cache and see whether the user is logged in and what that user’s preferences are. A centralized cache system such as Memcached (discussed in Chapter 25, Caching for Performance and Scale) would work well in this situation for storing user session data. Some systems have success using session databases, but the overhead of connections and queries seem excessive when other solutions are possible, such as caches for roughly the same cost in hardware and software. The key issue with session caching is to ensure that the cache hit ratio is very high; otherwise, the user experience will be awful. If the cache expires a session because it doesn’t have enough room to keep all the user sessions, the user who gets kicked out of the cache will have to log back in. As you can imagine, if this is happening 25% of the time, users will find it extremely annoying.

Three Solutions to Scaling with Sessions

There are three basic approaches to solving the complexities of scaling an application that uses session data: avoidance, decentralization, and centralization.

Avoidance

- Remove session data completely.
- Modulus users to a particular server via the code.
- Stick users on a particular server per session with session cookies from the load balancer.

Decentralization

- Store session cookies with all information in the browser's cookie.
- Store session cookies as an index to session objects in a database or file system with all the information stored there.

Centralization

- Store sessions in a centralized session cache like Memcached.
- Databases can be used as well, but are not recommended.

There are many creative methods of solving session-related complexities when scaling applications. Depending on the specific needs and parameters of your application, one or more of these approaches might work better for you than others.

Whether you decide to design your application to be stateful or stateless and whether you use session data or not are decisions that must be made on an application-by-application basis. In general, it is easier to scale applications that are stateless and sessionless. Although these choices aid in scaling, they may prove unrealistic to implement given the complexities they cause for the application development. When you do require the use of state—and in particular, session state—consider how you will scale your application on all three axes of the AKF Scale Cube before you need to do so. Scrambling to figure out the easiest or quickest way to fix a session issue across multiple servers might lead to poor long-term decisions. Crisis-driven architectural decisions should be avoided as much as possible.

Conclusion

All too often, the topic of synchronous versus asynchronous calls is overlooked when developing services or products—at least until it becomes a noticeable inhibitor to

scaling. With synchronization, two or more pieces of work must be done to accomplish a task. An example of synchronization is mutual exclusion (mutex), a process of protecting global resources from concurrently running processes, often accomplished through the use of semaphores.

Synchronous calls can become problematic for scaling for a variety of reasons. Indeed, an unsuspecting synchronous call can actually cause severe problems across the entire system. Although we don't encourage the complete elimination of these calls, we do recommend that you thoroughly understand how to convert synchronous calls to asynchronous ones. Furthermore, it is important to have individuals such as architects and managers overseeing system design to help identify when asynchronous calls are warranted.

A session is an established communication link between the client, typically the user's browser, and the server, that is maintained during the life of the session for that user. Keeping track of session data can become laborious and complex, especially when you are dealing with scaling an application on any of the axes of the AKF Scale Cube. Three broad classes of solutions—avoidance, centralization, and decentralization—may be used to resolve state and session complexities.

Inevitably, tradeoffs must be made when engineers use synchronous calls and write stateful applications. These methods engender the drawbacks of reduced scalability and the benefits of simple implementation. Most importantly, you should spend time up front discussing these tradeoffs, so your architecture can continue to scale.

Key Points

- Synchronization occurs when two or more pieces of work must be done to accomplish a task.
- Mutex is a synchronization method that defines how global resources are protected from concurrently running processes.
- Synchronous calls perform their action completely by the time the call returns.
- With an asynchronous method call, the method is called to execute in a new thread and it immediately returns control back to the thread that called it.
- The design pattern that describes the asynchronous method call is called either asynchronous design or asynchronous method invocation.
- Synchronous calls can, if used excessively or incorrectly, place an undue burden on the system and prevent it from scaling.
- Synchronous calls are simpler than asynchronous calls.
- Another part of the problem of synchronous calls is that developers typically see only a small portion of the application.

- An application that uses state is called stateful; it relies on the current state of execution to determine the next action to be performed.
- An application or protocol that doesn't use state is referred to as stateless.
- Hyper-Text Transfer Protocol (HTTP) is a stateless protocol because it doesn't need any information about the previous request to know everything necessary to fulfill the next request.
- A state machine is an abstract model of states and actions that is used to model behavior; it can be implemented in the real world in either hardware or software.
- One of the most common implementations of state is the user session.
- Choosing wisely between synchronous and asynchronous as well as stateful and stateless approaches is critical for scalable applications.
- Have discussions and make decisions early, when standards, practices, and principles can be followed.

Part IV

Solving Other Issues and Challenges

Chapter 27: Too Much Data	427
Chapter 28: Grid Computing	447
Chapter 29: Soaring in the Clouds	459
Chapter 30: Making Applications Cloud Ready	485
Chapter 31: Monitoring Applications	495
Chapter 32: Planning Data Centers	509
Chapter 33: Putting It All Together	531

Hudayfa

Det Kgl. Bibliotek

Solving Ode to Sud Ogstuenes

Chapter 57: Leo Vang's 120

Chapter 58: City Councilor

Chapter 59: Second life in a lamp

Chapter 60: A Little Different Crossroads

Chapter 61: You can't always get what you want

Chapter 62: You can't always get what you want

Chapter 63: You can't always get what you want

Chapter 64: You can't always get what you want

Chapter 65: You can't always get what you want

Chapter 66: You can't always get what you want

Chapter 67: You can't always get what you want

Chapter 68: You can't always get what you want

Chapter 69: You can't always get what you want

Chapter 70: You can't always get what you want

Chapter 71: You can't always get what you want

Chapter 72: You can't always get what you want

Chapter 73: You can't always get what you want

Chapter 74: You can't always get what you want

Chapter 75: You can't always get what you want

Chapter 27

Too Much Data

The skillful soldier does not raise a second levy, nor are his supply wagons loaded more than once.

—Sun Tzu

Hyper-growth, or even slow steady growth over time, presents unique scalability problems for our storage systems. The amount of data that we keep, and the duration for which we keep it, have significant cost implications for our business and can negatively affect our ability to scale cost-effectively

Time affects the value of our data in most systems. Although not universally true, in many systems, the value of data decreases over time. Old customer contact information, although potentially valuable, probably isn't as valuable as the most recent contact information. Old photos and videos aren't likely to be accessed as often and old log files probably aren't as relevant to us today. So as our costs increase with all of the additional data being stored, the value on a per-data-unit-stored basis decreases, presenting unique challenges for most businesses.

This chapter addresses data storage. Specifically, how do we store large volumes of data and process them, while keeping our business from being overly burdened? Which data do we get rid of, and how do we store data in a tiered fashion that allows all data to be accretive to shareholder value?

The Cost of Data

Data is costly. Your first response to this statement might be that the costs of mass storage devices have decreased steadily over time and with the introduction of cloud storage services, storage has become “nearly free.” But “free” and “nearly free” obviously aren’t the same things. A whole lot of something that is nearly free actually turns out to be quite expensive. As the price of storage decreases over time, we tend to care less about how much we use; in turn, our usage typically increases significantly. Prices might drop by 50%, but rather than passing that 50% reduction in

price off to our shareholders as a reduction in our cost of operations, we may allow the size of our storage to double because it is “cheap.”

Of course, the initial cost of this storage is not the only cost you incur with every piece of data you store on it. The more storage you have, the more storage management is needed. This might increase the overhead of systems administrators to handle the data, capacity planners to plan for the growth, or maybe even software licenses that allow you to “virtualize” your storage environment and manage it more easily. As your storage grows, so does the complexity and cost of managing that resource.

Furthermore, as your storage increases, the costs of handling that storage increase as well (“costs” here may be power costs or the cost of space to handle the storage). You might argue here that the advent of Infrastructure as a Service (IaaS) or cloud storage services eliminates many of these costs. We applaud you if you have put your infrequently accessed data on such a storage infrastructure. However, even when working with these solutions, a single massive array will still cost you less than 10 massive arrays, and less storage in the cloud will still cost you less than more storage in the cloud. If you are using cloud services, you still need the staff and processes to understand where that storage is located and to ensure it can be properly accessed.

And that’s still not all the costs! If this data resides in a database upon which you are performing transactions for end users, each query of that data increases with the size of the data being queried. We’re not talking about the cost of the physical storage at this point, but rather the time to complete the query. If you are querying upon a properly balanced index, it’s true that the time to query that data is not linear (it is more likely $\log_2 N$, where N is the number of elements). Nevertheless, this time increases with an increase in the size of the data. Sixteen elements in binary tree will not cost twice as much to traverse and find an element as eight elements—but the larger number of elements still costs more. This increase in steps to traverse data elements takes more processor time per user query, which in turn means that fewer things can be processed within any given amount of time.

To see how this works, suppose we have 8 elements and it takes us on average 1.5 steps to find our item with a query. Now suppose that with 16 elements, it takes us on average 2 steps to find our item. This is a 33% increase in processing time to handle 16 elements versus 8 elements. Although this seems like a good leverage scaling method, it’s still taking more time. This increased response time doesn’t just impact the database: Even if performed asynchronously, it likely increases the time an app server is waiting for the query to finish, the time the Web server is waiting for the app server to return data, and the time your customer is waiting for a page to load.

Consider the normal peak utilization time in the afternoon, which takes place between 1 and 2 p.m. If each query takes us 33% more time on average to complete and we want to run at 100% utilization during our peak traffic period, we might

need as many as 33% more systems to handle the additional data (16 elements versus the original 8) without impacting user response times. In other words, we either let each of the queries take 33% more time to complete and affect the user experience as new queries get backed up waiting for longer-running queries to complete given the constrained capacity, or we add capacity to limit the impact on users. At some point—without disaggregation of the data similar to the trick we performed with search in Chapter 24, Splitting Databases for Scale—user experience will inevitably begin to suffer. Although you may argue that faster processors, better caching, and faster storage will support a better user experience, none of these factors addresses the fact that having more data costs you more in processing time.

The costs of increased data aren't limited to storage overhead, increased processing time, and increased user response times. Undoubtedly, you'll also need to back up your storage from time to time. As your data grows, the work necessary to perform a "full backup" grows as well—and, of course, this work will be repeated with each full backup. While most of your data may not be changing, you are nevertheless rewriting it every time a full backup is conducted. Although incremental backups (backing up only the changed data) help alleviate this concern, you will more than likely perform a periodic full backup to forego the cost of needing to apply a multitude of incremental backups to a single full backup that might be years old. If you did only a single full backup and then relied on incremental backups alone to recover some section of your storage infrastructure, your recovery time objective (the amount of time to recover from a storage failure) would be long indeed!

With this discussion, we hope we've disabused you of the notion that storage is free. Storage prices may be falling, but they represent only one portion of your true cost to store information, data, and knowledge.

The Six Costs of Data

As the amount of data that you store increases, the following costs increase:

- Storage resources in which to store the data
- People and software to manage the storage
- Power and space to make the storage work
- Capital to ensure the proper power infrastructure
- Processing power to traverse the data
- Backup time and costs

Data costs don't consist of just the costs of the physical storage; sometimes the other costs identified here eclipse the actual storage costs.

The Value of Data and the Cost-Value Dilemma

All data is not created equally in terms of its value to our business. In many businesses, time negatively impacts the value that we can get from any specific data element. For instance, old data in most data warehouses is less likely to be useful in modeling business transactions. Old data regarding a given customer's interactions with your ecommerce platform might be useful to you, but it's not likely to be as useful as the most current data that you have. Detailed call records for the phone company from years ago aren't as valuable to the users as new call records, and old banking transactions from three years ago probably aren't as useful as the ones that occurred in the last few weeks. Old photos and videos might be referenced from time to time, but they aren't likely to be accessed as often as the most recent uploads. Although we won't argue that older data is *always* less valuable than new data, it holds true often enough to call it generally true and directionally correct.

If the value of data decreases over time, why do we keep so darn much of it? We call this question the *cost-value data dilemma*. In our experience, most companies simply do not pay attention to the deteriorating value of data and the increasing cost of maintaining increasing amounts of data over time. Often, new or faster technologies allow us to store the same data for a lower initial cost or store more data for the same cost. As unit cost of storage drops, our willingness to keep more of it increases.

Moreover, many companies point to the *option value* of data. How can you possibly know for what you might use that data in the future? Almost all of us can remember a point in our careers when we have said, "If only we had kept that data." The lack of a critical element of data one time often becomes a reason to keep all data for all time.

Another reason commonly cited for excessive data retention is *strategic competitive advantage*. Often, this reason is couched as, "We keep this data because our competition doesn't keep it." It is entirely possible that your data is a source of competitive advantage, though our experience suggests that the value of keeping data infinitely is not as much of an advantage as simply keeping it longer than your competition (but not infinitely).

Ignoring the cost-value data dilemma, citing the option value of data, and claiming competitive advantage through infinite data retention—all have potentially dilutive effects on shareholder value. If the real upside of the decisions (or lack of decisions, in the case of ignoring the dilemma) does not create more value than the cost, the decision is dilutive to shareholders. In the cases where legislation or a regulation requires you to retain data, such as emails or financial transactions, you have little choice but to comply with the letter of the law. In all other cases, however, it is possible to assign some real or perceived value to the data and compare it to the costs. Consider the fact that the value is likely to decrease over time and that the

costs of data retention, although declining on a per-unit basis, will likely increase in aggregate value in hyper-growth companies.

As a real-world analog, your company may be mature enough to associate a certain value and cost to a class of user. Business schools often spend a great deal of time discussing the concept of *unprofitable customers*. An unprofitable customer is one that costs you more to keep than it represents in revenue, inclusive of all referral business the customer generates. Ideally, you do not want to service or keep your unprofitable customers. The science and art of determining and pruning unprofitable customers is more difficult in some businesses than others.

The same concept of profitable and unprofitable customers nevertheless applies to your data. In nearly any environment, with enough investigation, you will likely find data that adds shareholder value and data that is dilutive to shareholder value. Just as some customers may be unprofitable to us as a business (customers that cost us more than we make from them in revenue), so some data may cost us more than it represents in value.

Making Data Profitable

The business and technology approach to deciding which data to keep and how to keep it is relatively straightforward: Architect storage solutions that allow you to keep all data that is profitable for your business, or is likely to be accretive to shareholder value, and simply remove the rest. Let's look at the most common factors driving data bloat and then examine ways to match our data storage costs to the value of the data.

Option Value

All options have some value to us. The precise value may be determined by what we believe the probability is that we will ultimately execute the option to our personal benefit. This may be a probabilistic equation that calculates both the possibility that the option will be executed and the likely benefit of the value of executing the option. Clearly, we cannot claim that the value of any option is infinite. Such a claim would suggest infinite shareholder wealth creation, which simply isn't possible.

The option value of our data, then, is some bounded (i.e., non-infinite) number. To suggest a bound on this number, we should start asking ourselves pertinent questions: "How often have we used data in the past to make a valuable decision?" "What was the age of the data used in that decision?" "What was the value that we ultimately created versus the cost of maintaining that data?" "Was the net result profitable?"

These questions aren't meant to advocate the removal of all data from your systems. Your platform probably wouldn't work if it didn't have some meaningful data in it. Rather, we are simply indicating that you should evaluate and question your data retention policy to ensure that all of the data you are keeping is, in fact, valuable. If you haven't used the data in the past to make better decisions, there is a good chance that you won't start using all of it tomorrow. Even when you do start using your data, you aren't likely to use all of it. As such, you should decide which data has real value, which data has value but should be stored in a lower-cost storage solution, and which data can be removed.

Strategic Competitive Differentiation

"Strategic competitive differentiation" is one of our favorite reasons to keep data. It's the easiest to claim and the hardest to disprove. The general thought is that keeping massive amounts of data (more than your competitors) can competitively differentiate a company. Ostensibly, the company can then make better decisions, and customers will have access to more and better data.

In most cases, however, the value of data degrades over time and infinite data does not equate to infinite value. These two ideas converge to highlight a decay curve in data, where the value of older data starts to decay significantly. As such, to address competitive differentiation, we need to understand the value of data at year Y versus the values at year $Y - 2$, $Y - 5$, and so on. We need to identify the point at which data is no longer profitable as well as the point at which additional data adds near-zero value to customer retention, better decision making, and so on. These two points may be the same, but most likely they are not.

After we recognize that some data has immense value, some data has lower value, some data "might have value," and some data has no value at all, we can determine a tiered cost storage solution for data with value and remove the data with very low or no value. We can also transform and compact the data to make sure that we retain most of the value at significantly lower costs.

Cost-Justify the Solution (Tiered Storage Solutions)

Let's assume that a company determines that some of its data has real value, but the cost of storing it makes the data unprofitable. This is the time to consider a tiered storage solution. Many companies settle on a certain type of storage based on the primary needs of their transaction processing systems. The result of this decision is that almost everything else relies upon this (typically) premium storage solution—even though not absolutely everything needs the redundancy, high availability, and response of your primary applications. For your lower-value (but nevertheless valuable) services and needs, consider moving the data to tiered storage solutions.

For instance, infrequently accessed data that does not require immediate response times might be provisioned on a slower, less costly, and less power-consuming storage solution. Another option is to split up your architecture to serve some of these data needs from a *y-axis* split that addresses the function of “serve archived data.” To conserve processing power, perhaps the requests to “serve archived data” might be made in an asynchronous fashion and emailed after the results are compiled.

A number of other options for reducing costs through tiered solutions are also available. Infrequently accessed customer data can be put on cloud storage systems. Data that is old and does not change (such as order history data) can be removed from databases and stored statically. The older such data gets, the less frequently it is accessed; thus, over time, it can be moved to lower and lower tiers of storage.

The solution here is to match the cost (or cost-justify) the solution with the value that it creates. Not every system or piece of data offers the same value to the business. We typically pay our employees based on their merit or value to the business, so why shouldn’t we approach systems design in the same fashion? If there is some, but not much, value in some group of data, simply build the system to support the value. This approach does have some drawbacks, such as the requirement that the product staff support and maintain multiple storage tiers, but as long as those additional costs are evaluated properly, the tiered storage solution works well for many companies.

Transform the Data

Often, the data we keep for transactional purposes simply isn’t in a form that is consumable or meaningful for our other needs. As a result, we end up processing the data in near real time to make it meaningful to corporate decision making or to make it useful to our product and platform for a better customer experience.

As an example of our former case, where we are emphasizing making good business decisions, consider the needs of a marketing organization concerned about individual consumer behavior. Our marketing organization might be interested in demographic analysis of purchases over time of a number of our products. Keeping the exact records of every purchase might be the most flexible approach to fulfill those needs, but the marketing organization is probably comfortable with being able to match buyer purchases of products by month. All of a sudden, our data requirements have shrunk: Because many of our customers are repeat purchasers, we can collapse individual transaction records into records indicating the buyer, the items purchased, and the month in which those items were purchased. Now, we might keep online transaction details for four months to facilitate the most recent quarterly reporting needs, and then roll up those transactions into summary transactions by individual for marketing purposes and by internal department for finance reasons.

Our data storage requirements might decrease by as much as 50% with this scheme. Furthermore, as we would otherwise perform this summarization during the time of the marketing request, we have reduced the response time of the application generating this data (it is now prepopulated), thereby increasing the efficiency of our marketing organization.

As an example of data processing intended to provide a better customer experience, we might want to make product recommendations to our customers while they are interacting with our platform. These product recommendations might give insight as to what other customers bought who have viewed or purchased similar items. It goes without saying that scanning all purchases to develop such a customer affinity-to-product map would likely be too complex to calculate and present while someone is attempting to shop. For this reason alone, we would want to precalculate the product and customer relationships. However, such calculation also reduces our need to store the details of all transactions over time. As a result in developing our precalculated affinity map, we have not only reduced response times for our customers, but also reduced some of our long-term data retention needs.

The principles on which data transformations are based are couched within a process that data warehousing experts refer to as “extract, transform, and load” (ETL). It is beyond the scope of this book to even attempt to scratch the surface of data warehousing, but the concepts inherent to ETL can help obviate the need for storing larger amounts of data within your transactional systems. Ideally, these ETL processes, besides removing the data from your primary transaction systems, will reduce your overall storage needs as compared to keeping the raw data over similar time periods. Condensing expensive detailed records into summary tables and fact tables focused on answering specific questions helps save both space and processing time.

Handling Large Amounts of Data

Having discussed the need to match storage costs with data value and to eliminate data of very low value, let’s now turn our attention to a more exciting problem: What do we do when our data is valuable but there is just far too much of it to process efficiently? The answer is the same as it is in complex mathematical equations: Simplify and reduce the equation.

If the data is easily segmented into resources or can be easily associated with services, we need simply apply the concepts we learned in Chapters 22 through 24. The AKF Scale Cube will solve your problems in these situations. But what about the case when an entire data set needs to be traversed to produce a single answer, such as the count of all words in all of the works contained within the Library of Congress, or potentially an inventory count within a very large and complex inventory

system? If we want to complete this work quickly, we need to find a way to distribute the work efficiently. This distribution of work might take the form of a multiple-pass system where the first pass analyzes (*maps*) the work and the second pass calculates (*reduces*) the work. Google introduced a software framework to support distributed processing of just such large data sets, called MapReduce.¹ The following is a description of that model and an example of how it can be applied to large problems.

At a high level, MapReduce has a Map function and a Reduce function. The Map function takes as its input a key-value pair and produces an intermediate key-value pair. This might not immediately seem useful to the layperson, but the intent is that the distributed process creates useful intermediate information for another distributed process to compile. The input key might be the name of a document or pointer to a subsection of a document. The value could be content consisting of all the words within the document or subsection itself. In our distributed inventory system, the key might be the inventory location and the value all of the names of inventory within that location with one name for each piece and quantity of inventory. For instance, if we had five screws and two nails, the value would be screw, screw, screw, screw, screw, and nail, nail.

The canonical form of Map looks like this in pseudocode:²

```
map(String input_key, String input_value):
// input_key: document name or inventory location name
//input_value: document contents or inventory contents
For each word w (or part p) in input_value:
    EmitIntermediate(w, "1") (or EmitIntermediate(p,"1"));
```

We've identified parenthetically that this pseudocode could work for both the word count example (also given by Google) and the distributed parts inventory example. Only one or the other would exist in reality for your application and you would eliminate the parenthesis.

Some *input_key* and *input_values* and output keys and values are presented in Figure 27.1. The first example is a set of phrases including the word “red” of which we are fond, and a small set of inventories for different locations.

Note here how Map takes each of the documents and simply emits each word with a count of 1 as we move through the document. For the sake of speed, we had a separate Map process working on each of the documents. Figure 27.2 shows the output of this process.

1. Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” <http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf>.

2. Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” <http://research.google.com/archive/mapreduce-osdi04-slides/>.

Document: Red

Red Hair
Red Shoes
Red Dress

Document: Yellow

Blonde Hair
Yellow Shoes
Yellow Dress

Document: Other

Red Fence
Blue Dress
Black Shoes

Outputs:

Red 1, Hair 1, Red 1,
Shoes 1, Red 1, Dress 1

Blonde 1, Hair 1, Yellow 1,
Shoes 1, Yellow 1, Dress 1

Red 1, Fence 1, Blue 1,
Dress 1, Black 1, Shoes 1

Figure 27.1 Input and Output Key–Value Pairs for Three Documents

Location: San Francisco

Nail, Screw, Pin, Nail

Location: New York

Screw, Nail, Pin, Pin

Location: Phoenix

Pin, Screw, Nail,
Screw

Outputs:

Nail 1, Screw 1, Pin 1, Nail 1 Screw 1, Nail 1, Pin 1, Pin 1 Pin 1, Screw 1, Nail 1, Screw 1

Figure 27.2 Input and Output Key–Value Pairs for Inventory in Different Locations

Again, we have taken each of our initial key–value pairs, where the key is the location of the inventory and the value is the individual components listed with one listing for each occurrence of that component per location. The output is the name of the component and a value of 1 per each component listing. To create this output, we used separate Map processes.

What is the value of such a construct? We can now feed these key–value pairs into a distributed process that will combine them and create an ordered result of key–value pairs, where the value is the number of items that we have of each type (either a word or a part). The trick in our distributed system is to ensure that each key is routed to one and only one collector or *reducer*. We need this affinity to a reducer

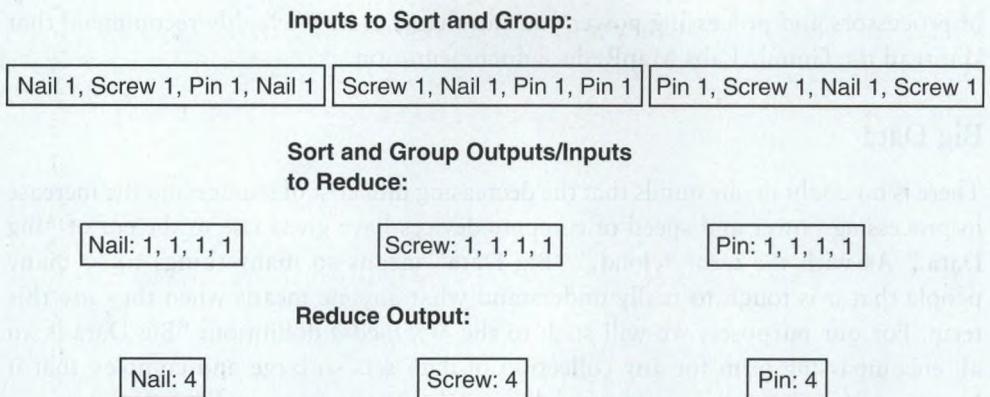


Figure 27.3 Reduce Output for Inventory System

(or tier of reducers, as we will discuss in a minute) to ensure an accurate account. If the part “screw” will go to reducer 1, all instances of “screw” must go to reducer 1. Let’s see how the Google reduce function works in pseudocode:³

```
reduce(String input_key, Iterator intermediate_values):
// output_key: a word or a part name
//output_values: count
For each v in intermediate_values:
    Result += ParseInt(v);
    Emit(AsString(result));
```

For our reduce function to work, we need to add a program to group the words or parts and append the values for each in a list. This is a rather trivial program that will sort and group the functions by key. It, too, could be distributed, assuming that the key-value pairs emitted from the map function are sent to the same function intended to sort and group and then submit to the reduce function. Passing over the trivial function of sorting and grouping, which is the subject of many computer science undergraduate textbooks, we can display our reduce function as in Figure 27.3 for our inventory system (we will leave the word count output as an exercise for our readers).

Multiple layers of sorting, grouping, and reducing can be employed to help speed up this process. For instance, if we had 50-map systems, they could send their results to 50 sorters, which could in turn send their results to 25 sorters and groupers, and so on, until we had a single sorted and grouped list of parts and value lists to send to our multiple reducer functions. Such a system is highly scalable in terms of the amount

3. Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” Slide 4. <http://labs.google.com/papers/mapreduce OSDI04-slides/>.

of processors and processing power you can throw at it. We highly recommend that you read the Google Labs MapReduce documentation.

Big Data

There is no doubt in our minds that the decreasing unit cost of storage and the increase in processing power and speed of compute devices have given rise to the era of “Big Data.” As with the term “cloud,” “Big Data” means so many things to so many people that it is tough to really understand what anyone means when they use this term. For our purposes, we will stick to the *Wikipedia* definition: “Big Data is an all-encompassing term for any collection of data sets so large and complex that it becomes difficult to process using traditional data processing applications.”⁴

While the vast majority of this book has been focused on transaction-processing products that face clients, the concepts we’ve presented here are nevertheless equally applicable to Big Data. In fact, the MapReduce functionality described previously was implemented in some of the earliest “Big Data” products to hit the Internet. Is there a larger “Big Data” problem than trying to gather a response set for a search query that spans the entire Internet? In fact, MapReduce serves as part of the backbone for some of the most successful NoSQL implementations (e.g., Hadoop, Mongo) to date. Moreover, few things have helped create value within the Big Data movement like the NoSQL movement.

Many books, both technical and business, have been written on the topic of Big Data. Rather than try (and fail) to create a meaningful contribution to the extant literature, we will instead focus on two themes and how they can help in your Big Data endeavors. These themes are partitioning of data along the *y*- and *z*-axes of our cube of scale.

The *y*-axis partitioning approach is useful when we want to investigate deep and meaningful relationships between small “vertical” groups of data. Such segmentation can help identify “why” a relationship exists. One such grouping might be “customers and products,” which contains all of our customer data and all of our product data and which we use to explain why a product sells better in the Northeast than the Southwest. By looking at customer and product attributes, we can try to identify the independent variables (should they exist) that are most closely correlated with the phenomenon. Because we have excluded all other data, response times will be faster than if we used a monolithic online analytical processing (OLAP) solution.

By comparison, *z*-axis segmentation is useful when we want to do broad analysis across horizontal slices of data to try to identify where relationships exist. These types of analysis tend to be exploratory in nature, attempting to identify which

4. “Big Data.” *Wikipedia*. http://en.wikipedia.org/wiki/Big_data; accessed September 17, 2014.

relationships exist rather than why those relationships exist. The data is segmented to include all attributes, but for only a subset of the total population of data, which allows us to perform broad queries at a reduced response time relative to a monolithic system.

The astute reader may point out that many NoSQL options will do some of this work, especially the *z*-axis splits. While many do offer this functionality, they can perform such analysis on extremely large data sets only if they are provisioned properly. In other words, you must either rent a lot of capacity in the cloud or spend a lot of capital to answer all of your questions with fast response times across all of your data. If you subject a single environment to all of the questions that you want answered from your data, you will either go bankrupt feeding your cash to that environment or wait forever for responses.

Our preferred approach is to think in terms of three environments: a monolith that has all data, but is not meant to process a lot of queries; a *z*-axis split of 1/Nth of all of the data that is meant for rapid query responses; and a configurable *y*-axis environment that we can load with various deep data sets or slices for confirmation of the *z*-axis findings. We can then follow a progression of identifying relationships (induction) in the *z*-axis solution, validating existence across the entire customer base in the *y*-axis solution, and finalizing the findings in the monolithic solution. Note that each of these steps can be either a relational or NoSQL solution. That said, nonrelational systems seem intuitively best suited to answer questions about the relationships between objects, as they have fewer relational boundaries. Figure 27.4 depicts the three-environment solution discussed here.

By splitting our data into three data sets, we can allow the monolithic data set to be driven primarily by data storage needs. This reduces its overall cost relative to a solution that must also handle all of our queries. We use this data set to feed the “fast and broad” (*z*-axis) and “deep and narrow” (*y*-axis) data sets. This scheme also serves to perform final validation of anything that we find.

The *z*-axis split allows us to quickly find relationships on an indiscriminate cross section of our data. This keeps both query processing node needs and data node needs in check, as data sets are smaller. With this solution, we can rapidly iterate through ideas, peruse data to find potential meaning (induction), and queue these potential relationships up for proof in the *y*-axis. Operationally, should we need to use this data in real time in our products, the *z*-axis split (by customer) can also exist in each swim lane. The fast and broad data set is populated by an extract, transform, and load (ETL) program. This program can operate either in batch mode or using a real-time but asynchronous “trickle load” from either the monolith or the product.

The *y*-axis split allows us to find causation for relationships and perform a primary validation of those relationships. It scales by both processing needs and data needs, but because it is constrained it should not be as costly as attempting to scale the monolith. It, too, is fed by an ETL program.

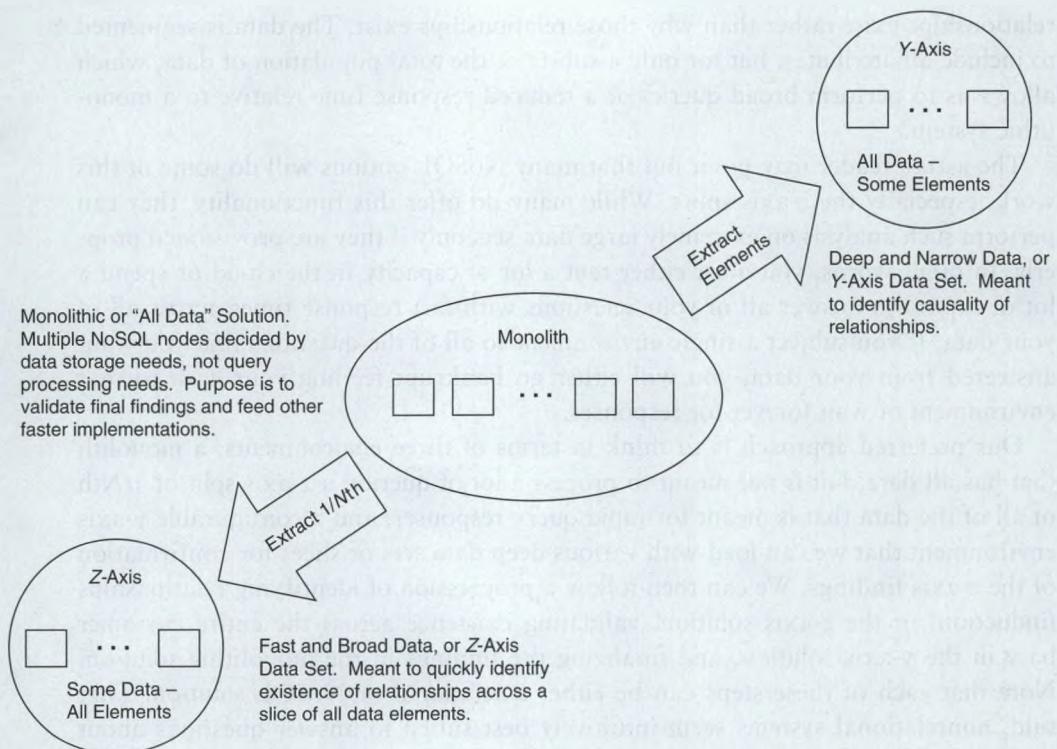


Figure 27.4 *y- and z-Axis Big Data Splits*

Using this approach, or a variant, helps us not only meet our real-time needs in getting answers to our questions, but also keep overall costs down. Because the systems scale independently and are based on individual variables, the costs tend to be lower and overall response times faster than with a single monolith.

A NoSQL Primer

No book on scalability would be complete without at least a nod to NoSQL. As with Big Data, many books have been written on the various flavors of NoSQL and a great deal of information is available from sources like Gartner and its kin. Rather than attempt to do a poor job at re-creating all of this work, we offer some thoughts on how the various families of NoSQL solutions compare and when you might use them. Our intent here is to provide structure to the NoSQL universe to assist you in applying this technology.

The NoSQL revolution was, at least in part, born of the insights derived from Brewer's theorem (see Chapter 24, Splitting Databases for Scale). They tend to

eschew—completely or at least in part—many of the aspects of Structured Query Language (SQL) and relax the consistency constraint within the ACID properties of databases (see Chapters 24 and 25 for definitions and discussions of these). These changes allow for lower costs of scale and better “worst case” response times for queries that are unstructured in nature. In many cases, NoSQL solutions also offer better “best case” and “average case” response times for precomputed cache implementations (refer to the discussion of object caches in Chapter 25, Caching for Performance and Scale). In addition, the relaxation of consistency allows for easier distribution of data to support disaster recovery, high availability, and proximity to customers through native replication capabilities. The general principles we apply in deciding which tool to use (SQL/relational or NoSQL/nonrelational systems) focus on the structure of the data and the purpose of retrieval.

When products rely on highly structured data with a limited and predefined set of operations on those data, relational databases are appropriate. SKU-based product catalog systems are a great example of such a solution. In this approach, the business imposes relationships upon objects, such as a hierarchical structure that adds meaning to items (soap and hair products may be under a personal hygiene hierarchy). Locations within warehouses may also be important as a structural component of such a system. Operations such as purchase, stock, and pick for shipping are all fairly well defined. Indexes can be efficiently created to minimize operations on the data such as retrieval, updates, insertions, and deletion.

When solutions or products rely on lightly structured and nonhierarchical models, or when the operations and relationships between data elements are not well known, NoSQL solutions excel. An example is fast retrieval object caches, where we need only match data elements to users or operations. Another great example is Big Data or deep analytic systems, where we ask questions about relationships between objects that are not well known. Perhaps we are trying to answer the question of whether user location (geographic), seasonal temperature averages, and shipping costs all have an effect on varietal and quantity of wine purchase by season. Relational systems may superimpose relationships upon these entities that confound this analysis, or at the very least make it laborious and slow in response time. The appropriate NoSQL solution would not have these relationships, so it may be both easier to query and faster to respond.

We like to think of NoSQL solutions as being classified into four broad categories: key-value stores (such as Memcached from Chapter 25), column stores, document stores, and graph databases. Technically speaking, graph databases aren’t a NoSQL solution, but they often get thrown into conversations regarding relational database alternatives so we’ve included them here. These four families of NoSQL solutions scale differently in terms of cost and, accordingly, offer varying degrees of flexibility in the query operations that they support. Figure 27.5 graphs these families against cost of scale and query flexibility.

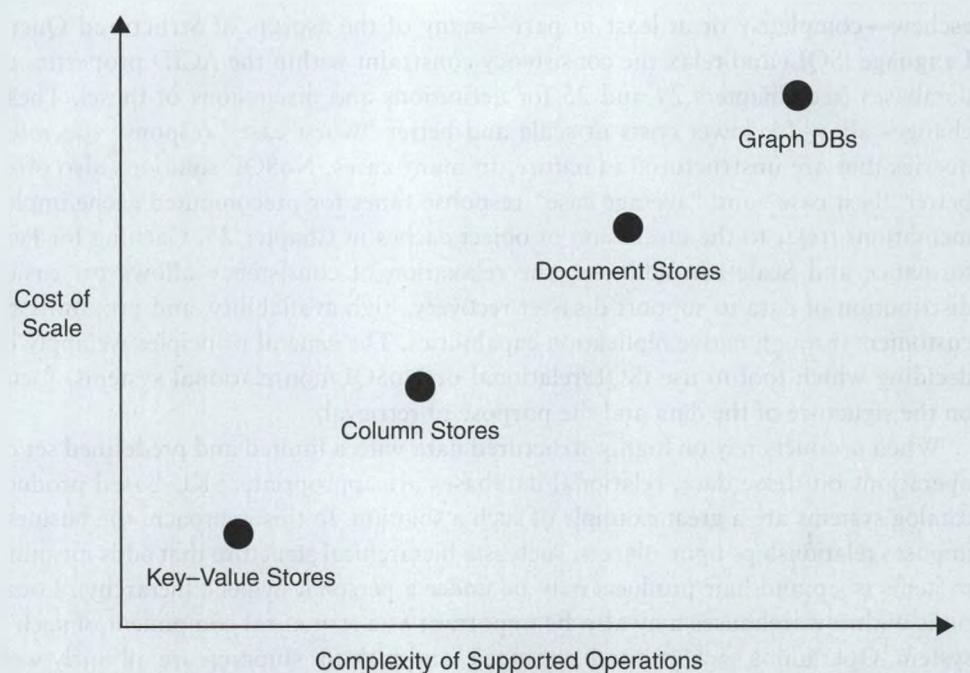


Figure 27.5 NoSQL Types Plotted Against Cost of Scale and Flexibility of Query Operations

Key-value stores allow for incredible low-cost growth (low cost of scale) given the simplicity of the data that they keep. These solutions typically map a single key to a single value. The operations set deployed against this data tends to be simple, consisting of adding data (set), updating data (replace), and retrieving data (get). The low cost of scale, limitations on query complexity, and resulting speed of use make key-value stores ideal choices for queues, caches, and distributed object stores for fast lookups. They are not meant for most Big Data or analytics purposes.

Column stores are an evolution of the key-value system. These systems tend to have multiple attributes assigned to any key, similar to what you would find in a relational database table. While the name implies a columnar-only format, some implementations use rows rather than columns. Typically a key-value pair is used to map to a column within the solution. This column holds the remaining attributes of the item being mapped. The structure allows for a greater amount of data to be kept while still minimizing the cost of scale. Query flexibility is increased, as more items are available and associated to keys. Column stores, therefore, are great solutions for keeping massive amounts of comparatively nonvolatile information. Examples might be tracking multiple aspects of users for the purposes of retargeting, important

customer messages, and semi-static associations within social networks. These solutions may be appropriate as fast-read z-axis implementations (as described in the Big Data discussion), but are limited in the depth of data they can provide and the complexity of the query formations they can support—properties that are important for many deep analytics implementations.

Document stores are perhaps the most flexible solutions and, in turn, the most costly per unit of scale within the NoSQL family. Document stores tend to have the loosest relationships between data, making them very good choices for unstructured analytics and analysis of relationships between objects. Many are purpose built for specific types of objects, such as JavaScript Object Notation (JSON). Document stores tend to have the greatest flexibility (and associated complexity) in the types of queries they allow against document objects. For this reason, they are good choices for y-axis deep analytic data marts and monolithic feeder/validation solutions (like those described in the Big Data section). Given the affinity with JSON, this family is also JavaScript friendly and is often used in product solutions.

Graph databases are inherently more relational than true NoSQL solutions. As such, they tend to provide the greatest flexibility in query operations and, in turn, come with a higher cost of scale. They excel at quickly traversing relationships between entities, such as traversing nodes within a social network. These solutions excel at evaluating real-time models between entities, such as dynamic pricing calculators in real-time pricing systems (e.g., second price auctions).

Figure 27.6 maps the various NoSQL implementations into their respective families. Note that because some implementations are difficult to classify, we've mapped them across familial boundaries.

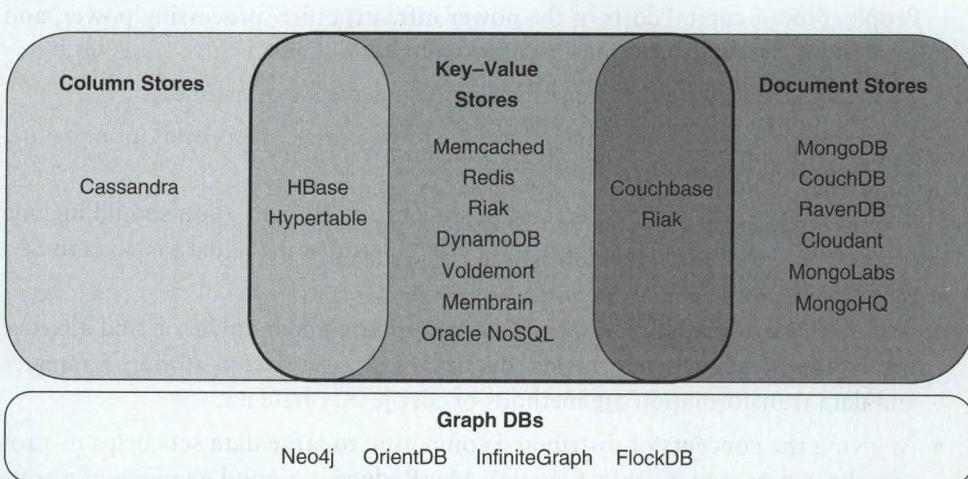


Figure 27.6 NoSQL Types Plotted Against Family

Conclusion

This chapter discussed the storage and processing of large data sets. We examined the paradoxical relationship of cost and value for data. As data ages and data size grows, its cost to the organization increases. At the same time, as data ages in most companies, its value to the company and platform typically decreases. The reasons for clinging to data whose value has largely expired include ignorance, perceived option value, and perceived strategic competitive differentiation. Our remedies for perceived option value and perceived competitive differentiation are based in the notion of applying real dollar values to these perceived values to properly justify the existence (and the cost) of the data.

After identifying the value and costs of data, you should consider implementing tiered storage solutions that match the cost and access speed of data to the value that it creates for your shareholders. On one end of such a tiered strategy are high-end, very fast storage devices; on the opposite end is the deletion or purging of low-value data. Data transformation and summarization can help reduce the cost, and thereby increase the profitability, of data where the reduction in size does not significantly change the value of the data.

One approach to parallelize the processing of very large data sets is Google's MapReduce approach. MapReduce has been widely adopted by many industries as a standard for processing large data sets quickly in a distributed fashion.

Key Points

- Data is costly, and its cost includes more than just the cost of the storage itself. People, power, capital costs of the power infrastructure, processing power, and backup time and costs all influence the true cost of data.
- The value of data in most companies tends to decrease over time.
- Companies often keep too much data due to ignorance, perceived option value, and perceived competitive differentiation.
- Perceived option value and perceived competitive differentiation should include values and time limits on data to properly determine if the data is accretive or dilutive to shareholder value.
- You should eliminate data that is dilutive to shareholder value, or find alternative storage approaches to make this data accretive. Tiered storage strategies and data transformation are methods of cost-justifying data.
- Applying the concepts of distributed computing to large data sets helps us process those data sets quickly. Google's MapReduce is a good example of a software framework for acting on large data sets.

- Leveraging the *y*- and *z*-axes of scalability can help to identify relationships faster (*z*-axis), determine causation of the relationships (*y*-axis), and validate results across the entire data set (monolith) at relatively low cost.
- NoSQL solutions relax the ACID component of consistency to speed up some queries and allow for easier distribution of data.
- The appropriate NoSQL solution can aid in analytics, as the lack of structure doesn't confound or impede broad analytical queries across objects.
- Multiple families of NoSQL solutions exist, each with benefits and comparative drawbacks. Choose the right tool for your specific need.

Chapter 28

Grid Computing

We cannot enter into alliances until we are acquainted with the designs of our neighbors.

—Sun Tzu

Grid computing offers the scaling of demand for computing cycles by computationally intense applications or programs. When you understand the pros and cons of grid computing and have some ideas about how this type of technology might be used, you will be well armed to use this knowledge in your scalability efforts.

Grid computing allows an application to use significant computational resources to process data or solve problems more rapidly. Parallel processing is a core component of scaling, analogous to the x -, y -, and z -axis splits in the AKF Scale Cube. Depending on how the separation of processing is done or viewed, the splitting of the application for grid computing might take the shape of one or more of the axes.

History of Grid Computing

Grid computing as a concept has been around for almost two decades. It describes the use of two or more computers to process individual parts of an overall task. The tasks that are best structured for these types of solutions are ones that are computationally intensive and divisible into smaller tasks. Ian Foster and Carl Kesselman are credited as the fathers of the grid from their book *The Grid: Blueprint for a New Computing Infrastructure* (1998, Morgan Kaufmann Publishers).¹ These two individuals, along with others, also were instrumental in developing Globus Toolkit—an open source product supported by Globus Alliance that is considered the de facto standard for building grids.

1. As declared in the April 2004 issue of *University of Chicago Magazine*. <http://magazine.uchicago.edu/0404/features/index.shtml>.

For tasks and jobs to be performed in a grid environment, software must be used to orchestrate the division of labor, monitor the computation of subtasks, and aggregate completed jobs. This type of processing can be thought of as parallel processing that occurs on a network distributed basis. Prior to the concept of grid computing, the only way to achieve this scale of parallel processing was to use a mainframe computer system. Modern grids are often composed of many thousands of nodes across public or private networks of computers.

Public networks range from volunteers on the Internet allowing the use of their computers' unused CPU clock cycles to pay-for-usage grids, such as the Sun Grid launched in February 2005. Private networks include dedicated farms of small commodity servers equipped with grid middleware that allows for parallel computing. Other private networks are found in corporate offices, where personal computers are used after hours for parallel computing. One of the most well-known public network examples of grid computing is the SETI@home project, which uses computers connected to the Internet in the search for extraterrestrial intelligence (SETI). Individuals can participate by running a program on their personal computers that downloads and analyzes radio telescope data. The program then sends the results back to the central system that aggregates completed jobs. This type of public network grid computing is known as CPU scavenging.

As mentioned previously, several middleware providers offer support for building grid systems. One of the earliest to emerge on the scene was the Globus Toolkit, but others include the Sun Grid Engine and UNICORE (UNiform Interface to COnputing REsources). Many of these middleware products are applied in universities, government, and industry to create grids.

With all this processing power available, how fast can our applications run? Similar in concept to Brooks' law from the book *The Mythical Man-Month* by Frederick P. Brooks, Jr., applications can be divided only so many times. Although adding more processors to the job probably won't make it actually slow down, as projects do when more engineers are added late within a project, such moves do nothing to help the system accomplish the job faster.

Amdahl's law, developed by Gene Amdahl in 1967, states that the portion of a program that cannot be parallelized will limit the total speedup from parallelization.² Stated another way, the nonsequential parts of a program benefit from parallelization, but the rest of the program does not. There actually was no formula for Amdahl's law provided in the original paper, but it can be approximated, as shown in Figure 28.1, by the total improvement, a factor of the original run time, equaling the inverse of the sequential parts of the program. Thus, if 75% of the program

2. G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities." In AFIPS Conference Proceedings, Vol. 30 (Atlantic City, NJ, April 18–20). Reston, VA: AFIPS Press, 1967:483–485.

$$\text{Improvement} = \frac{1}{s}$$

Figure 28.1 *Amdahl's Law*

can be parallelized, leaving 25% that must be executed sequentially, we should see approximately a 4 \times increase in the total run time of the application by running it in a parallel computing environment.

As you might expect, there are some problems associated with running applications and programs on grid systems. This is especially true on public networks, but this scheme can also be problematic on private networks. One of the biggest problems is how to handle the data that the program requires to execute. Storage management of data, security of the data, and transmission of the data are all areas of concern when dealing with network distributed parallel processing or grid computing. Usually, the amount of data that is sent along with the executable program is small enough that by itself it is neither valuable nor sensitive. Nevertheless, the perceived risk of sending data to computer systems that you do not control can be a tough sell, especially when processing sensitive information such as personally identifiable information (PII).

Pros and Cons of Grids

Grid environments are ideal for applications that need computationally intensive environments and for applications that can be divided into elements that are then simultaneously executed. With that as a basis, we will discuss the benefits and drawbacks of grid computing environments. The pros and cons will be weighed differently by different organizations. If your application can be divided easily, either by luck or by design, you might not care that the only way to achieve great benefits is with applications that can be divided. However, if you have a monolithic application, this drawback may be so significant as to completely discount the use of a grid environment. As we discuss each of the pros and cons, keep in mind that some will matter more or less depending on your organization.

Pros of Grids

The advantages of grid computing models include high computational rates, shared infrastructure, utilization of unused capacity, and cost. Each of these is explained in more detail next. The ability to scale up computation cycles quickly as necessary for processing is obviously directly applicable to scaling an application, service, or program. In terms of scalability, it is important to grow the computational capacity—but it is equally important to do so efficiently and cost-effectively.

High Computational Rates

One obvious benefit of parallelized processing is the high computational rates made possible by this approach. Grid computing infrastructure is designed for applications that need computationally intensive environments. The combination of multiple hosts with software for dividing tasks and data allows for the simultaneous execution of multiple tasks. The amount of parallelization is limited by the hosts available—the amount of division possible within the application and, in extreme cases, the network linking everything together. Amdahl's law defines the upper bound of this benefit from the perspective of the application.

Shared Infrastructure

The second benefit of grid computing derives from the use of shared infrastructure. Most applications that utilize grid computing do so either daily, weekly, or on some other periodic basis. Outside of the periods in which the computing infrastructure is used for grid computing purposes, it can be utilized by other applications or technology organizations. We will discuss the limitations of sharing the infrastructure simultaneously in the "Cons of Grids" section.

This benefit is focused on sharing the infrastructure sequentially. Whether they form a private or public grid, the host computers in the grid can be utilized almost continuously around the clock. Of course, this requires proper scheduling of jobs within the overall grid system so that as one application completes processing, the next one can begin. It also requires either that applications be flexible in the times that they run or that they can be stopped in the middle of a job and delayed until capacity is freed up later in the day. If an application must run every day at 1 a.m., the job before it must complete prior to this time or be designed to stop in the middle of the processing and restart later without losing valuable computations. For anyone familiar with job scheduling on mainframes, this practice should sound familiar, because (as we mentioned earlier) the mainframe was the only way to achieve such intensive parallel processing before grid computing.

Utilization of Unused Capacity

The third benefit that we see in some grid computing implementations is the utilization of unused capacity. Grid computing implementations vary, such that some are wholly dedicated to grid computing all day, whereas others are utilized as other types of computers during the day and connected to the grid at night when no one is using them. When grids rely on surplus capacity, the approach is known as CPU scavenging. One of the most well-known grid scavenging programs is SETI@home, which utilizes unused CPU cycles on volunteers' computers in a search for extraterrestrial intelligence as detected in radio telescope data.

There are obviously drawbacks to utilizing spare capacity—for example, the unpredictability in the number of hosts available and the speed or capacity of each host. When dealing with large corporate computer networks or standardized systems that are idle during the evening, these drawbacks are minimized.

Cost

A fourth potential benefit of grid computing is cost. Users can realize the benefit of scaling efficiently in a grid, as this approach takes advantage of the distributed nature of applications. This strategy can be thought of in terms of scaling the *y*-axis, as discussed in Chapter 23, Splitting Applications for Scale, and shown in Figure 23.1. As one service or particular computation has more demand placed on it, instead of scaling the entire application or suite of services along an *x*-axis (horizontal duplication), you can be much more specific and scale only the service or computation that requires the growth. In turn, you can spend your funds much more efficiently, on only the capacity that is truly necessary.

The other advantage in terms of cost can come from scavenging spare cycles on desktops or other servers, as described in regard to the SETI@home program.

Pros of Grid Computing

Here, we identify four major benefits of grid computing. These are listed in no particular order and are not all-inclusive. Many more benefits exist, but the four outlined here are representative of the types of benefits you could expect from including grid computing in your infrastructure.

- *High computation rates.* With the amalgamation of multiple hosts on a network, an application can achieve very high computational rates or computational throughput.
- *Shared infrastructure.* Although grids are not necessarily great infrastructure components to share with other applications simultaneously, they are generally not used around the clock and can be shared by applications sequentially.
- *Unused capacity.* When it utilizes unused hosts during off hours, the grid offers a great use for this untapped capacity. Personal computers are not the only untapped capacity. For example, testing environments may not be utilized during the late evening hours and can be integrated into a grid computing system.
- *Cost.* Whether the grid is scaling a specific program within your service offerings or taking advantage of scavenged capacity, either effort can make computations more cost-effective. This is yet another reason to look at grids as scalability solutions.

The actual benefits that you realize from grid computing will depend on your specific application and implementation.

Cons of Grids

As with the benefits of grid computing, the significance or importance that you assign to each of the drawbacks of grid computing will be directly related to the applications that you are considering for the grid. Three major drawbacks exist with respect to grid computing: the difficulty in sharing the infrastructure simultaneously, the inability to work well with monolithic applications, and the increased complexity of utilizing these infrastructures.

Not Shared Simultaneously

The first drawback to using grids is that it may be difficult, if not impossible, to share the grid computing infrastructure simultaneously. Certainly, some grids are large enough that they have sufficient capacity to run many applications simultaneously, but they are really still running in separate grid environments, with the hosts just reallocated for a particular time period. For example, if you have a grid that consists of 100 hosts, you could run 10 applications on 10 separate hosts each. Although you should consider this scheme to be sharing the infrastructure, it is not sharing the infrastructure simultaneously. Running more than one application on the same host defeats the purpose of massive parallel computing and eliminates the benefits gained by using the grid infrastructure.

Grids are not great infrastructures to share with multiple tenants. With grid computing, the goal in running on a grid is to parallelize and increase the computational bandwidth for your application. Sharing or multitenancy can occur serially, with jobs running one after the other, in a grid environment where each application runs in isolation; that is, when one job is completed, the next job runs. This type of scheduling is common within systems that run large parallel processing infrastructures and are designed to be utilized simultaneously to compute large problem sets.

Thus, when you're running an application on the grid, both your application and the system must be flexible—that is, they must be able to either start and stop processing as necessary or run at a fixed time during each time period, usually daily or weekly. Because applications need the infrastructure to themselves, they are often scheduled to run during certain windows. If the application begins to exceed this window, perhaps because it has more data to process, the window must be rescheduled to accommodate this growth or else all other jobs in the queue will be delayed.

Monolithic Applications

One con of grid computing infrastructure is that it does not work well with monolithic applications. If you cannot divide the application into parts that can be run in parallel, the grid will not enhance its processing at all. As we stated in the discussion of Amdahl's law, nonsequential parts of a program will benefit from the parallelization,

but the rest of the program will not. Those parts of a program that must run in order, sequentially, are by definition not able to be parallelized.

Complexity

The last major drawback of grid computing is the increased complexity of the grid. Hosting and running an application by itself is often complex enough, considering the interactions that are required with users, other systems, databases, disk storage, and so on. Add to this already complex and highly volatile environment the need to run on top of a grid environment, and matters become even more complex. The grid is not just another set of hosts. Running on a grid requires a specialized operating system that, among many other things, manages which host has which job, what happens when a host dies in the middle of a job, and which data the host needs to perform the task; the same operating system is also charged with gathering and returning the processed results after the job is complete, deleting the data from the host, and aggregating the results. This multitude of tasks adds a lot of complexity. If you have ever debugged an application that has hundreds of instances of the same application on different servers, you can certainly imagine the immense challenge of debugging one application running across hundreds of servers.

Cons of Grid Computing

Here, we have identified three major drawbacks of grid computing. These are listed in no particular order and are not all-inclusive. Many more cons exist, but these three are representative of what you should expect if you include grid computing in your infrastructure.

- *Not shared simultaneously.* The grid computing infrastructure is not designed to be shared simultaneously without losing some of the benefit of running on a grid in the first place. As a consequence, jobs and applications are usually scheduled ahead of time and not run on demand.
- *Monolithic app.* If your application cannot be divided into smaller tasks, there is little to no benefit of running on a grid. To take advantage of the grid computing infrastructure, you must be able to break the application into nonsequential tasks that can run independently.
- *Complexity.* Running on a grid environment adds another layer of complexity to your application stack, which is probably already complex. If a problem occurs, debugging—including determining whether the problem exists because of a bug in your application code or the environment in which it is running—becomes much more difficult.

The actual drawbacks that you realize from grid computing will depend on your specific application and implementation.

If you are still up in the air about utilizing grid computing infrastructure, the next section will give you some ideas about where you might consider using a grid. As you read through this section, keep in mind the benefits and drawbacks covered earlier, because they should influence your decision of whether to proceed with a similar project in your own organization.

Different Uses for Grid Computing

In this section, we will cover some ideas and examples of using grid computing that we have either seen or discussed with clients and employers. By sharing these projects, we aim to give you a sampling of the possible implementations. Grid computing is a useful tool to utilize when scaling products and tools that support product development (e.g., parallel compilation).

This section describes four potential uses for grid computing—namely, running your production environment on a grid, using a grid for compilation, implementing parts of a data warehouse environment on a grid, and back-office processing on a grid. Many more implementations are possible, of course, but these should spark some ideas that you can use to jumpstart your own brainstorming session.

Production Grid

The first example calls for using grid computing in your production environment. This may not be possible for products that require real-time user interactions, such as those offered by Software as a Service companies. However, for IT organizations that rely on very mathematically complex applications to control manufacturing processes or shipping, this might be a great fit. Lots of these applications have historically resided on mainframe or midrange systems. Today, many technology organizations are finding it increasingly more difficult to support these larger and older machines from both a vendor and an engineering perspective. There are few engineers who know how to run and program these machines, and fewer still who would prefer to learn these skill sets instead of Web programming skills.

The grid computing environment offers solutions to the problems of machine and engineering support for older technologies. Specifically, migrating to a grid that runs lots of commodity hardware as opposed to one strategic piece of hardware can reduce your dependency on a single vendor for support and maintenance. Not only does this push the balance of power into your court, but it may yield significant cost savings for your organization. At the same time, you should be able to more easily find already-trained engineers or administrators who have experience running grids, or at the very least find employees who are excited about learning one of the newer technologies.

Build Grid

Alternatively, you might consider using a grid computing infrastructure for your build or compilation machines. Many applications, if they ran on a single host or developer machine, would take days to compile the entire code base. In this situation, a build farm or grid environment can come in very handy. Compiling is ideally suited for grids because so many divisions of work can be envisioned, and all of the parts can be performed nonsequentially. The later stages of the build, which require linking, start to become more sequential and, therefore, can be run on a grid. Nevertheless, the early stages of compilation are ideal candidates for a division of labor.

Most companies compile or build an executable version of the checked-in code each evening so that anyone who needs to test that version can have it available and to ensure that the code will actually build successfully. A great source of untapped compilation capacity at night is the testing environments. These are generally used during the day, but can be tapped in the evening to help augment the build machines. This is a simple implementation of CPU scavenging that can save quite a bit of money in additional hardware cost.

For C, C++, Objective C, and Objective C++ projects, builds implementing a distributed compilation process can be as simple as running distcc, which its sponsoring site (<http://www.distcc.org>) claims is a fast and free distributed compiler. This program works by simply running the distcc daemon on all the servers in the compilation grid, placing the names of these servers in an environmental variable, and then starting the build process.

Build Steps

Source code goes through many different types of compilers and many different processes to become code that can be executed by a machine. At a high level, computer languages can be classified as either compiled languages or interpreted languages. Forget about just-in-time (JIT) compilers and bytecode interpreters; compiled languages are ones for which the code written by the engineers is reduced to machine-readable code ahead of time using a compiler. Interpreted languages use an interpreter to read the code from the source file and execute it at run time.

Here are the rudimentary steps that are followed by most compilation processes and the corresponding input/output:

In Source code

1. *Preprocessing*. This is typically used to check for syntactical correctness.

Out/In Source code

2. *Compiling*. This step converts the source code to assembly code based on the language's definitions of syntax.

Out/In Assembly code

3. *Assembling.* This step converts the assembly language into machine instructions or object code.

Out/In Object code

4. *Linking.* This final step combines the object code into a single executable.

Out Executable code

A formal discussion of compiling is beyond the scope of this book, but this four-step process offers the high-level overview of how source code is turned into code that can be executed by a machine.

Data Warehouse Grid

The next example that we will cover entails using a grid as part of the data warehouse infrastructure. A data warehouse includes many components, ranging from the primary source databases to the end reports that users view. One particular component that can make use of a grid environment is the transformation phase of the extract–transform–load step (ETL) in the data warehouse. The ETL process determines how data is pulled or extracted from the primary sources, transformed into a different form (usually a denormalized star schema form), and then loaded into the data warehouse. This transformation can be computationally intensive and, therefore, is a prime candidate for the power of grid computing.

The transformation process may be as simple as denormalizing data, or it may be as complex as rolling up many months' worth of sales data for thousands of transactions. Processing that is very intense, such as monthly or even annual rollups, can often be broken into multiple pieces and divided among a host of computers. The resulting parts are very amenable to processing in a grid environment. As we noted in Chapter 27, Too Much Data, massive amounts of data are often the cause of not being able to process jobs such as the ETL in the time period required by either customers or internal users. Certainly, you should consider how to limit the amount of data that you are keeping and processing, but it is also possible that the data growth is occurring because of an exponential growth in traffic, which is what you want. One solution in such a case is to implement a grid computing infrastructure for the ETL to finish these jobs in a timely manner.

Back-Office Grid

The last example that we will cover involves back-office processing. An example of such back-office processing takes place every month in most companies when they

close the financial books. This is often a time of massive amounts of processing, data aggregation, and computations. This work is usually carried out on an enterprise resource planning (ERP) system, financial software package, homegrown system, or some combination of these. Attempting to use off-the-shelf software processing on a grid computing infrastructure when the system was not designed to do so may prove quite challenging, but it can be done. Often, very large ERP systems allow for significant customization and configuration. If you have ever been responsible for this process or waited days for this process to be finished, you will agree that being able to run it on possibly hundreds of host computers and finishing within hours would be a monumental improvement.

Many back-office systems are very computationally intensive—end-of-month processing is just one. Others include invoicing, supply reordering, resource planning, and quality assurance testing. Use these ideas as a springboard to develop your own list of potential places for improvement.

MapReduce

We covered MapReduce in Chapter 27, Too Much Data, but we should point out here that MapReduce is an implementation of distributed computing, which is another name for grid computing. In essence, MapReduce is a special-case grid computing framework used for text tokenizing and indexing.

Conclusion

In this chapter, we covered the pros and cons of grid computing and provided some real-world examples of where grid computing might fit. We discussed four pros: high computational rates, shared infrastructure, unused capacity, and cost. We also covered three cons: the environment is not shared well simultaneously, monolithic applications need not apply, and increased complexity may result.

We provided four real-world candidates for grid computing: the production environment of some applications, the transformation part of the data warehousing ETL process, the building or compiling process for applications, and the back-office processing of computationally intensive tasks. In each of these cases, fast and large amounts of computations may be needed. Not all similar applications can make use of the grid, but parts of many of them can be implemented on a grid. Perhaps it doesn't make sense to run the entire ETL process on a grid, but the transformation process might be the key part that needs the additional computations.

Grid computing does offer some very positive benefits when it is implemented correctly and the drawbacks are minimized. Use this very important technology and concept wisely in the fight to scale your organization, processes, and technology. Grids offer the ability to scale computationally intensive programs and should be considered for production as well as supporting processes. As grid computing and other technologies become widely available and more mainstream, technologists need to stay current on them, at least in sufficient detail to make good decisions about whether they make sense for the organization and its applications.

Key Points

- Grid computing offers high computation rates.
- Grid computing provides a shared infrastructure for applications that use these resources sequentially.
- Grid computing offers a good use of unused capacity in the form of CPU scavenging.
- Grid computing is not a good choice when there is a need to share resources simultaneously with other applications.
- Grid computing is not a good choice for monolithic applications.
- Grid computing does add some amount of complexity to the processing environment.
- Desktop computers and other unused servers have potential as untapped computational resources.

Chapter 29

Soaring in the Clouds

This is called, using the conquered foe to augment one's own strength.

—Sun Tzu

Cloud computing is probably the most important advancement in technology since the Internet. Although most people think of it as a recent innovation, the reality is that the cloud has taken more than a decade to become a reality. In this chapter, we cover the history that led up to the launch of cloud computing, discuss the common characteristics of clouds, and finish the chapter by examining the pros and cons of cloud computing.

Cloud computing is important to scalability because it offers the promise of cheap, on-demand storage and compute capacity. This has many advantages, and a few disadvantages, for physical hardware scaling. To be well versed in scaling applications, you must understand these concepts and appreciate how they could be implemented to scale an application or service.

Although the development of the technology and concepts for cloud computing have been in process for many years, the discussion and utilization of these advancements in mainstream technology organizations remain relatively new. Because of this, even definitions of the subject are not always completely agreed upon. We have been fortunate to be around the cloud environment through our clients for quite some time and have seen many players become involved in this field. As with the technology-agnostic designs in Chapter 20, Designing for Any Technology, we believe that it is the architecture—and not the technology—that is responsible for a product's capability to scale. As such, we consider clouds as an architectural component and not as a technology. With this perspective, the particular vendor or type of service is the equivalent of a type of technology chosen to implement the architecture. We will cover some of the technology components so that you have examples and can become familiar with them, but we will focus primarily on their use as architectural components. We also refer to IaaS solutions in Chapter 32, Planning Data Centers, where we discuss how they play into the decision of whether to own data centers, lease colocation space, or rent time in the cloud.

History and Definitions

The term *cloud* has been around for decades. No one is exactly sure when it was first used in relation to technology, but it dates at least as far back as the era when network diagrams came into vogue. A network diagram is a graphic representation of the physical or logic layout of a network, such as a telecommunications, routing, or neural network. The cloud on network diagrams was used to represent unspecified networks.

In the early 1990s, *cloud* evolved into a term for Asynchronous Transfer Mode (ATM) networks. ATM is a packet switching protocol that breaks data into cells and provides OSI layer 2, the data link. It was the core protocol used on the public switched phone network. When the World Wide Web began in 1991 as a CERN project built on top of the Internet, the cloud began to be used as a term and a symbol for the underlying infrastructure.

OSI Model

The Open Systems Interconnection Reference Model, or just OSI Model, is a descriptive abstraction of the layered model of network architecture. It identifies the different components of the network and explains how they are interrelated. The OSI Model includes seven layers. Starting from the lowest layer, they are as follows:

1. *Physical*. This layer contains the physical devices such as cards, cables, hubs, and repeaters.
 2. *Data Link*. This layer is responsible for the functional transmission of data between devices and includes protocols such as Ethernet.
 3. *Network*. This layer provides the switching and routing functionality and includes protocols such as Internet Protocol (IP).
 4. *Transport*. This layer provides reliability by keeping track of transmissions, and resending them if necessary. It includes Transmission Control Protocol (TCP).
 5. *Session*. This layer controls the communication by establishing, managing, and terminating connections between computers, such as with sockets.
 6. *Presentation*. This layer provides data presentation and encryption, such as with Secure Sockets Layer (SSL).
 7. *Application*. This layer lies between the software application and the network and includes implementation such as Hyper-Text Transfer Protocol (HTTP).
-

Cloud computing can also trace its lineage through the application service providers (ASPs) of the 1990s, which were the embodiment of the concept of outsourcing computer services. This concept later became known as Software as a Service (SaaS). The ASP model is an indirect descendant of the service bureaus of the 1960s and 1970s, which were an attempt at fulfilling the vision established by John McCarthy in his 1961 speech at Massachusetts Institute of Technology.¹ John McCarthy is the inventor of the programming language Lisp and the recipient of the 1971 Turing Award; he is also credited with coining the term *artificial intelligence*.²

The idea of the modern cloud concept was extended in October 2001 by IBM in its Autonomic Computing Manifesto.³ The essence of this paper was that the information technology infrastructure was becoming too complex and that it could collapse under its own weight if the management was not automated. Around this time, the concept of Software as a Service started to grow.

Another confluent event occurred around this time at the beginning of the 21st century—the dot-com bubble. As many tech startups were burning through capital and shutting down, those that would ultimately survive and thrive were tightening their belts on capital and operational expenditures. Amazon.com was one such company; it began modernizing its data centers using early concepts of virtualization over massive amounts of commodity hardware. Although it needed lots of capacity to deal with peak usage, Amazon decided to sell its unused capacity (which it had in spades during non-peak times) as a service.⁴

Out of the offering of spare capacity as a service came the concept and label of Infrastructure as a Service (IaaS). This term, which first appeared around 2006, typically refers to offerings of computer infrastructure such as servers, storage, networks, and bandwidth as a service instead of by subscription or contract. This pay-as-you-use model covered items that previously required either capital expenditure to purchase outright, long-term leases, or month-to-month subscriptions for partial tenancy of physical hardware.

From Infrastructure as a Service, we have seen an explosion of *Blah* as a Service offerings (where *Blah* means “fill in the blank with almost any word you can imagine”). We even have Everything as a Service (EaaS) now. All of these terms actually do share some common characteristics such as a purchasing model of “pay as you go” or “pay as you use it,” on-demand scalability of the amount that you use, and the concept that many people or multiple tenants can use the service simultaneously.

1. According to Wikipedia: http://en.wikipedia.org/wiki/Application_service_provider.

2. John McCarthy's home page at Stanford University, <http://www-formal.stanford.edu/jmc/>.

3. The original manifesto can be found at <http://www.research.ibm.com/autonomic/manifesto/>.

4. *BusinessWeek*. November 13, 2006. http://www.businessweek.com/magazine/content/06_46/b4009001.htm.

SaaS, PaaS, IaaS, and EaaS

All of the “*Blah as a Service*” concepts have certain characteristics in common. Among these are paying for what you use instead of buying it up front, scaling the amount you need without prior notice, and engaging in multiple tenancy (i.e., having many different people use the same service).

- *Software as a Service (SaaS)*. The original *Blah as a Service* term, SaaS started with customer relationship management (CRM) software as some of the earliest offerings. Almost any form of software can be offered in this manner, and it can be provided either over the Web or via download.
- *Platform as a Service (PaaS)*. This model provides all the required components for developing and deploying Web applications and services. These components include workflow management, integrated development environments, testing, deployment, and hosting.
- *Infrastructure as a Service (IaaS)*. This is the concept of offering computing infrastructure such as servers, storage, network, and bandwidth for use as necessary by clients. Amazon’s EC2 was one of the earliest IaaS offerings.
- *Everything as a Service (EaaS, XaaS, or *aaS)*. This is the idea of being able to retrieve on demand small components or modules of software that can be pieced together to provide a new Web-based application or service. Components could include retail, payments, search, security, and communications.

As these concepts evolve, their definitions will continue to be refined, and subcategories are sure to develop.

Public Versus Private Clouds

Some of the biggest names in technology are providing or have plans to provide cloud computing services. These companies include Amazon.com, Google, Hewlett-Packard, and Microsoft. Their services are publicly available clouds, of which anyone from individuals to other corporations can take advantage. However, if you are interested in running your application in a cloud environment but have concerns about a public cloud, there is the possibility of running a private cloud. By *private cloud*, we mean implementing a cloud on your own hardware in your own secure environment. With more open source cloud solutions becoming available, such as Eucalyptus and OpenStack, this is becoming a realistic solution.

There are obviously both pros and cons when running your application in a cloud environment. Some of these benefits and drawbacks are present regardless of

whether you use a private or public cloud. Certain drawbacks, however, are directly related to the fact that a public cloud is used. For instance, there may be a perception that the data is not as strongly protected as it would be inside of your network, similar to grid computing, even if the public cloud is very secure. One of the pros of a cloud is that you can allocate just the right amount of memory, CPU, and disk space to a particular application, thereby taking better advantage and improving utilization of the hardware. Thus, if you want to improve your hardware utilization and not deal with the perceived security concerns of a public cloud, you may want to consider running your own private cloud.

Characteristics and Architecture of Clouds

At this point in the evolution of the cloud computing concept, all cloud implementations share some basic characteristics. These characteristics have been mentioned briefly to this point, but it is now time to understand them in more detail. Almost all public cloud implementations have four specific characteristic, some of which do not apply to private clouds—namely, pay by usage, scale on demand, multiple tenants, and virtualization. Obviously, scaling on demand is an important characteristic when viewing the use of clouds from a scalability perspective, but don't dismiss the other characteristics as unimportant. For cash-strapped startup companies, paying as you go instead of purchasing hardware up front or signing multiyear contracts could mean the difference between surviving long enough to be successful or failing ignominiously.

Pay by Usage

The idea of pay as you go or pay according to your usage is commonplace in the Software as a Service world and has been adopted by the cloud computing services. Before cloud computing was available, to grow your application and have enough capacity to scale, you had limited options. If your organization was large enough, it probably owned or leased servers that were hosted in a data center or colocation facility. This model requires lots of upfront capital expenditure as well as a healthy monthly expense to continue paying bandwidth, power, space, and cooling costs. An alternative was to contract with a hosting service that provided the hardware, with the client then paying either a long-term lease or high monthly cost for the use of the hardware. Both models are reasonable and have benefits as well as drawbacks. Indeed, many companies still use one or both of these models and will likely do so for many years to come.

The cloud offers another alternative. Instead of long-term leases or high upfront capital outlays, this model allows you to avoid the upfront costs of purchasing hardware and instead pay based on your utilization of CPU, bandwidth, or storage, or possibly all three.

Scale on Demand

Another characteristic of cloud computing is the ability to scale on demand. As a subscriber or client of a cloud, you have the theoretical ability to scale as much as you need. Thus, if you need terabytes of storage or gigahertz or more, these computing capacities will be available to you. There are, of course, practical limits to this scalability, including how much actual capacity the cloud provider has to offer, but with the larger public clouds, it is reasonable to think that you could scale to the equivalent of several hundreds or thousands of servers with no issues. Some of our clients, however, have been large enough that a cloud provider did not have enough capacity for them to fail from one data center to another. In a private cloud, this constraint becomes your organization's limitation on physical hardware. The time that it takes to complete this process is near real time compared to the standard method of provisioning hardware in a data center.

Let's look at the typical process first, as if you were hosting your site at a colocation facility, and then consider the case in which you run your operation in a cloud environment. Adding hundreds of servers in a colocation facility or data center can take days, weeks, or months, depending on the organization's processes. For those readers who have never worked in an organization that hosted its processes with a colocation facility, this is a typical scenario that you might encounter.

Most organizations have budgeting and request processes that must be navigated no matter where or how the site is hosted. After the budget or the purchase order is approved, however, the processes of provisioning a new server in a cloud and provisioning one in a colocation facility are almost completely different. For a colocation facility, you need to ensure that you have the space and power available to accommodate the new servers. This can entail going to a new cage in a colocation provider if more space or power is not available in your current cage. If a new cage is required, contracts must be negotiated and signed for the lease of the new space and cross-connects are generally required to connect the cage's networks. After the necessary space and power are secured, purchase orders for the servers can be placed. Of course, some companies will stockpile servers in anticipation of capacity demand. Others will wait until the operations team alerts them that capacity is at a point where expanding the server pools is required.

Ordering and receiving the hardware can take weeks. After the hardware arrives at the colocation facility, it needs to be placed in the racks and powered up. After this is accomplished, the operations team can get started ghosting, jumpstarting, or kick-starting the server, depending on the operating system. Only then can the latest version of the software be loaded and the server added into the production pool. The total time for this process is at least days for the most efficient operations teams who already have hardware and space available. In most organizations, it takes weeks or months.

Now, let's consider how this process might look if you were hosting your site in a cloud environment and decided that you needed 20 more servers for a particular

pool. The process would start off similarly, with the budget or purchase order request to add to the monthly expense of the cloud services. After this is approved, the operations or engineering team would use the control panel of the cloud provider to simply request the number of virtual servers, specifying the desired size and speed. Within a few minutes, the systems would be available to load the machine image of choice and the latest application code could be installed. The servers could likely be placed into production within a few hours. This ability to scale on demand is a common characteristic of cloud computing.

Multiple Tenants

Although the ability to scale on demand is enticing, all that capacity is not just waiting for you. Public clouds have many users running a variety of applications on the same physical infrastructure—a concept known as *multitenanting* or having multiple tenants existing on the same cloud.

If all works as designed, these users never interact or impact one another. Data is not shared, access is not shared, and accounts are not shared. Each client has its own virtual environment that is walled off from other virtual environments. What you do share with other tenants in a cloud is the physical servers, network, and storage. You might have a virtual dual-processor server with 32GB of RAM, but it is likely running on an eight-processor server with 128GB of RAM that is being shared with several other tenants. Your traffic between servers and from the servers to the storage area goes across common networking gear. There are no routers, switches, or firewalls dedicated to individual tenants. The same goes for the storage. Tenants share storage on virtual network-attached storage (NAS) or storage area network (SAN) devices, which make it appear as if they are the only ones using the storage resource. The reality, of course, is that multiple tenants are using that same physical storage device.

The downside of this multitenanting scheme is that you don't know whose data and processes reside on the servers, storage devices, or network segment that you are on. If a neighbor gets DDOS'd (subjected to a distributed denial-of-service attack), that event can impact your network traffic. Similarly, your operations might be affected if a neighbor experiences a significant increase in activity that overloads the storage solutions or floods common network paths. We've had clients whose businesses were affected by this exact scenario on a public cloud. With the "noisy neighbor" problem, another service residing on the shared infrastructure affects your performance through excessive consumption of shared resources. Because of this variability, especially in input/output (I/O) capacity, many cloud providers now offer dedicated hardware or provisioning of input/output operations per second (IOPS). This obviously is a move away from the multitenant model and necessarily increases the price charged by the cloud provider.

Virtualization

All cloud computing offerings implement some form of a hypervisor on the servers that provides virtualization. This concept of virtualization is really the core architectural principle behind clouds. A *hypervisor* is either a hardware platform or a software service that allows multiple operating systems to run on a single host server, essentially “dicing” the server into multiple virtual servers. It is also known as a virtual machine monitor (VMM). Many vendors offer hardware and software solutions, such as VMware, Parallels, and Oracle VM. As mentioned in the discussion of multitenancy, such virtualization allows multiple users to exist on common hardware without knowing about or (we hope) impacting each other. Other virtualization, separation, and limitation techniques are used to restrict access of cloud clients to only those amounts of bandwidth and storage that they have purchased. The overall purpose of these techniques is to control access and provide, to the greatest extent possible, an environment that appears to be completely the client’s own. The better this is done, the less likely it is that clients will notice one another’s presence.

Another virtualization technique that is gaining popularity is containers. With this virtualization method, the kernel of an operating system allows for multiple isolated user space instances. Like virtual machines, these instances look and act like a real server from the point of view of the application. On UNIX-based operating systems, this technology represents an advanced implementation of the standard chroot and provides resource management features to limit the impact between containers. Docker.io is an open source project that implements Linux containers (LXC) to automate the deployment of applications inside software containers. It uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces to allow independent “containers” to run within a single Linux instance.

Common Cloud Characteristics

Cloud computing services share four basic characteristics:

- *Pay as You Go.* Users, subscribers, or clients pay for only the amount of bandwidth, storage, and processing that they consume.
- *Scale on Demand.* Cloud clients have access to theoretically unlimited capacity for scaling their applications by adding more bandwidth, servers, or storage.
- *Multiple Tenants.* Clouds service multiple—often many thousands—of clients. Clients share hardware, networks, bandwidth, and storage. Physical devices are not dedicated to clients.
- *Virtualization.* Multitenancy is accomplished through virtualization, the process whereby hardware can have multiple operating systems running on it simultaneously.

Some of these characteristics may or may not be present in private clouds. The concept of virtualization is at the core of all clouds, regardless of whether they are public or private. The idea of using farms of physical servers in different virtual forms to achieve greater utilization, multitenancy, or any other benefit is the basic premise behind the architecture of a cloud. The ability to scale as necessary is likely to be a common characteristic regardless of whether the resource is a private or public cloud. A physical restriction is placed on the amount of scale that can occur in any cloud, based on how large the cloud is and how much extra capacity is built into it.

Having multiple tenants, however, is not required in a private cloud. Tenants in a private cloud can mean different departments or different applications, not necessarily and probably not different companies. Pay as you go is also not required but possible in private clouds. Depending on how cost centers or departments are charged in an organization, each department might have to pay based on its usage of the private cloud. If a centralized operations team is responsible for building and running its own profit and loss center, it may very well charge departments or divisions for the services provided. This payment scheme may be based the computational, bandwidth, and storage usage in a private cloud.

Differences Between Clouds and Grids

Now that we've covered some of the history and basic characteristics of clouds and have done the same with grids in Chapter 28, Grid Computing, it's time to compare the two concepts. The terms *cloud* and *grid* are often confused and misused. Here, we'll cover a few of the differences and some of the similarities between the two to ensure that we are all clear on when each should be considered for use in our systems.

Clouds and grids serve different purposes. Clouds offer virtual environments for hosting user applications on one or many virtual servers. This makes clouds particularly compelling for applications that have unpredictable usage demands. When you aren't sure if you need 5 or 50 servers over the next three months, a cloud can be an ideal solution. Clouds allow users to share the infrastructure. Many different users can be present on the same physical hardware consuming and sharing computational, network, and storage resources.

Grids, in contrast, are infrastructures for dividing programs into small parts to be executed in parallel across two or more hosts. These environments are ideal for computationally intensive workloads. Grids are not necessarily great infrastructures to share with multiple tenants. You are likely running on a grid to parallelize and significantly increase the computational bandwidth for your application; sharing the infrastructure with other users simultaneously defeats that purpose. Sharing or multitenancy can occur serially, one after the other, in a grid environment where each application runs in isolation; when one job is completed, the next job runs. This challenge of enabling

multitenancy on grids is one of the core jobs of a grid operations team. Grids are also ideal only for applications that can be divided into elements that can be simultaneously executed. The throughput of a monolithic application cannot be helped by running on a grid. The same monolithic application can likely be replicated onto many individual servers in a cloud, however, and the throughput can be scaled by the number of servers added. Stated as simply as we can, clouds allow you to expand and contract your architecture; grids decompose work into parallelizable units.

While clouds and grids serve different purposes, there are many crossovers and similarities between them. The first major overlap is that some clouds run on top of a grid infrastructure. A good example is AppLogic from 3Tera, which is a grid operating system that is offered as software but also used to power a cloud that is offered as a service. Other similarities between clouds and grids include on-demand pricing models and scalable usage. If you need 50 extra servers in a cloud, you can get them allocated quickly and you pay only for the time that you are using them. The same is true in a grid environment. If you need 50 more nodes for improving the processing time of the application, you can have this capacity allocated rather quickly and you pay for only the nodes that you use.

At this point, you should understand that clouds and grids are fundamentally different concepts and serve different purposes, but have similarities and share common characteristics, and are sometimes intertwined in implementations.

Pros and Cons of Cloud Computing

Almost everything in life has both benefits and drawbacks; rarely is something completely beneficial or completely problematic. In most cases, the pros and cons can be debated, which increases the difficulty of making decisions for your business. Making matters even more complex is the reality that the pros and cons do not affect all businesses equally. Each company must weigh each of the identified benefits and drawbacks for its own situation. We will focus on how to use the pros and cons to make decisions later in the chapter. First, however, we cover what we consider the basic and most important of the benefits and drawbacks to cloud computing. Later, we will help put relative weightings to these factors as we discuss various implementations for different hypothetical businesses.

Pros of Cloud Computing

There are three major benefits to running your infrastructure on a cloud: cost, speed, and flexibility. Each one of these will have a varying degree of importance in your particular situation. In turn, you should weight each one in terms of how applicable the benefit is to you.

Cost

The cost model of consumption-based economics, or paying just for what you need as you need it, is a compelling one. This model works especially well if your organization is a cash-strapped startup. If your business model is one that actually pays for itself as your company grows and your expense model follows the same pattern, you have effectively eliminated a great deal of risk for your company. There are certainly other models that feature limited initial cash outlays, such as managed hosted environments, but they require that you purchase or lease equipment on a per-server basis and rule out the prospect of returning the equipment when you are not using it. For a startup, being able to last long enough to become successful is the first step toward scaling. At any company, being able to manage costs so that they stay in line with the volume of business is critical to ensure the ability to scale.

Figure 29.1 depicts a normal cost progression. As demand increases, you must stay ahead of that demand and purchase or lease the next server or storage unit or whatever piece of hardware to ensure you are capable of meeting demand. Most organizations are not great at capacity planning. This lack of skill may cause the gap between cost and demand to be larger than necessary or, even worse, allow demand to exceed capacity. In such a situation, a scramble will inevitably ensue to purchase more equipment while your customers are experiencing poor performance. The key when purchasing or leasing equipment in this scenario is to get the cost and demand lines as close as possible without letting them cross. Of course, with a cloud cost model, in which services are paid for only when used, these lines can be much tighter, almost touching in most cases.

Staffing flexibility is another benefit of cloud computing in terms of cost. Some cloud proponents maintain that you can skip staffing an operations team if you leverage cloud computing properly. While the authors' experiences suggest that this is a bit of an exaggeration, it is true that the cloud simplifies the complexity of

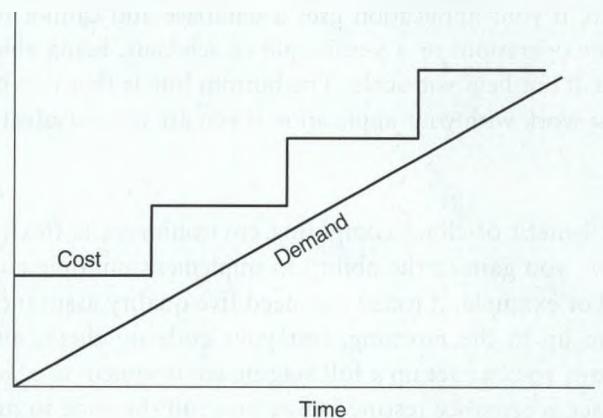


Figure 29.1 Stepwise Cost Function

operations, so that a reduction in staff is possible. Small companies can almost certainly get away with building an operating application in the cloud without the need for an operations team.

Speed

The next benefit that we see from the cloud environment is speed—specifically speed as it relates to time to market for our products. Here we are talking about time for procurement, provisioning, and deployment. Of all the colocation, data centers, managed hosting, and other infrastructure models, nothing is faster when it comes to adding another server than operating in a cloud environment (unless, of course, you've built similar tooling and run your own private cloud). Because of the virtual nature of cloud computing, deployment and provisioning proceed very quickly. If you are running a site that expects a spike of traffic over the weekend because of some sporting event, you can throw a few more virtual hosts into the pool on Friday afternoon and release them back Monday morning. With such a plan, you have these resources to use over the weekend to add capacity, but you don't pay for them the week after the spike in traffic ends. The ability to increase an application's usage of virtual hosts very quickly can serve as an effective method of scaling through peak traffic periods.

Today, many clouds augment capacity automatically through auto-scaling. If the application is designed to take advantage of this functionality, it can realize hands-off, near-real-time provisioning of compute and storage capacity. Even if you don't take advantage of auto-scaling, provisioning is still faster than it has ever been in the past with other models. Note that we don't mean to imply that it is wise to scale only on the *x*-axis with additional hardware instances. If your application has this capability and you've determined that this is a wise strategic architectural decision, then the greater speed adds a lot to your ability to deploy more hosts quickly. Nevertheless, you may not be able to utilize such speed if your application maintains state and your system lacks a mechanism for keeping users assigned to one host or centralizing the stateful session data. Also, if your application uses a database and cannot handle an *x*-axis split for read/write operations or a *y*-axis split of schemas, being able to quickly add more hardware will not help you scale. The bottom line is that this benefit of speedy deployments must work with your application if you are to take advantage of it.

Flexibility

The third major benefit of cloud computing environments is flexibility. What you give up in control, you gain in the ability to implement multiple configurations for different needs. For example, if today you need five quality assurance test instances, you can set them up in the morning, test your code on them, and remove them tonight. Tomorrow, you can set up a full staging environment to allow your customers to perform user acceptance testing before you roll the code to production. After your customers are satisfied, you can remove the environment and stop paying for it.

If you need a load testing environment that requires a bunch of individual hosts to provide multiple connections, ramping up a dozen virtual hosts for an hour of load testing is easily done in most cloud environments.

This flexibility to add, remove, or change your environments almost at whim is something that previous infrastructures haven't offered. After a team gets used to this ability to change and reconfigure, they won't want to be constrained by physical devices.

Benefits of Cloud Computing

There are three major categories of pros that we see with cloud computing, listed here in no particular order:

- *Cost.* The "pay as you use" model allows the amount that you spend to stay closer to the actual usage and is particularly helpful for cash-strapped companies.
- *Speed.* The speed in procurement, deployment, and provisioning cannot be matched by other infrastructure models.
- *Flexibility.* The ability to repurpose an existing environment from a quality assurance environment to a staging environment to an environment for load and performance, while not having to pay for three separate environments, is a tremendous benefit.

The importance of these considerations and how much you weight them when determining whether the cloud is the right environment for your organization should be based on your particular company's needs at a particular time.

Cons of Cloud Computing

There are five major categories of concerns or drawbacks for public cloud computing. These cons do not all apply to private clouds, but because the greatest utility and greatest public interest is in the use of public clouds, we will stick to using public clouds for our analysis. These five categories—security, portability, control, performance, and cost—are obviously very broad areas, so we will have to delve into each one in more detail to fully understand them.

Security

Not a month goes by that the public is not bombarded with media reports about leaked personal information or a security breach. This causes us to ask the question, "How do cloud providers store and safeguard our information?" The same question can be asked of many of our SaaS vendors. The slight difference is that with a SaaS implementation, the vendor often knows whether it is collecting and storing

sensitive information such as personally identifiable information (name, address, Social Security number, phone number, and so on), so it takes extra precautions and publishes its steps for safeguarding this information. In contrast, cloud providers have no idea what is being stored on their systems—that is, whether their customers are storing credit card numbers or blogs—and, therefore, do not take any extra precautions to restrict or block access to those data by their internal employees. Of course, there are ways around this, such as not storing any sensitive information on the cloud system, but those workarounds add more complexity to your system and potentially expose you to more risks. As stated earlier, this factor may or may not be a very important consideration for your particular company or application that you are considering hosting on a cloud.

A counter-argument is that in most cases the cloud provider handles security better than a small company could. Because most small startup companies lack the skill and dedicated staff to focus on infrastructure security issues, there's some benefit to hosting in a cloud, where ideally the vendor has already done some thinking about security.

Portability

We long for the day when we can port an application from one cloud to another without code or configuration changes—but this day has not yet arrived, nor do we think it will do so in the near future because it is not beneficial for cloud vendors to make this process easy. Of course, it's not impossible to migrate from one cloud to another or from a cloud to a physical server hosting environment, but this effort can be a nontrivial endeavor depending on the cloud and particular services being utilized. For instance, if you are using Amazon's Simple Storage Solution and you want to move to another cloud or to a set of physical servers, you will likely have to rework your application to implement storage in a simple database. Although not the most challenging engineering project, this rework does take time and resources that could be used to work on product features.

One of the principles discussed in Chapter 12, Establishing Architectural Principles, was “use commodity hardware”; this vendor-agnostic approach to hardware is important to scale in a cost-efficient manner. Not being able to port across clouds easily goes against this principle and, therefore, is a drawback that should be considered. Nevertheless, this situation is improving. Cloud providers such as Amazon Web Services are now making it easier to get in and out of their clouds with services such as import/export functionality that enables you to import virtual machine images from an existing environment to Amazon EC2 instances and export them back to on-premises environments.

Control

Whenever you rely solely on a single vendor for any part of your system, you are putting your company's future in the hands of another organization. We like to control

our own destiny as much as possible, so relinquishing a significant amount of control to a third party is a difficult step for us to take. This approach is probably acceptable when it comes to operating systems and relational database management systems, because ideally you will be using a vendor or product line that has been around for years and you aren't likely to build or manage anything better with your engineering team—unless, of course, you are in the business of operating systems or relational database management systems.

When it comes to hosting environments, many companies move away from managed environments because they get to a point where they have the technical talent on staff to handle the operational tasks required for hosting their own hardware and they get fed up with vendors making painful mistakes. Cloud environments are no different. They are staffed by people who are not your employees and who do not have a personal stake in your business. This is not to say that cloud or hosting providers have inferior employees. Quite the opposite: Their personnel are usually incredibly talented, but they do not know or understand your business. The provider has hundreds or thousands of servers to keep up and running. It doesn't know that this one is any more important than that one; they are all the same to the cloud or hosting provider. Giving up control of your infrastructure to this type of third party, therefore, adds an amount of risk into your business.

Many cloud vendors have not even reached the point of being able to offer guaranteed availability or uptime. When vendors do not stand behind their products with remuneration clauses specific for failures, it would be wise to consider their service as being on a "best effort" basis, which means you need to have an alternative method of receiving that service. As we mentioned in the discussion of portability, running on or switching between multiple clouds is not a simple task.

Performance

Another major category of concerns that we have in regard to clouds is performance related. From our experiences with our clients on cloud computing infrastructures, the expected performance from equivalent pieces of physical hardware and virtual hardware is not the same. This issue is obviously very important to the scalability of your application, especially if you have singletons—that is, single instances of batch jobs or parts of your application running on only a single server. Obviously, running a single instance of anything is not an effective way to scale, but it is common for a team to start on a single server and not test the job or program on multiple servers until they are needed. Migrating to a cloud and realizing that the processing of the job is falling behind on the new virtual server might put you in panic mode, causing you to hurriedly test and validate that that job can run correctly on multiple hosts.

Virtual hardware underperforms its physical counterparts in some aspects by orders of magnitude. The standard performance metrics in such cases include memory speed, CPU, disk access, and so on. There is no standard degradation or equivalence

among virtual hosts; in fact, it often varies within cloud environments and certainly varies from one vendor to another. Most companies and applications either don't notice this degradation or don't care about it, but when performing a cost-benefit analysis for switching to a cloud computing vendor, you need to test this yourself with your application. Do not take a vendor's word and assume that the cloud offers a truly equivalent virtual host. Each application has its own sensitivity and bottlenecks with regard to host performance. Some applications are bottlenecked on memory, such that slowing down memory by even 5% can cause the entire application to scale much more poorly on certain hosts. This performance matters when you are paying thousands of dollars in computing costs per month. What might have been a 12-month break even now becomes 18 or 24 months in some cases.

Cost

While we listed cost as a benefit of clouds, it can also be a drawback. Large, rapidly growing companies can usually realize greater margins by using wholly owned equipment than they can by operating in the cloud. This difference arises because IaaS operators, while being able to purchase and manage their equipment cost-effectively, are still looking to make a profit on their services. As such, many larger companies can negotiate purchase prices that allow them to operate on a lower cost basis after they calculate the overhead of teams and the amortization of equipment.

This is not to say that IaaS solutions don't have a place for large, fast-growing companies. They do! Systems that are not utilized around the clock and sit idle for large portions of the day are a waste of capital and can likely run more cost-effectively in the cloud. We will discuss this issue more in Chapter 32, Planning Data Centers, when we discuss how companies should think about their data center strategies from a broad perspective.

Drawbacks of Cloud Computing

There are five major categories of drawbacks that we see with cloud computing, listed here in no particular order:

- *Security.* Unlike SaaS companies, which know exactly which sensitive or personally identifiable information is being entered into their systems, cloud providers don't know and try not to care about their customers' data content. This lack of information leaves a potential gap in the security of your data.
- *Portability.* Although it is a simple matter to get up and running on a cloud, it can be difficult to move from the cloud to physical servers or other clouds depending on your application's implementation.

- *Control.* Outsourcing your infrastructure means giving a third party complete control over whether your application is available. Unlike ISPs, which can offer redundant resources, redundancy is not easy to accomplish with clouds at this point.
- *Performance.* Even though clouds are sold on computational equivalencies, the actual performance varies significantly between vendors and between physical and virtual hardware. You have to test this performance difference yourself to see if it matters to your application.
- *Cost.* Some large companies may find that they achieve higher margins by owning equipment and products that stay mostly busy.

The importance of any of these factors and how much you should be concerned with them are determined by your particular company's needs at a particular time.

So far, we have covered what we see as the top drawbacks and benefits of cloud computing as they exist today. As we have mentioned throughout this section, how these factors affect your decision to implement a cloud computing infrastructure will vary depending on your business and your application. In the next section, we'll highlight some of the different ways in which you might consider utilizing a cloud environment as well as how you might assess the importance of some of the factors discussed here based on your business and systems.

University of California–Berkeley on Clouds

Researchers at UC Berkeley have outlined their take on cloud computing in a paper "Above the Clouds: A Berkeley View of Cloud Computing."⁵ In this paper, they identify the top 10 obstacles that companies must overcome to utilize the cloud:

1. Availability of service
2. Data lock-in
3. Data confidentiality and audit ability
4. Data transfer bottlenecks
5. Performance unpredictability
6. Scalable storage
7. Bugs in large distributed systems

5. Michael Armbrust et al. "Above the Clouds: A Berkeley View of Cloud Computing." <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf>.

8. Scaling quickly
9. Reputation fate sharing
10. Software licensing

The article concludes by stating that these researchers believe cloud providers will continue to improve and overcome these obstacles. Furthermore, "developers would be wise to design their next generation of systems to be deployed into Cloud Computing."

Where Clouds Fit in Different Companies

In this section, we describe a few of the various implementations of clouds that we have either seen or recommended to our clients. Of course, you can host your application's production environment on a cloud, but many other environments are also available in today's software development organizations. Likewise, there are many ways to utilize different environments together, such as combining a managed hosting environment with a colocation facility. Obviously, hosting your production environment in a cloud offers "scale on demand" ability from a virtual hardware perspective. At the same time, you cannot be sure that your application's architecture can make use of this virtual hardware scaling; you must ensure that such compatibility exists ahead of time. This section also covers some other ways that clouds can help your organization scale. For instance, if your engineering or quality assurance teams are waiting for environments to become available for their use, the entire product development cycle is slowed down, which means scalability initiatives such as splitting databases, removing synchronous calls, and so on get delayed and affect your application's ability to scale.

Environments

For your production environment, you can host everything in one type of infrastructure, such as managed hosting, colocation, your own data center, a cloud computing environment, or some other scheme. At the same time, there are creative ways to utilize several of these options together to take advantage of their benefits but minimize their drawbacks. To see how this works, let's look at an example of an ad serving application.

The ad serving application consists of a pool of Web servers that accept the ad request, a pool of application servers that choose the right advertisement based on the information conveyed in the original request, an administrative tool that allows publishers and advertisers to administer their accounts, and a database for persistent storage of information. The ad servers in our application do not need to access the database for each ad request. Instead, they make a request to the database once

every 15 minutes to receive the latest advertisements. In this situation, we could obviously purchase a bunch of servers to rack in a colocation space for each of the Web server, ad server, administrative server, and database server pools. We could also just lease the use of these servers from a managed hosting provider and let the third-party vendor worry about the physical server. Alternatively, we could host all of this in a cloud environment on virtual hosts.

We think there is another alternative, as depicted in Figure 29.2. Perhaps we have the capital needed to purchase the pools of servers and we have the skill set in our team members required to handle setting up and running our own physical environment, so we decide to rent space at a colocation facility and purchase our own servers. But we also like the speed and flexibility gained from a cloud environment. Recognizing that the Web and app servers don't talk to the database very often, we decide to host one pool of each in a colocation facility and another pool of each on a cloud. The database will stay at the colocation but snapshots will be sent to the cloud to be used as a disaster recovery. The Web and application server pools in the cloud can be increased with greater traffic demands to help us cover unforeseen spikes.

Another use of cloud computing is in all the other environments that are required for modern software development organizations. These environments include, but are not limited to, production, staging, quality assurance, load and performance, development, build, and repositories. Many of these should be considered candidates for implementing in a cloud environment because of the possibly reduced cost,

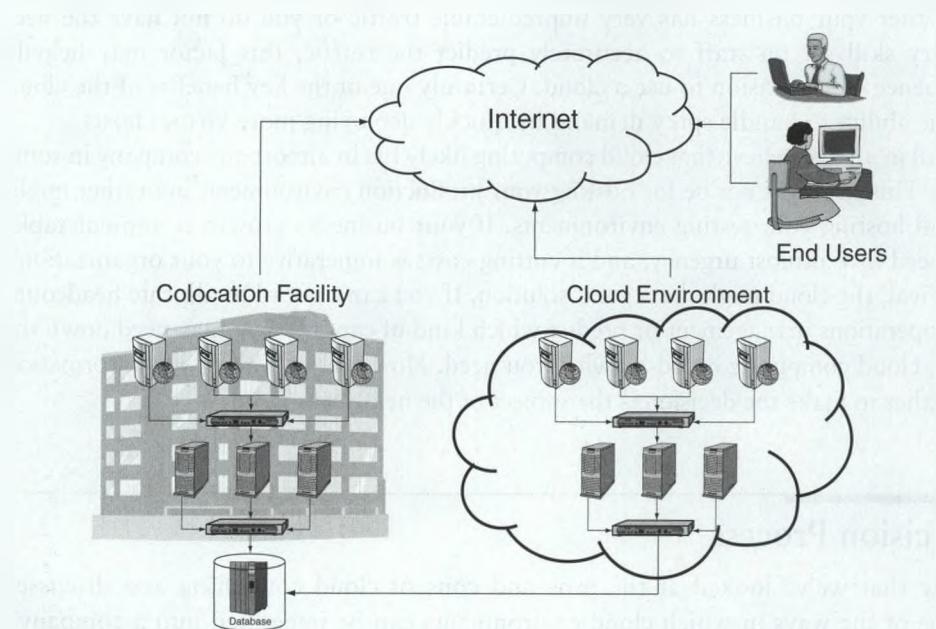


Figure 29.2 Combined Colocation and Cloud Production Environment

as well as the flexibility and speed in setting up these resources when needed and tearing them down when they are no longer needed. Even enterprise-class SaaS companies and *Fortune 500* corporations that might never consider hosting production instances of their applications on a cloud could benefit from utilizing the cloud for other types of environments.

Skill Sets

What are some of the other factors when considering whether to utilize a cloud, and if you do utilize the cloud, then for which environments is it appropriate? One consideration is the skill set and number of personnel that you have available to manage your operations infrastructure. If you do not have both networking and system administration skill sets among your operations staff, you need to consider this factor when determining whether you can implement and support a colocation environment. The most likely answer in that case is that you cannot. Without the necessary skill set, moving to a more sophisticated environment will actually cause more problems than it will solve. The cloud has similar issues; if someone isn't responsible for deploying and shutting down instances and this is left to each individual developer or engineer, it is very possible that the bill at the end of the month will be much higher than you expected. Instances that are left running are wasting money unless someone has made a purposeful decision that the instance is necessary.

Another type of skill set that may influence your decision is capacity planning. Whether your business has very unpredictable traffic or you do not have the necessary skill set on staff to accurately predict the traffic, this factor may heavily influence your decision to use a cloud. Certainly one of the key benefits of the cloud is the ability to handle spiky demand by quickly deploying more virtual hosts.

All in all, we believe that cloud computing likely fits in almost any company in some role. This fit might not be for hosting your production environment, but rather might entail hosting your testing environments. If your business's growth is unpredictable, if speed is of utmost urgency, and if cutting costs is imperative to your organization's survival, the cloud might be a great solution. If you can't afford to allocate headcount for operations management or predict which kind of capacity you may need down the line, cloud computing could be what you need. How you pull all of this information together to make the decision is the subject of the next section.

Decision Process

Now that we've looked at the pros and cons of cloud computing and discussed some of the ways in which cloud environments can be integrated into a company's

infrastructure, the last step is to provide a process for making the final decision on whether to pursue and implement cloud computing. The overall process that we recommend here is to first determine the goals or purpose of wanting to investigate cloud computing, then create alternative implementation options that achieve those goals. Weigh the pros and cons based on your particular situation. Rank each alternative based on the pros and cons. Based on the final tally of pros and cons, select an alternative.

Let's walk through an example to see how this decision-making process works. The first step is to determine which goals we hope to achieve by utilizing a cloud environment. Perhaps the goals are to lower the operations cost of infrastructure, decrease the time to procure and provision hardware, and maintain 99.99% availability for an application hosted on this environment. Based on these three goals, you might decide on three alternatives. The first is to do nothing, remain in a colocation facility, and forget about all this cloud computing talk. The second alternative is to use the cloud for only surge capacity but remain in the colocation facility for most of the application services. The third alternative is to move completely onto the cloud and out of the colocation space. This has accomplished steps 1 and 2 of the decision process.

Step 3 is to apply weights to all of the pros and cons that we can come up with for our alternative environments. Here, we will use the four cons and three pros that we outlined earlier. Note that "cost" can be either a pro or a con. For our example, we'll use it as a pro. We will use a 1, 3, or 9 scale to rank these and thereby differentiate the factors that we care about. The first con is security; we care about it to some extent, but we don't store personally identifiable information or credit card data, so we weight it a 3. We continue with portability and determine that we don't really feel the need to be able to move quickly between infrastructures so we weight it a 1. Next is control, which we really care about, so we rank it a 9. Finally, the last of the cons is performance. Because our application is not very memory or disk intensive, we don't feel that this is a major deal for us, so we weight it a 1. For the pros, we really care about cost, so we weight it a 9. The same with speed: It is one of the primary goals, so we care a lot about it and rate it as a 9. Last is flexibility, which we don't expect to use much, so we rank it a 1.

The fourth step is to rank each alternative on a scale from 0 to 5 based on how well it demonstrates each of the pros and cons. For example, with the "use the cloud for only surge capacity" alternative, the portability drawback should be ranked very low because it is not likely that we will need to exercise that option. Likewise, with the "move completely to the cloud" alternative, the controls are more heavily influential because there is no other environment, so it gets ranked a 5.

The completed decision matrix can be seen in Table 29.1. After all of the alternatives are scored against the pros and cons, the numbers can be multiplied and

Table 29.1 Decision Matrix

Cons	Weight 1, 3, or 9	No Cloud	Cloud for Surge	Completely Cloud
Security	-3	0	2	5
Portability	-1	0	1	4
Control	-9	0	3	5
Performance	-1	0	3	3
Pros				
Cost	9	0	3	5
Speed	9	0	3	3
Flexibility	1	0	1	1
TOTAL		0	18	6

summed. The weight of each pro is multiplied by the rank or score of each alternative; these products are summed for each alternative. For example, alternative 2, Cloud for Surge, has been ranked a 2 for security, which is weighted as -3. All cons are weighted with negative scores so the math is simpler. The product of the rank and the weight is -6, which is then summed with all the other products for alternative 2, for a total score: $(2 \therefore -3) + (1 \therefore -1) + (3 \therefore -9) + (3 \therefore -1) + (3 \therefore 9) + (3 \therefore 9) + (1 \therefore 1) = 18$.

The final step is to compare the total scores for each alternative and apply a level of common sense to it. Here, we have alternatives with 0, 9, and -6 scores; thus alternative 2 is clearly the better choice for us. Before automatically assuming that this is our final decision, we should verify that based on common sense and other factors that might not have been included, it is a sound decision. If something appears to be off or you want to add other factors such as operations skill sets, redo the matrix or have several people do the scoring independently to see how they would score the matrix differently.

Decision Steps

The following steps will help you decide whether to introduce cloud computing into your infrastructure:

1. Determine the goals or purpose of the change.
2. Create alternative designs for how to use cloud computing.

3. Place weights on all the pros and cons that you can come up with for cloud computing.
4. Rank or score the alternatives using the pros and cons.
5. Tally the scores for the various alternatives by multiplying each score by the weight and summing.

This decision matrix process will help you make data-driven decisions about which cloud computing alternative implementation is best for you.

The most likely question with regard to introducing cloud computing into your infrastructure is not *whether* to do it but rather *when* and *how* is the right way to do it. Cloud computing is not going away; in fact, it is likely to be the preferred but not only infrastructure model of the future. All of us need to keep an eye on how cloud computing evolves over the coming months and years. This technology has the potential to change the fundamental cost and organizational structures of most SaaS companies.

Conclusion

The history of cloud computing dates back several decades, and the concept of the modern-day cloud can be credited to IBM's manifesto. However, the evolution of cloud computing into its current form has been made possible by many different people and companies, including one of the first public cloud services, EC2.

Cloud computing has three major pros: cost, speed, and flexibility. The "pay per use" model is extremely attractive to companies and makes great sense. Working in a virtual environment also offers unequaled speed of procurement and provisioning. An example of flexibility is how you can utilize a set of virtual servers today as a quality assurance environment: shut them down at night and bring them back up the next day as a load and performance testing environment. This is a very attractive feature of the virtual host in cloud computing.

Cons of cloud computing include concerns about security, portability, control, performance, and cost. The security factor reflects concerns about how data is handled after it is in the cloud. The provider has no idea which type of data is stored there, and clients have no idea who has access to the data. This discrepancy between the two causes some concern. The portability factor addresses the fact that porting between clouds or clouds and physical hardware is not necessarily easy depending on the application. The control issues arise from the integration of another third-party vendor into your infrastructure that has influence over not just one part of your system's availability, but probably the entirety of your service's availability. Performance

is a con because it can vary dramatically between cloud vendors as well as compared to physical hardware. In terms of cost, the “pay per use” model makes getting started very attractive financially, but many large companies find that they can lower their compute costs by owning hardware equipment if it remains mostly in use. Your company and the applications that you are considering hosting on the cloud environment should dictate the degree to which you care about any of these cons.

Cloud computing can fit into different companies’ infrastructures in different ways. Some of these alternatives include existing not only as part of or all of the production environment but also in other environments such as quality assurance or development. As part of the production environment, cloud computing could be used for surge capacity or disaster recovery or, of course, to host all of production. The examples presented in this chapter were designed to show you how you might make use of the pros or benefits of cloud computing to aid your scaling efforts, whether directly for your production environment or more indirectly by aiding your product development cycle. This could take the form of capitalizing on the speed of provisioning virtual hardware or the flexibility in using the environments in different ways each day.

Each organization must make the decision of whether to use cloud computing in its operations. A five-step process for reaching this decision includes (1) establishing goals, (2) describing alternatives, (3) weighting pros and cons, (4) scoring the alternatives, and (5) tallying the scores and weightings to determine the highest-scoring alternative. The bottom line: Even if a cloud environment is not right for your organization today, you should continue looking at clouds because they will continue to improve, and it is very likely that cloud computing will be a good fit at some time.

Key Points

- The term *cloud* has been around for decades and was used primarily in network diagrams.
- The idea of the modern cloud concept was put forth by IBM in its Autonomic Computing Manifesto.
- Developing alongside the idea of cloud computing was the concept of Software as a Service, Infrastructure as a Service, and many more “as a Service” concepts.
- Software as a Service refers to almost any form of software that is offered via a “pay as you use” model.
- Infrastructure as a Service is the idea of offering infrastructure such as storage, servers, network, and bandwidth in a “pay as you use” model.

- Platform as a Service provides all the required components for developing and deploying Web applications and services.
- Everything as a Service is the idea of being able to have small components that can be pieced together to provide a new service.
- Pros of cloud computing include cost, speed, and flexibility.
- Cons of cloud computing include security, control, portability, performance, and cost.
- There are many ways to utilize cloud environments.
- Clouds can be used in conjunction with other infrastructure models by using them for surge capacity or disaster recovery.
- You can use cloud computing for development, quality assurance, load and performance testing, or just about any other environment, including production.
- A five-step process is recommended when deciding where and how to use cloud computing in your environment.
- All technologists should be aware of cloud computing; almost all organizations can take advantage of cloud computing in some way.

Fremstillet af Det Kgl. Biblioteks Nota-service til Huldayfa Hassan Ige Waberi. Eksemplaret er personligt og må ikke deles.

Chapter 30

Making Applications Cloud Ready

And if we are able thus to attack an inferior force with a superior one, our opponents will be in dire straits.

—Sun Tzu

In Chapter 29, Soaring in the Clouds, we covered the history, pros and cons, and typical uses of cloud computing. In this chapter, we continue our discussion of cloud computing but focus on ensuring your applications are ready for the cloud. Attempting to move a monolithic application into the cloud often results in extremely poor performance and a scramble to get back into a colocation or data center facility. This chapter outlines how to evaluate and prepare your application so that moving into the cloud will be a positive experience and not a disaster.

The Scale Cube in a Cloud

Unfortunately, because of misinformation and hype, many people believe that the cloud provides instant high availability and unlimited scalability for applications. Despite functionality such as auto-scaling, this is not the case. Technologists are still responsible for the scaling and availability of their applications. The cloud is just another technology or an architectural framework that can be used to achieve high availability and scalability; but one that cannot guarantee your application is available or scalable. Let's look at how a cloud provider might attempt to provide each axis of the AKF Scale Cube and see what we need to be responsible for ourselves.

x-Axis

Recall from Chapters 22, 23, and 24 that the *x*-axis of the AKF Scale Cube represents cloning of services or data with absolutely no bias. If we have a service or

platform consisting of N systems, each of the N systems can respond to any request and will give exactly the same answer as any other system. This is an x -axis scaling model. There is no bias to the service, customer, or data element. For a database, each x -axis implementation requires the complete cloning or duplication of an entire data set. We called this a horizontal duplication or cloning of data. The x -axis approach is fairly simple to implement in most cases.

In a cloud environment, we implement an x -axis split just as we would in a data center or colocation facility. That is, we replicate the code onto multiple instances or replicate the database onto other instances for read replicas. In a cloud environment, we can do this in several ways. We can launch multiple instances from a machine image and then deploy the identical code via a deployment script. Alternatively, we can build a brand-new machine image from an instance with the code already deployed. This will make the code part of the image. Deploying new images automatically deploys the code. A third way is to make use of a feature many clouds have, auto-scaling. On AWS you build an “Auto Scaling” group, set launch configurations, and establish scaling policies.

Each of these three methods of scaling via the x -axis in a cloud is straightforward and easy to implement. By comparison, y - and z -axis scaling is not so simple.

***y*- and *z*-Axes**

The y -axis of the cube of scale represents a separation of work responsibility within your application. It is most frequently thought of in terms of functions, methods, or services within an application. The y -axis split addresses the monolithic nature of an application by separating that application into parallel or pipelined processing flows.

The z -axis of the cube is a split based on a value that is “looked up” or determined at the time of the transaction; most often, this split is based on the requestor of the transaction or the customer of your service. This is the most common implementations of the z -axis, but not the only possible implementation. We often see ecommerce companies split along the z -axis on SKUs or product lines. Recall that a y -axis split helps us scale by reducing instructions and data necessary to perform a service. A z -axis split attempts to do the same thing through non-service-oriented segmentation, thereby reducing the amount of data and transactions needed for any one customer segment.

In a cloud environment, these splits do not happen automatically. You as the architect, designer, or implementer must handle this work. Even if you use a NoSQL solution such as MongoDB that facilitates data partitioning or sharding across nodes, you still are required to design the system in terms of how the data is split and where the nodes exist. Should we shard by customer or by SKU? Should the nodes all reside in a single data center or across multiple data centers? In the AWS vernacular, this would be across availability zones (US-East-1a, US-East-1b) or across regions (i.e.,

US-East-1 [northern Virginia], US-West-1 [Northern California]). One factor that you should consider is how highly available the data needs to be. If the answer is “very,” then you might want to place replica sets across regions. This will obviously come with costs such as data transfer rates, but will also provide increased latency for data replication. Replica sets across availability zones such as US-East-1a and US-East-1b have much lower latency, as they are data centers within a metro area.

Just because you are working a cloud environment, it doesn’t mean that you automatically get high availability and scalability for your applications. You as the technologist are still responsible for designing and deploying your system in a manner that ensures it scales and is highly available.

Overcoming Challenges

For all the benefits that a cloud computing environment provides (as discussed in Chapter 29, Soaring in the Clouds), some challenges must still be considered and in some cases overcome to ensure your application performs well in the cloud. In this section, we address two of these challenges: availability and variability in input/output.

Fault Isolation in a Cloud

Just because your environment is a cloud, that does not mean it will always be available. There is no magic in a cloud—just some pretty cool virtualization technology that allows us to share compute, network, and storage resources with many other customers, thereby reducing the cost for all customers. Underneath the surface are still servers, disks, uninterruptible power supplies (UPS), load balancers, switches, routers, and data centers. One of our mantras that we use in our consulting practice is “Everything fails!” Whether it is a server, network gear, a database, or even an entire data center, eventually every resource will fail. If you don’t believe us about data centers failing, just do a Web search for “data center failures.” You’ll find reports such as Ponemon Institute’s 2013 survey, which showed that 95% of respondents experienced an unplanned data center outage.¹ A cloud environment, whether it is a private cloud or a public cloud environment, is no different. It contains servers and network gear that fail, and occasionally entire data centers that fail, or in AWS’s case entire regions, which consist of multiple data centers in a metro area, that fail.

This isn’t to say that AWS doesn’t have great availability or uptime; rather, the intent here is to point out that you should be aware that its services occasionally fail as well. In fact, Amazon offers a guaranteed monthly service level agreement (SLA)

1. http://www.emersonnetworkpower.com/documentation/en-us/brands/liebert/documents/white%20papers/2013_emerson_data_center_cost_downtime_sl-24680.pdf.

of 99.95% availability of its compute nodes (EC2) or it will provide service credits.² However, if AWS has 99.95% availability, your service running on top cannot have anything higher. You cannot achieve (assuming AWS meets but doesn't exceed its SLA) 99.99% availability even if your application has no downtime or outages. If your application has 99.9% availability and the hosting provider has 99.95% availability, then your total availability is likely to be $0.999 \times 0.9995 = 0.9985$ or 99.85%. The reason is that your application's outages (43.2 minutes per month at 99.9%) will not always coincide with your hosting provider's outages (21.6 minutes per month at 99.95%).

AWS US-East-1 Region Outages

Amazon Web Services (AWS) is a terrific service that has great availability and scalability. However, it, too, experiences outages that you as a technology leader must be prepared for. Here are a few of the outages that AWS has had in its US-East-1 region in northern Virginia.

- June 2012 Outage³: This particular incident manifested itself in two forms. The first was unavailability of instances and volumes running in the affected data center, which was limited to the affected availability zone. Other availability zones in the US East-1 region continued functioning. The impact was degradation of service "control planes," which allow customers to take action and create, remove, or change resources. Control planes aren't required for the ongoing use of resources, but they are useful during outages, when customers try to react to the loss of resources in one availability zone by moving to another.⁴
- October 2012 Outage⁵: Customers reported difficulty using service APIs to manage their resources during this event for several hours. AWS throttled its APIs during the event, which disproportionately impacted some customers and affected their ability to use the APIs.
- September 2013 Outage⁶: Network connectivity issues disrupted popular applications such as Heroku, GitHub, and CMSWire along with other customers hosted in the US-East-1 region.

2. <http://aws.amazon.com/ec2/sla/>; accessed October 6, 2014.

3. <http://aws.amazon.com/message/67457/>.

4. See Netflix's report on this outage: <http://techblog.netflix.com/2012/12/a-closer-look-at-christmas-eve-outage.html>.

5. <https://aws.amazon.com/message/680342/>.

6. <http://www.infoq.com/news/2013/09/aws-east-outage>.

Hosting your application or service in a single data center or even a single availability zone in AWS that encompasses multiple data centers is not enough to protect your application from outages. If you need high availability, you should not delegate this responsibility to your vendor. It's your responsibility and you should embrace it, ensuring your application is designed, built, and deployed to be highly available.

Some of the difficulties of managing this feat within a cloud are the same that you would encounter in a traditional data center. Machine images, code packages, and data are not automatically replicated across regions or data centers. Your management and deployment tools must be developed so that they can move images, containers, code, and data. Of course, this endeavor doesn't come for free from a network bandwidth perspective, either. Replicating your database between East Coast and West Coast data centers or between regions takes up bandwidth or adds costs in the form of transfer fees.

While Infrastructure as a Service (IaaS) providers, such as the typical cloud computing environment, are generally not in the business of replicating your code and data across data centers, many Platform as a Service (PaaS) providers do attempt to offer this service, albeit with varying levels of success. Some of our clients have found that the PaaS provider did not have enough capacity to dynamically move their applications during an outage. Again, we would reiterate that your application's availability and scalability are your responsibility. You cannot pass this responsibility to a vendor and expect it to care as much as you do about your application's availability.

Variability in Input/Output

Another major issue with cloud computing environments for typical SaaS applications or services is the variability in input/output from a storage perspective. Input/output (I/O) is the communication between compute devices (i.e., servers) and other information processing systems (i.e., another server or a storage subsystem). Inputs and outputs are the signals or data transferred in and out, respectively.

Two primary measurements are used when discussing I/O: input/output per second (IOPS) and megabytes per second (MBPS). IOPS measures how many I/O requests the disk I/O path can satisfy in a second. This value is generally in inverse proportion to the size of the I/O requests; that is, the larger the I/O requests, the lower the IOPS. This relationship should be fairly obvious, as it takes more time to process a 256KB I/O request than it does an 8KB request. MBPS measures how much data can be pumped through the disk I/O path. If you consider the I/O path to be a pipeline, MBPS measures how big the pipeline is and, therefore, how many megabytes of data can be pushed through it. For a given I/O path, MBPS is in direct proportion to the size of the I/O requests; that is, the larger the I/O requests, the higher the MBPS. Larger requests give you better throughput because they incur less disk seek time than smaller requests.

In a typical system, there are many I/O operations. When the memory requirements on the compute node exceed the allocated memory, paging ensues. Paging is a memory-management technique that allows a computer to store and retrieve data from secondary storage. With this approach, the operating system uses same-size blocks called “pages.” The secondary storage often comprises the disk drives. In a cloud environment, the attached storage devices (disk drives or SSD) are not directly attached storage but rather network storage where data must traverse the network from the compute node to the storage device. If the network isn’t highly utilized, the transfer can happen very quickly, as we’ve come to expect with storage. However, when the network becomes highly utilized (possibly saturated), I/O may slow down by orders of magnitude. When you hosted your application in your data center with directly attached storage (DAS), you might have seen 175–210 IOPS and 208 MBPS on 15K SAS drives.⁷ If these rates drop to 15 IOPS and 10 MBPS, could your application still respond to a user’s request in a timely manner? The answer is probably “no,” unless you could dynamically spin up more servers to distribute the requests across more application servers.

“How could a cloud computing environment have a reduced I/O performance by an order of magnitude?” you ask. The answer relates to the “noisy neighbor” effect. In a shared infrastructure, the activity of a neighbor on the same network segment may affect your virtual machine’s performance if that neighbor uses more of the network than expected. This type of problem is referred to in economics theory as the “tragedy of the commons,” wherein individuals, acting independently and rationally according to their own self-interest, behave contrary to the whole group’s long-term best interests by depleting some common resource. Often the guilty party or noisy neighbor can’t help itself from consuming too much network bandwidth. Sometimes the neighbor is the victim of a distributed denial-of-service (DDOS) attack. Sometimes the neighbor has DDOS’d itself with a bug in its code. Because users can’t monitor the network in a cloud environment (this type of monitoring isn’t allowed to be done by customers using the service but it is obviously performed by the cloud provider), the guilty party often doesn’t know the significance of the impact that its seemingly localized problem is having on everyone else.

Since you cannot isolate or protect yourself from the noisy neighbor problem and you don’t know when it will occur, what can you do? You need to prepare your application to either (1) pay for provisioned or guaranteed IOPS; (2) handle significant I/O constraints by increasing compute capacity (auto-scaling via *x*-axis replication); or (3) redirect user requests to another zone, region, or data center that is not experiencing the degradation of I/O. The third approach also requires a level of

7. Symantec. “Getting the Hang of IOPS v1.3.” <http://www.symantec.com/connect/articles/getting-hang-iops-v13>.

monitoring and instrumentation to recognize when degradation is occurring and which zones or regions are processing I/O normally. Handling the variability in I/O by paying more is certainly one option—but just like buying larger compute nodes or servers to handle more user requests instead of continuing to buy commodity hardware, it's a dangerous path to take. While it might make sense in the short term to buy your way out of a bind, in the long run this approach will lead to problems. Just as you can only buy so much compute capacity in a single server and it gets more expensive per core as you add more such capacity, this approach with a cloud vendor simply gets more expensive as your needs grow.

A far better approach than paying to eliminate the problem is to design your application and deploy it in such a manner that when (not “if”) the noisy neighbor acts up, you can expand your compute capacity to handle the problem. The auto-scaling we discussed earlier is a form of horizontal scaling along the *x*-axis. This method can also be applied to the database should I/O problems affect your database tier. Instead of more compute nodes, we would spin up more read replicas of our database to scale this problem out. If auto-scaling isn't an option, perhaps because your application does many more writes than reads (this is almost never the case with SaaS software, as user requests are usually 80% to 90% reads and very few writes), then you might look to the third option of shifting the traffic to a different data center.

Although shifting user requests to a different data center can be done via a disaster recovery scenario with a complete failover, this “nuclear option” is usually difficult to recover from and technologists are generally reluctant to pull the trigger. If your application has a *y*- or *z*-axis split that is separated into different data centers or regions, shifting traffic is much less of a big deal. You already have user requests being processed in the other facility and you're just adding some more.

Of course, you can use any combination of these three options—buying, scaling, or shifting—to ensure that you are prepared to handle these types of cloud environment problems. Whatever you decide, leaving the fate and availability of your service or application in the hands of a vendor is not a wise move. Let's now look at a real-world example of how a company prepared its application for a cloud environment.

Intuit Case Study

In this chapter, we've focused on making an existing application or SaaS offering cloud ready. Sometimes, however, you have to start designing an application from the ground up to be capable of running in a cloud environment. The SaaS offering that we discuss in this section was built specifically to run in a cloud environment, with the developers taking into consideration the constraints and problems that come with that decision up front in the design and development phases.

Intuit's Live Community makes it easy to find answers to tax-related questions by facilitating connections with tax experts, TurboTax employees, and other TurboTax users. You can access Live Community from any screen in Intuit's TurboTax application to post a question and get advice from the community. One of Intuit's Engineering Fellows, Felipe Cabrera, described the Live Community cloud environment as one in which "everything fails" and "[these failures] happen more often than in our gold-plated data centers." With this understanding of the cloud environment, Intuit designed the application to run across multiple availability zones within Amazon Web Services (AWS). It also designed the solution with the capability to move between AWS regions for the purposes of disaster recovery. Elastic load balancers distribute users' requests across machine instances in multiple availability zones.⁸ This fault isolation between zones helps with the availability issues that sometimes affect machine instances or entire availability zones.

To accommodate the variability in I/O or noisy neighbor problems, the team implemented several different architectural strategies. The first was an *x*-axis split of the database, implemented by creating a read replica of the database that the application can use for read (select) queries. The next strategy was designing and developing the application so that it could make use of AWS Auto Scaling,⁹ a Web service designed to launch or terminate Amazon EC2 instances automatically based on user-defined policies, schedules, and health checks. This allows for the application tier to scale via the *x*-axis as demand increases or decreases or as the application begins performing better or worse because of the variability in I/O.

To monitor the performance of the critical customer interactions, the team built an extensive testing infrastructure. This infrastructure is often modified and extended. It allows the team to monitor whether changes to the service have the expected impact on the performance of customer interactions.

While these designs helped the Live Community scale in a cloud environment, they didn't come without hard-won lessons. One painful lesson dealt with the use of a read database slave for select queries. The library that the developers employed to enable this split in read and write queries did not fail gracefully. When the read database slave was not available, the entire application halted. The problem was quickly remedied, but the database split designed to improve scalability ended up costing some availability until the developers fixed the hard dependency issue. This incident also highlighted the lack of a mechanism to test the failure of an availability zone.

Another painful lesson was learned with auto-scaling. This AWS service allows EC2 (virtual machines) to be added or removed from the pool of available servers based on predefined criteria. Sometimes the criterion is CPU load; other times it

8. For details of this service, see <http://aws.amazon.com/about-aws/whats-new/2013/11/06/elastic-load-balancing-adds-cross-zone-load-balancing/>.

9. For details of this service, see <http://aws.amazon.com/documentation/autoscaling/>.

might be a health check. The primary scale constraint for Intuit's Live Community application was concurrent connections. Unfortunately, the team didn't instrument the auto-scaling service to watch this metric. When the application servers ran out of connections, they stopped serving traffic, which caused a customer incident where the Live Community wasn't available for a period of time. Once this was remedied (by adding a health check and by scaling based on concurrent connections), auto-scaling worked very well for Intuit's application, which has a huge variability in seasonal demand because of the tax season (January to April 15) each year.

As you can see from this example, even a great company like Intuit, which purposefully designs its application to be cloud ready, can run into problems along the way. What enabled the company to be so successful with the Live Community¹⁰ service in the cloud is that it had knowledge about the challenges of a cloud environment, spent time up front to design around these challenges, and reacted quickly when it encountered issues that it could not anticipate.

Conclusion

In this chapter, we covered the AKF Scale Cube in a cloud environment, discussed the two major problems for applications being moved from a data center hosting environment into a cloud environment, and looked at a real-world example of a company that prepared its application for the cloud environment.

We can accomplish x -axis splits in the cloud in several ways—launch multiple instances from a machine image and then deploy the identical code, build a brand-new machine image from an instance with the code already deployed, or make use of auto-scaling, a feature found in many cloud environments. However, in a cloud environment, y - and z -axis splits do not happen automatically. You as the architect, designer, or implementer must handle this work. Even if you use a technology that facilitates data partitioning or sharding across nodes, you are still required to design the system in terms of how the data is split and where the nodes exist.

The two main challenges for SaaS companies seeking to move their applications from a data center hosting environment to a cloud environment are availability and variability in I/O. Everything fails, so we need to be prepared when a cloud environment fails. There are numerous examples of cloud providers that have experienced outages of entire data centers or even multiple data centers. If you need high availability, you should not delegate this responsibility to your vendor. It's your responsibility and you should embrace it, ensuring your application is designed, built, and deployed to be highly available.

10. The Live Community team has evolved over time. Four people critical to the service have been Todd Goodyear, Jimmy Armitage, Vinu Somayaji, and Bradley Feeley.

Creating fault-isolated, highly available services in the cloud poses the same challenges that you would find in a traditional data center. Machine images, code packages, and data are not replicated automatically. Your management and deployment tools must be developed such that they can move images, containers, code, and data.

Another major issue with cloud computing environments for typical SaaS applications or services is the variability in input/output from a storage perspective. Because you can't predict when the variability will occur, you need to prepare your application to either (1) pay for provisioned or guaranteed IOPS, (2) handle significant I/O constraints by increasing compute capacity (auto-scaling via the *x*-axis replication), or (3) redirect user requests to another zone, region, data center, or other resource that is not experiencing the degradation of I/O.

Building for scale means testing for scale and availability. Testing failover across regions in a cloud is a must. Figuring out how to test slow response time is also an important but often overlooked step.

Key Points

- In a cloud environment, we implement an *x*-axis split just as we would in a data center or colocation facility. We can do this several ways:
 - Launch multiple instances from a machine image and then deploy the identical code
 - Build a brand-new machine image from an instance with the code already deployed
 - Use auto-scaling, a feature that many clouds have
- In a cloud environment, the *y*- and *z*-axis splits do not happen automatically. You as the architect, designer, or implementer must handle this work.
- Availability of compute nodes, storage, network, and even data centers can be a problem in cloud environments. While we have to deal with these issues in a data center environment, we have more control and monitoring of them in a data center or colocation.
- The variability of input/output as measured by IOPS and MBPS can be a significant issue with getting applications ready to transition into the cloud.
- You can prepare an application for running in the cloud with the variability of I/O by doing the following:
 - Paying for provisioned or guaranteed IOPS
 - Handling significant I/O constraints by increasing compute capacity (auto-scaling via the *x*-axis replication)
 - Redirecting user requests to another zone, region, data center, or other resource that is not experiencing the degradation of I/O

Chapter 31

Monitoring Applications

Gongs and drums, banners and flags, are means whereby the ears and eyes of the host may be focused on one particular point.

—Sun Tzu

When experiencing rapid growth, companies need to identify scale bottlenecks quickly or suffer prolonged and painful outages. Small increases in response time today may be harbingers of brownouts tomorrow. This chapter discusses the reason why companies struggle with monitoring their products and suggests ways to fix that struggle by employing a framework for maturing monitoring over time.

“Why Didn’t We Catch That Earlier?”

If you are like us, you have likely spent a good portion of your career answering this question. While it is certainly important, an even more valuable question and one that will return greater long-term shareholder value is “Which flaws in our process allowed us to launch the service without the appropriate monitoring to catch such an issue as this?” At first glance, the two questions appear similar, but they really have completely different areas of focus. The first question, “Why didn’t we catch that earlier?” deals with the most recent incident. The second question addresses the systemic problems that allow systems to be developed without the monitoring to validate they are performing to expectations. Recall from Chapter 8, Managing Incidents and Problems, that problems cause incidents and that a single problem may be related to multiple incidents. In this vernacular, the first question addresses the incident and the second question addresses the problem.

In our experience, the most common reason for not identifying incidents early and catching problems through monitoring is that most systems aren’t designed to be monitored. Monitoring, if done at all, is implemented as an afterthought. Often, the team responsible for determining whether the product is performing to expectations

has no hand in defining or designing the product in question. Imagine if all the dashboard lights and gauges on your car were aftermarket accessories installed by a dealer prior to sale. Sounds ridiculous, right? Why then, would we install monitoring on the products we produce only during product deployment?

“Design to be monitored” is an approach wherein one builds monitoring *into* the product rather than *around* it. A system that is designed to be monitored might evaluate the response times of all of the services with which it interacts and alert when response times are out of the normal range for that time of day. This same system might also evaluate the rate of error logging it performs over time and alert when that rate significantly changes or the composition of the errors changes. Most importantly, “design to be monitored” assumes that we understand the key performance indicators (KPIs) that define success for the product initiative and that we monitor these KPIs in real time. For commerce solutions, these KPIs almost always include revenue per minute, shopping cart abandons, add to cart calls, and so forth. For payment solutions, KPIs include payment volume, average payment size, and fraud statistics. In business-to-business (B2B) Software as a Service (SaaS) solutions, the KPIs may include rates of critical function calls from which the customers derive the greatest utility, such as customer additions and updates in a CRM solution. And of course, response times for key functionality are always important.

A Framework for Monitoring

How often have your postmortems identified that your monitoring systems gave early indications of problems upon which no action was taken? “How,” you might ask yourself, “did these alerts go unnoticed?”

A common answer is that the monitoring system simply gives too many false positives (or false negatives); put another way, there is “too much noise” in the alerting. “If we could just take some of the noise out of the system, my team could sleep better and address the real issues that we face,” your DevOps staff might say. “We need to implement a new monitoring system or reimplement the existing system if we want to get better.”

We’ve heard the reasons for implementing new and better monitoring systems time and again. Rarely is it the best answer to the problem of ineffective monitoring. The real issue isn’t simply that the monitoring system is not meeting the needs of the company; it is that the approach to monitoring is all wrong. Most teams have properly implemented monitoring to identify potential problems—but they have not implemented the appropriate monitoring to identify incidents. In effect, they start in the wrong place. It’s not enough to just build monitoring into our products. We need to evolve monitoring over time based on lessons that we learn along the way.

Just as Agile software development methods attempt to solve the problem associated with not knowing all of your requirements before you develop a piece of software, so must we have an agile and evolutionary development mindset for our monitoring platforms and systems. The evolutionary method that we propose answers three questions, with each question supporting the delineation of incidents and problems that we identified in Chapter 8.

The first question that we ask in our evolutionary model for monitoring is, “Is there an incident?” Specifically, we are interested in determining whether the product is behaving in a manner that is inconsistent with our desired business results or inconsistent with past behaviors. The former helps us understand where we are not meeting expectations, and the later identifies changes in customer behaviors that might indicate product difficulties and customer-impacting incidents. In our experience, many companies tend to completely bypass these very important questions and immediately dive into an unguided exploration of the next question we should ask, “Where is the problem located?” or, even worse, “What is the problem?”

In monitoring, bypassing the “Is there an incident?” question in favor of the next two questions is disastrous. No team can identify all possible scenarios and problems that might lead to customer-impacting events. As such, without incident-level monitoring, you are guaranteed to miss at least some incidents because you don’t have the appropriate problem monitoring in place. The result is an increase in the time to detect incidents, which in turn means an increase in incident duration and therefore incident impact. Not building systems that first answer the “Is there an incident?” question results in two additional issues. First, our teams waste person-years of effort in chasing problems that never lead to customer-impacting incidents. This time would be better spent focusing on problems that have caused incidents or in working on features. Second, we commit another product sin: Customers inform us about our own problems. Customers don’t want to be the ones telling you about incidents with your systems or products. Customers expect that, at best, they will be telling you something that you already know and that you will be deep in the process of fixing whatever issue they are experiencing. Systems that answer the question “Is there an incident?” are very often built around monitoring of key customer transactions and critical business key performance indicators. They may also be diagnostic services built into our platform, similar to the statistical process control example given later in this chapter. We will discuss these possibilities in greater detail in the “User Experience and Business Metrics” section later in this chapter.

The next question to answer in evolutionary fashion is, “Where is the problem?” At this point, we have built a system that tells us definitively that a problem (an incident) exists somewhere in our system; in the best-case scenario, it is correlated with a single or a handful of business metrics. Now we need to isolate where the problem exists. These types of systems very often are broad category collection agents

that give us indications of resource utilization over time. Ideally, they are graphical in nature, and perhaps we are even applying a neat little statistical process control chart trick (covered later in this chapter). Maybe we even have a nice user interface that gives us a *heat map* indicating areas or sections of our system that are not performing as we would expect. These types of systems are really meant to help us quickly identify where we should be applying our efforts in isolating exactly what the problem is or what the root cause of our incident might be.

The last question in our evolutionary model of monitoring is “What is the problem?” Note that we’ve moved from identifying that there is an incident, consistent with our definition in Chapter 8, to isolating the area causing that incident, to identification of the problem itself. As we move from identifying that something is impacting our customers to determining the cause for the incident, two things happen. First, the amount of data that we need to collect as we evolve from the first to the third question grows. We need only a few pieces of data to identify whether something, somewhere is wrong. But to be able to answer the “What is the problem?” question across the entire range of possible problems that we might have, we need to collect a whole lot of data over a substantial period of time. Second, we naturally narrow our focus from the very broad “We have an incident” to the very narrow “I’ve found what the problem is.” The two are inversely correlated in terms of size, as Figure 31.1 indicates. The more specific the answer to the question, the more data we need to collect to determine the answer.

To be able to answer precisely for all problems what the source is, we must have quite a bit of data. The actual problem itself can likely be identified with one very small slice of this data, but to get that answer we have to collect data for all potential

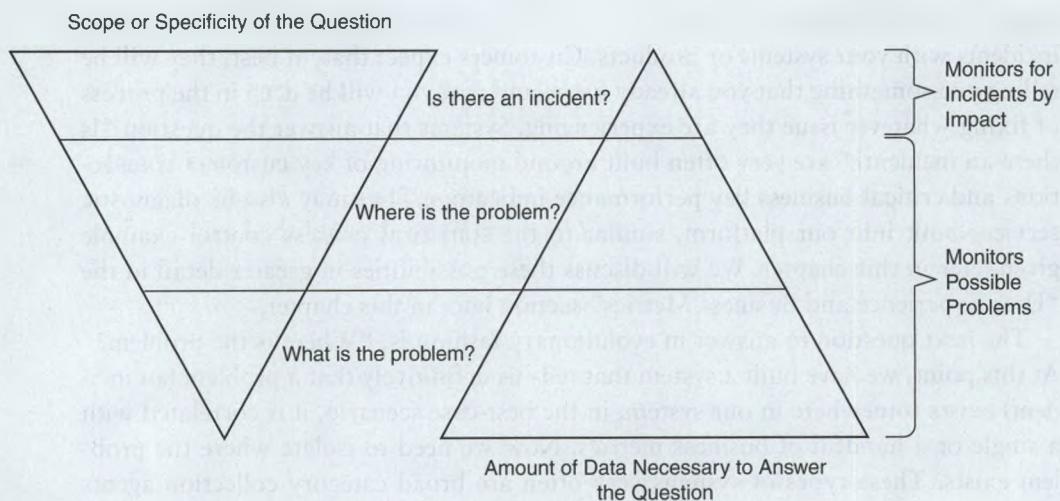


Figure 31.1 Correlation of Data Size to Problem Specificity

problems. Often, we can easily segment our three questions into three different types or approaches to monitoring. “Is there an incident?” can often be implemented by finding a handful of user experience or real-time business metrics monitors. “Where is the problem?” can often be accomplished by implementing out-of-the-box system-level monitors and third-party solutions that monitor our products through synthetic transactions. “What is the problem?” often relies on the way in which we log and collect data for our proprietary systems, including application log files and system resource utilization monitors.

The preceding approach is methodical, in that it forces us first to build systems that identify problems before we attempt to monitor everything within our platform or product. We do not mean to imply that absolutely no work should be done in answering the “Where is the problem?” and “What is the problem?” questions until the “Is there an incident?” investigation is finished; rather, we should focus on applying most of our effort first in answering the first question. Because the “Where is the problem?” solution is so easy to implement in many platforms, applying 10% of your initial effort to this question initially will pay huge dividends. At least 50% of your time in new-product monitoring should be spent identifying the right metrics and transactions to monitor so that you can always answer the question “Is there an incident?” Spend the remainder of your time building out problem monitors for the “What is the problem?” question, and expect that you will increase this percentage over time once your incident monitoring is complete.

User Experience and Business Metrics

User experience and business metric monitors are meant to answer the question “Is there an incident?” Often, you need to implement both of them to get a good view of the overall health of a system, but in many cases, you need only a handful of monitors to be able to answer the question of whether an incident exists with a high degree of certainty. For instance, in an ecommerce platform wherein revenue and profits are generated primarily from sales, you might choose to look at revenue, searches, shopping cart abandonment, and product views. You may decide to plot each of these in real time against 7 days ago, 14 days ago, and the average of the last 52 similar weekdays. Any significant deviation from a well-established curve may be used to alert the team that a potential problem is occurring.

Figure 31.2 displays a graph from an ecommerce site that plots its revenue over time and compares it to the same day in the previous week to determine possible incidents. Note that the current day experiences a significant issue at roughly 16:00 local time. As described in Chapter 6, Relationships, Mindset, and the Business Case, the delta between these lines can be used to calculate availability. From an incident monitoring perspective, however, the significant deviation in curve performance is a likely indicator of an incident.

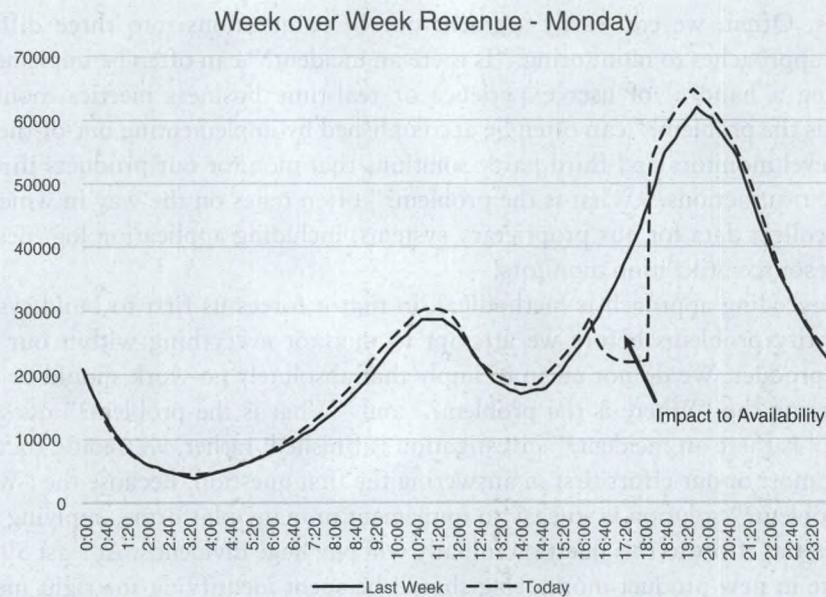


Figure 31.2 Example Incident Monitor of Revenue: Week over Week

Figure 31.3 shows a similar approach but leverages a statistical process control chart (SPCC). SPCC is a statistical method that allows us to plot upper and lower control limits, usually 3 standard deviations from a mean for a number of data points. The control limits are used to identify when something may be “out of control”—for our purposes, that means worthy of looking into as an incident. A variety of approaches may be taken to determine how many data points over which time interval constitutes a control violation. We will leave it to the reader to investigate SPCC in greater depth. It is worth noting, however, that SPCC-like approaches allow for the automation of alerts for incident monitors. In our case, we plotted our mean using 30 days of past “same day of week” data.

Incident monitors for advertising platforms may focus on cost per click by time of day, total clicks, calculated click-through rates, and bid-to-item ratios. These, too, may be plotted against the values from 7 days ago, 14 days ago, and the average of the last 52 similar weekdays. We may also apply SPCC against the average calculated. Again, the idea here is to identify major business and customer experience metrics that are both early and current indicators of problems. Third-party providers also offer *last mile* and *customer experience* monitoring solutions that are useful in augmenting business metrics and user experience monitoring. Last mile and user agent monitoring solutions from Keynote and Gomez, for example, help us better understand when customers at distant locations can’t access our services and when those services are performing below our expectations. User experience solutions such as CA’s Wily products

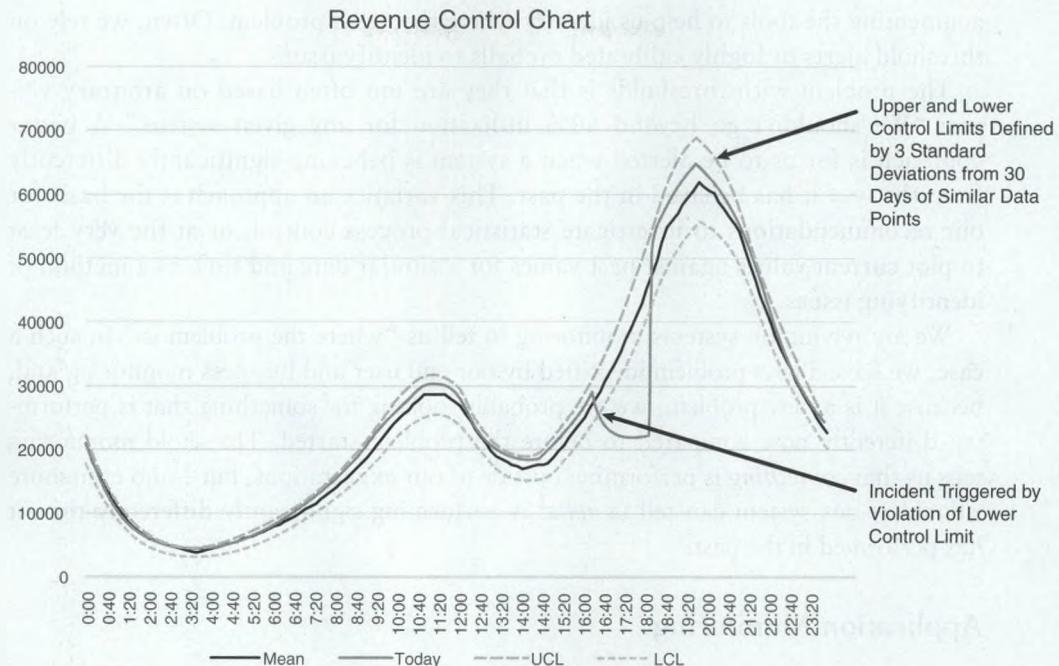


Figure 31.3 Statistical Process Control Chart Incident Monitor for Revenue

and Coradiant's products help us better understand customer interactions, actions, and perceived response times.

Other metrics and thresholds that might affect your business can be considered, such as response times and the like, but often these data points are more indicative of "where the problem is" rather than "whether a problem exists." The best metrics here are directly correlated to the creation of shareholder value. A high shopping cart abandonment rate and significantly lower click-through rates are both likely indicative of user experience problems that are negatively impacting your business.

Systems Monitoring

As we hinted earlier in this chapter, systems monitoring is one of the areas that companies tend to cover well. We use the term *systems monitoring* here to mean any grouping of hardware and software that shares several components. We might have a service consisting of several functions or applications running on a pool of servers and lump this loose group of hardware and software into a "system." Most monitoring systems and platforms have agents that do this type of monitoring fairly well right out of the box. You simply need to install the agents, configure them for your system, and plug them into your monitoring framework. Where we tend to fail is in

augmenting the tools to help us identify where there is a problem. Often, we rely on threshold alerts or highly calibrated eyeballs to identify issues.

The problem with thresholds is that they are too often based on arbitrary values: “We shouldn’t go beyond 80% utilization for any given system.” A better approach is for us to be alerted when a system is behaving significantly differently from the way it has behaved in the past. This variation in approach is the basis for our recommendations to investigate statistical process control, or at the very least to plot current values against past values for a similar date and time as a method of identifying issues.

We are relying on systems monitoring to tell us “where the problem is.” In such a case, we have a new problem identified by our end user and business monitoring and, because it is a new problem, we are probably looking for something that is performing differently now compared to before the problem started. Threshold monitoring tells us that *something* is performing outside of our expectations, but—and even more valuable—our system can tell us *what* is performing significantly differently than it has performed in the past.

Application Monitoring

Application monitoring is important to help us answer the question, “What is the problem?” Often, to determine “What is the problem?” we need to write some custom monitoring code. To do this well, we probably need to build the code into our product offering itself. Although some out-of-the-box agents will tell us exactly what the problem is, as in the case of a slow I/O subsystem caused by one or more bad disks, it is seldom the case that an out-of-the-box agent can help us diagnose precisely which part of our proprietary application has gone awry. Although self-healing applications are a bit of a pipe dream and not likely to be cost-effective in terms of development time, the notion that an application can be self-diagnosing for the most common types of failures is both an admirable and achievable aspiration.

For many companies, it is a relatively simple task, if undertaken early in development, to produce a set of reusable tools to help determine the cause of failures. These tools exist as services that are compiled or linked to the application or potentially as just services called by the application during run time. Often, they are placed at critical chokepoints within the application, such as when an error message is sent or when another service or resource is called. Error logging routines can be augmented to classify error types in a meaningful fashion and to keep track of the rate of error counts over time. Methods or functions can keep track of execution times by time of day and log them in a meaningful fashion for other processes to perform calculations. Remote service calls can log the response time of synchronous or asynchronous services on which an application relies.

Monitoring Issues and a General Framework

Most monitoring platforms suffer from two primary problems:

- The systems being monitored were not designed to be monitored.
- The approach to monitoring is bottom up rather than top down and misses the critical question “Is there a problem affecting customers right now?”

Solving these problems is relatively easy:

- Design your systems to be monitored before implementation.
- Develop monitors to first answer the question “Is there an incident?” These are typically business metric and customer experience monitors.
- Develop monitors to next answer the question “Where is the problem?” These are typically systems-level monitors.
- Develop monitors to finally answer the question “What is the problem?” Often, these solutions are monitors that you build into your application consistent with your design principles.

It is very important to follow the steps in order, from top down, to develop a world-class monitoring solution.

Measuring Monitoring: What Is and Isn't Valuable?

Remember Chapter 27, Too Much Data? In that chapter, we argued that not all data is valuable to the company and that all data has a cost. Well, guess what? The same is true for monitoring! If you monitor absolutely everything you can think of, there is a very real chance that you will use very little of the data that you collect. All the while, you will be creating the type of noise we described as being the harbinger of death for most monitoring platforms. Moreover, you are wasting employee time and company resources, which in turn costs your shareholders money.

The easiest way to determine which monitors provide real value and which do not is to step through our evolutionary monitoring framework in a top-down fashion and describe the value created by each of these tiers and how to limit the cost of implementation.

Our first question was “Is there an incident?” As we previously indicated, there are likely a handful of monitors—let’s say no fewer than three and no more than 10—that will serve as both predictive and current indicators that there will be or

currently is a problem. As the number of items that we are tracking is relatively small, data retention should not be a tremendous concern. It would be great to be able to plot this data in minute-by-minute or hourly records as compared to at least the last two weeks of data for similar days of the week. If today is Tuesday, we probably want the last two Tuesdays' worth of data. We probably should just keep that data for at least the last two weeks, but maybe we expand it to a month before we collapse the data. In the grand scheme of things, this data will not take a whole lot of space. Moreover, it will save us a lot of time in predicting and determining if there will be or currently is a problem.

The next question we ask is “Where is the problem?” In Figure 31.1, our pyramid indicates that although the specificity is narrowing, the amount of data is increasing. This should cause us some concern, as we will need many more monitors to answer this question. It is likely that the number of monitors is somewhere between one and two orders of magnitude (10 to 100 times) greater than our original sets of monitors. In very large, complex, and distributed systems, this might be an even larger number. We still need to compare the current information to that from previous similar days, ideally at a granular level. We'll need to be much more aggressive in our rollup and archival/deletion strategies, however. Ideally, we will summarize data potentially first by the hour and then eventually just move the data into a moving average calculation. Maybe we can plot and keep graphs but remove the raw data over time. We certainly do not want the raw data sitting around ad infinitum, as the probability that most of it will be used is low, the value is low, and the cost is high.

Finally, we come to the question of “What is the problem?” Again, we have at least an order of magnitude increase in data from our previous monitoring to address this question. We are adding raw output logs, error logs, and other data to the mix. The volume of this stuff grows quickly, especially in chatty environments. We probably hope to keep about two weeks of the data, where two weeks is determined by assuming that we will catch most issues within two weeks. You may have better information on what to keep and what to remove for your system, but again you simply cannot subject your shareholders to your desire to check up on anything at any time. That desire has a near-infinite cost and a very, very low relative return.

Monitoring and Processes

At last, we come to the point of how all of this monitoring fits into our operations and business processes. Our monitoring infrastructure is the lifeblood of many of our processes. The monitoring we perform to answer the questions for the continuum from “Is there an incident?” to “What is the problem?” will likely create the data necessary to inform the decisions within many of the processes we described

in Part II, “Building Processes for Scale,” and even some of the measurements and metrics we described within Chapter 5, Management 101.

The monitors that produce the data necessary to answer the question of “Is there an incident?” yield critical data for measuring our alignment to the creation of shareholder value. Recall that we discussed availability as a metric in Chapter 5. The goal is to be able to consistently answer “No” to the question of “Is there an incident?” If you can do that, then you have high availability. Measuring availability from a customer perspective and from a business metric perspective, as opposed to a technology perspective, gives you the tools needed both to answer the “Is there an incident?” question and to measure your system against your availability goal. The difference between revenue or customer availability and technology availability is important and drives cultural changes that have incredible benefit to the organization. Technologists have long measured availability as a product of the availability of all the devices within their care. That absolutely has a place and is important to such concerns as cost, mean time between failures, headcount needs, redundancy needs, mean time to restore, and so on. Nevertheless, it doesn’t really relate to what the shareholders or customers care about most—namely, whether the service is available and generating the greatest value possible. As such, measuring the experience of customers and the generation of profits in real time is much more valuable for both answering our first and most important monitoring question and measuring availability. With only a handful of monitors, we can satisfy one of our key management measurements, help ensure that we are identifying and reacting to impending and current events, and align our culture to the creation of shareholder and customer value.

The monitors that drive “Where is the problem?” are also very often the sources of data that we will use in our capacity planning and headroom processes (as described in Chapter 11, Determining Headroom for Applications). The raw data here will help us determine where constraints exist in our system and focus our attention on budgeting to horizontally scale those platforms or drive the architectural changes necessary to scale more cost-effectively. This information also helps feed the incident and crisis management processes described in Chapter 8, Managing Incidents and Problems, and Chapter 9, Managing Crises and Escalations. It is obviously useful during the course of an incident or crisis, and it definitely proves valuable during postmortem activities when we are attempting to discover how we could have isolated the incident earlier or prevented the incident from happening at all. The data also feeds into and helps inform changes to our performance testing processes.

The data that answers the question of “What is the problem?” is useful in many of the processes described for the “Where is the problem?” question. Additionally, it helps us test whether we are properly designing our systems to be monitored. The engineering staff should compare the output of our postmortems and operations reviews against the data and information we produce to help identify and diagnose

problems. The goal is to feed this information back into the code review and design review processes and thereby create better and more intelligent monitoring that helps us identify issues before they occur or isolate them more rapidly when they do occur.

That leaves us with the management of incidents and problems, as identified in Chapter 8, and the management of crises and escalations, as identified in Chapter 9. In the ideal world, incidents and crises can be predicted and avoided through a robust and predictive monitoring solution; in the real world, at the very least they should be identified at the point at which they start to cause customer problems and impact shareholder value. In many mature monitoring solutions, the monitoring system itself will be responsible not only for the initial detection of an incident but also for the reporting or recording of that incident. In this fashion, the monitoring system is responsible for both the detect and report steps of the DRIER model (introduced in Chapter 8).

Conclusion

This chapter discussed monitoring. The primary reasons for the repeated failures of most monitoring initiatives and platforms are that our systems are not designed to be monitored and that our general approach to monitoring is flawed. Too often, we attempt to monitor from the bottom up, starting with individual agents and logs rather than attempting to first create monitors that answer the question, “Is there an incident?”

The best organizations design their monitoring platforms from the top down. These systems are capable, with a high degree of accuracy, of answering the first question, “Is there an incident?” Ideally, these monitors are tightly aligned with the business and technology drivers that create shareholder value. Most often, they take the form of real-time monitors on transaction volumes, revenue creation, cost of revenue, and customer interactions with the system. Third-party customer experience systems can be employed to augment real-time business metric systems to answer this most important question.

The next step is to build systems to answer the question, “Where is the problem?” Often, these systems are out-of-the-box third-party or open source solutions that you install on systems to monitor resource utilization. Some application monitors might also be employed. The data collected by these systems help inform other processes, such as the capacity planning process and the problem resolution process. Care must be taken to avoid a combinatorial explosion of data, as that data is costly and the value of immense amounts of old data is very low.

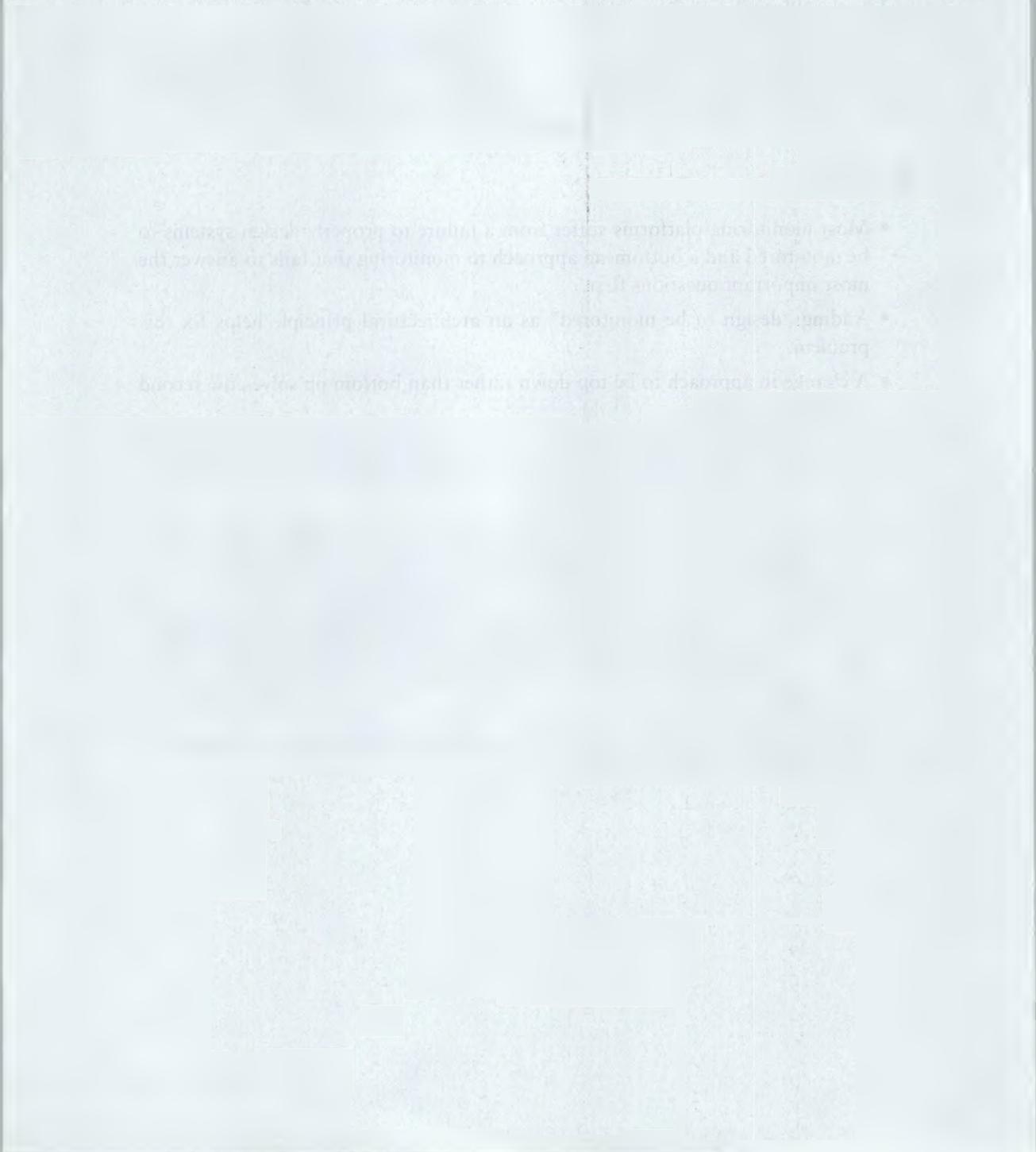
Finally, we move to the question of “What is the problem?” Answering it very often requires us to rely heavily on the architectural principle of “design to be

monitored.” Here, we monitor individual components, and often these are proprietary applications for which we are responsible. Again, the concerns about explosive growth of data are present, and we must fight to ensure that we keep the right data and do not dilute shareholder value.

Focusing first on “Is there an incident?” will pay huge dividends throughout the life of your monitoring system. It is not necessary to focus 100% of your monitoring efforts on answering this question, but you should certainly spend a majority (50% or more) of your time on the question until you have it absolutely nailed.

Key Points

- Most monitoring platforms suffer from a failure to properly design systems to be monitored and a bottom-up approach to monitoring that fails to answer the most important questions first.
- Adding “design to be monitored” as an architectural principle helps fix this problem.
- A change in approach to be top down rather than bottom up solves the second half of the problem.
- Answering the questions of “Is there an incident?”, “Where is the problem?”, and “What is the problem?”—in that order—when designing a monitoring system is an effective top-down strategy.
- “Is there an incident?” monitors are best implemented by aligning the monitors to the measurements of shareholder and stakeholder value creation. Real-time business metrics and customer experience metrics should be employed.
- “Where is the problem?” monitors may very well be out-of-the-box third-party or open source solutions that are relatively simple to deploy. Be careful with data retention and attempt to use real-time statistics when employing these measurements.
- “What is the problem?” monitors are most likely homegrown and integrated into your proprietary application.



Chapter 32

Planning Data Centers

Having collected an army and concentrated his forces, he must blend and harmonize the different elements thereof before pitching his camp.

—Sun Tzu

This chapter covers data centers, whether owned or leased, and the use of Infrastructure as a Service (IaaS) solutions. The intent is to give you the tools necessary to be successful in designing highly available solutions at relatively low cost in a multiparadigm infrastructure-hosting world.

Data Center Costs and Constraints

In the last 15 years, something in data centers changed so slowly that few if any of us caught on until it was too late. Just as a military sniper moves very slowly into a firing position while the enemy watches but remains unaware of his presence, so did power consumption, constraints, and costs sneak up on us. Once we finally recognized this sea change, all of us scrambled to change our data center capacity planning models.

For years, processors have increased in speed, as observed by Gordon Moore and as described in Moore's law. This incredible increase in speed resulted in computers and servers drawing more power over time. The ratio of clock speed to power consumption varies with the technologies employed and the types of chips. Some chips employed technology to reduce clock speed and hence power consumption when idle, and multicore processors allegedly have lower power consumption for higher clock speeds. But given similar chip set architectures, a faster processor will typically mean more power consumption.

Until the mid-1990s, most data centers had enough power capacity that their primary constraint was the number of servers one could shoehorn into the footprint or square footage of the data center. As computers decreased in size (measured in

rack units [U]) and increased in clock speed, the data center became increasingly efficient. Efficiency here is measured strictly against the computing capacity per square foot of the data center, with greater efficiency realized through more computers crammed into the same square footage and with each computer having more clock cycles per second to perform work. This increase in computing density also increased power usage on a per square foot basis. Not only did the computers draw more power per square foot, but they also generated more heat. This required more HVAC services to cool the area, which in turn consumed even more power. If you were lucky enough to be in a colocation facility with a contract wherein you were charged by square foot of space, you weren't likely aware of this increase in cost; the colocation facility owner ate the cost, which decreased the margins for its services. If you owned your own data center, more than likely the facilities team identified the steady but slow increase in cost but did not pass that information along to you until you needed to use more space and found out that you were out of power.

Rack Units

A rack unit (U or RU) is a measurement of height in a 19-inch- or 23-inch-wide rack. Each unit equals 1.75 inches in height. A 2U server, therefore, is 3.5 inches in height. The term *half rack* is an indication of width rather than height; this term is applied to a 19-inch rack. A half rack server or component, then, is 9.5 inches wide.

This shift to a world in which power utilization suddenly constrained growth created a number of interesting problems. The first problem manifested itself in an industry once based upon square footage assumptions. Colocation and managed server providers found themselves in contracts for square footage largely predicated on the number of servers. As previously indicated, the increase in power utilization decreased the margins for their services until the provider could renegotiate contracts. The buyers of the colocation facilities, in turn, looked to move them to locations where power density was greater. Successful providers of services changed their contracts to charge for both space and power, or strictly upon power used. The former approach allowed companies to flex prices based on the price of power, thereby lowering the variability within their operating margins, whereas the latter approach attempted to model power costs over time to ensure that the companies were always profitable. Often, both types of contractual arrangements would reduce the power and space component to a number of racks and power utilization per rack within a given footprint in a data center.

Companies that owned their own data centers and found themselves constrained by power could not build new data centers quickly enough to allow themselves to grow. Consequently, they would utilize colocation facilities until such time as they could build new data centers.

Regardless of whether one owned or leased space, the world changed underneath our feet. In the new world order, power—not space—dictated capacity planning for data centers. This led to the emergence of some other important aspects of data center planning, not all of which have been fully embraced or recognized by every company.

Location, Location, Location

You've probably heard the real estate mantra, "Location, location, location." Nowhere is this mantra more important these days than in the planning of owned or rented space for data centers. Data center location has an impact on nearly everything we do, from fixed and variable costs to quality of service to the risk we impose upon our businesses.

With the shift in data center constraints from location to power came another shift—this time, in the fixed and variable costs of companies' product offerings. Previously, when space was the primary driver and limitation of data center costs and capacity, location was still important, but for very different reasons. When power was not as large a concern as it is today, companies simply sought to build data centers in a place where land and building materials were cheap. As a result, data centers might be built in major metropolitan areas where land was abundant and labor costs were low. Often, companies would factor in an evaluation of geographic risk. As a consequence, areas such as Dallas, Atlanta, Phoenix, and Denver became very attractive. Each of these areas offered plenty of space for data centers, the needed skills within the local population to build and maintain them, and low geographic risk.

Smaller companies that rented or leased space and services and were less concerned about risk would look to locate data centers close to their employee population. This led to colocation providers building or converting facilities in company-dense areas like the Silicon Valley, Boston, Austin, and the New York/New Jersey area. These smaller companies favored ease of access to the servers supporting their new services over risk mitigation and cost. Although the price was higher for the space as compared to the lower-cost alternatives, many companies felt the benefit of proximity overcame the increase in relative cost.

When power became the constraining factor for data center planning and utilization, companies started shifting their focus to areas where they could not only purchase land and build centers for an attractive price, but also obtain power at a

relatively low price. An additional focus became where, geographically, power could be used most efficiently. This last point actually leads to some counterintuitive locations for data centers.

Air conditioning (the “AC” portion of HVAC) operates most efficiently at lower elevations above sea level. We won’t go into the reasons why, as there is plenty of information available on the Internet to confirm the statement. Air conditioning also operates more efficiently in areas of lower humidity, as less work is performed by the air conditioner to remove humidity from the air being conditioned. Low elevation above sea level and low humidity work together to produce a more efficient air-conditioning system; the system, in turn, draws less power to perform a similar amount of work. This factor explains why areas like Phoenix, Arizona, which is a net importer of power and as such sometimes has a higher cost of power, remain favorites among companies building data centers. Although the per unit cost of power and summertime cooling demands are higher in Phoenix than in other areas, the annualized efficiency of the HVAC systems, driven by the low winter months’ demand, reduces overall power consumption, making it an attractive area to consider.

Some other interesting areas also emerged as great candidates for data centers due to an abundance of low-cost power. The area served by the Tennessee Valley Authority (TVA), including the state of Tennessee, parts of western North Carolina, northwest Georgia, northern Alabama, northeast Mississippi, southern Kentucky, and southwest Virginia is one such region. Another favorite of companies building data centers is the region called the Columbia River Gorge between Oregon and Washington. Both areas have an abundance of low-cost power thanks to their hydroelectric power plants.

Location also impacts companies’ quality of service. Companies want to be in an area that has easy access to quality bandwidth, an abundance of highly available power, and an educated labor pool. They would like the presence of multiple carriers to reduce their transit or *Internet pipe* costs. When multiple carriers are available, companies can potentially pass their traffic over at least one carrier in the event that another carrier is having availability or quality problems. Thus, companies want the power infrastructure not only to be comparatively low cost, but also highly available with a low occurrence of interruptions due to age of the power infrastructure or environmental concerns.

Finally, location affects companies’ risk profile. If our organization is operating in a single location with a single data center and that location has high geographic risk, the probability that we might suffer an extended outage increases. A geographic risk can be anything that causes structural damage to the data center, power infrastructure failures, or network transport failures. The most commonly cited geographic risks to data centers and businesses are earthquakes, floods, hurricanes, and

tornadoes. Other location-specific risks must also be considered. For example, high crime rates in an area might lead to fears about interrupting services. Extremely cold or hot weather that taxes the location's power infrastructure might cause an interruption to operations. Areas that are attractive to terrorists for geopolitical reasons carry higher geographic risk as well.

Even within a general geographic region, some areas have higher risks than others. Proximity to freeways can increase the likelihood of a major accident causing damage to our facility or increase the likelihood that our facility might have to be evacuated due to a chemical spill. Do we have quick and easy access to fuel sources in the area should we need to use our backup generators? Does the area allow for easy access for fuel trucks for our generators? How close are we to a fire department?

Although location isn't everything, it absolutely impacts several areas critical to a company's cost of operations, quality of service, and risk profile. The right location can reduce fixed and variable costs associated with power usage and infrastructure, increase quality of service, and reduce an organization's risk profile.

In considering cost, quality of service, and risk, there is no single panacea. In choosing to go to the Columbia River Gorge or TVA areas, you will be reducing your costs at the expense of needing to train the local talent or potentially bring in your own talent, as many companies have done. In choosing Phoenix or Dallas, you will have access to an experienced labor pool but will pay more for power than you would in either the TVA or Columbia River Gorge area. There is no single right answer when selecting a location; you should work to optimize the solution to fit your budget and needs. There are, however, several wrong answers in our minds. We always suggest to our clients that they never choose an area of high geographic risk unless there simply is no other choice. Should they choose an area of high geographic risk, we always ask that they create a plan to get out of that area. It takes only one major outage to make the decision a bad one, and the question is always when—rather than if—that outage will happen.

Location Considerations

When considering a location for your data center or colocation partner, the following are some things to ponder:

- What is the cost of power in the area? Is it high or low relative to other options? How efficiently will your HVAC run in the area?
- Is there an educated labor force in the area from which you can recruit employees? Are they educated and experienced in the building and operation of a data center?

- Are there a number of transit providers in the area, and how good has their service been to other consumers of their service?
- What is the geographic risk profile in the area?

Often, you will find yourself making tradeoffs between questions. You may find an area with low power costs, but without an experienced labor pool. The one area we recommend never sacrificing is geographic risk.

Data Centers and Incremental Growth

Data centers and colocation space present an interesting dilemma to incremental growth, and that dilemma is even more profound for companies experiencing moderate growth than it is for companies experiencing rapid or hyper-growth. Data centers are, for many of our technology-focused clients, what factories are for companies that manufacture products: They are a way to produce the product, they are a limitation on the quantity that can be produced, and they are either accretive or dilutive to shareholder value, depending on their level of utilization.

Taking and simplifying the automotive industry as an example, new factories are initially dilutive as they represent a new expenditure of cash by the company in question. The new factory probably has the newest available technology, which in turn is intended to decrease the cost per vehicle produced and ultimately increase the gross margin and profit margins of the business. Initially, the amortized value of the factory is applied to each of the cars produced and the net effect is dilutive, because initial production quantities are low and each car loses money for the company. As the production quantity increases and the amortized fixed cost of the factory is divided by an increasing volume of cars, the profit of those cars in the aggregate starts to offset the cost and finally overcome it. The factory starts to become accretive when its cost per car produced is lower than that of the factory with the next lowest cost per car produced. Unfortunately, to hit that point, we often have to run the factory at a fairly high level of utilization.

The same holds true with a data center. The building of a data center usually represents a fairly large expenditure for any company. For smaller companies that are leasing new or additional data center space, that space still probably represents a fairly large commitment for the company in question. In many, if not most, of the cases where building new space, rather than purchasing or leasing it, is being considered, we will likely put better and faster hardware in the new space than we had in most of our older data center space. Although this move will likely increase the power utilization and the associated power costs, the hope is that we will reduce our overall spending

by doing more with less equipment in our new space. Even so, we will have to utilize some significant portion of this new space before the recurring lease and power costs or amortized property, plant, and equipment costs will be offset by the new transactions.

To illustrate this point, let's make up some hypothetical numbers. For this discussion, reference Table 32.1. Suppose you run the operations of a fictitious company, AllScale Networks, and that you currently lease 500 square feet of data center and associated power at a total cost of \$3,000 per month "all in." You are currently constrained by power within your 500 square feet and need to consider additional space quickly before your systems are overwhelmed by user demand. You have options to lease another 500 square feet at \$3,000 per month or another 1,000 square feet at \$5,000 per month.

The cost to build out the racks and power infrastructure (but not the server or network gear) is \$10,000 for the 500 square feet versus \$20,000 for the 1,000 square feet. The equipment that you expect to purchase and put into the new space has been tested for your application, and you believe that it will handle about 50% more traffic or requests than your current systems (indexed at 1.5, the original 500 square foot Request Index in Table 32.1). It will, however, draw about 25% more power to do so (indexed at 1.25% of the original Power Index in Table 32.1). Given that power density is the same for the cages, you can rack only roughly 80% of the systems you previously had, as each system draws 1.25.: the power of the previous systems. These are represented as 0.8 and 1.6 under Space Index in Table 32.1 for the 500 square foot and 1,000 square foot options, respectively. The resulting performance efficiency is $(0.8 \div 1.5) = 1.2$.: the throughput for the 500 square foot option and 2.4.: for the 1,000 square foot option referenced as the Performance Index.

Finally, the performance per dollar spent, as indexed to the original 500 square foot cage, is 1.2 for the 500 square foot option and 1.44 for the 1,000 square foot option. This change is due to the addition of 500 square feet more space at a reduced price of \$2,000 for the 1,000 square foot option.

Note that we've ignored the original build-out cost in this calculation, but you could amortize that figure over the expected life of the cage and include it in all of

Table 32.1 Cost Comparisons

Cage	Cost per Month	Request Index	Power Index	Space Index	Performance Index	Performance Per Dollar
Original 500 sq. ft.	\$3,000	1.0	1.0	1.0	1.0	1.0
Additional 500 sq. ft.	\$3,000	1.5	1.25	0.8	1.2	1.2
Additional 1,000 sq. ft.	\$5,000	1.5	1.25	1.6	2.4	1.44

the numbers in the table. We also assume that you will not replace the systems in the older 500 square foot cage, and we have not included the price of the new servers.

It previously took you two years to fill up your 500 square feet, and the business believes the growth rate has doubled and should stay on its current trajectory. All indicators are that you will fill up another 500 square feet in about a year. What do you do?

We won't answer that question, but rather leave it as an exercise for you. The answer, however, is financially based if answered properly. It should consider how quickly the data center becomes accretive or margin positive for the business and shareholders. You should also factor in lost opportunities for building out data center space twice rather than once. It should be obvious by now that data center costs are "lumpy," meaning that they are high relative to many of your other technology costs and take some planning to ensure that they do not negatively impact your operations.

How about our previous assertion that the problem is larger for moderate-growth companies than for hyper-growth companies? Can you see why we made that statement? The answer is that the same space purchased or leased by a hyper-growth company will become accretive more rapidly than the space obtained by a moderate-growth company. As such, space considerations are much more important for slower-growth companies unless they expect to exit other facilities and close them over time. The concerns of the hyper-growth company focus more intently on staying well ahead of the demand for new space than on ensuring the space hits the accretive point of utilization.

One recent addition to data center considerations, the renting of infrastructure in the "cloud" (Infrastructure as a Service), is important to consider in this discussion of data center usage.

When Do I Consider IaaS?

In this section we discuss when to consider an Infrastructure as a Service (IaaS) strategy versus a colocation or data center hosting strategy. Many large companies, such as Netflix, run a significant portion of their product and service offerings within solutions such as Amazon Web Services.¹ IaaS can play a role in replacing data centers or colocation facilities for nearly any company. When helping companies decide whether to rent capacity (using IaaS), lease space and provision capacity (colocation facilities), or own the solution (wholly owned data centers), we evaluate the size of the company, the risk associated with the product, and the utilization of the servers running the product.

1. "AWS Case Study: Netflix." <http://aws.amazon.com/solutions/case-studies/netflix/>; accessed September 29, 2014.

The size of a company is an important factor when deciding how to approach the infrastructure and data centers for your product and services. Small companies, for instance, really can't "own" a physical data center, because the capital costs of ownership are prohibitive. Recognizing this fact, here we look to risk and server utilization to make a decision between leasing space and leveraging IaaS completely. For brand-new startups, because product risk is high (measured by the probability of product adoption and ultimate success), we prefer to start products within an IaaS solution. When you're just starting out, why commit a large portion of your limited capital on infrastructure? Figure 32.1 depicts our evaluation of renting (IaaS/public cloud), leasing (colocation facility with either owned or leased equipment), and owning (having your own data center) as a function of the level of risk within your product.

For small but growing companies with maturing products and comparatively lower risk, we evaluate server counts and server utilization to help us make the decision. If the product or service keeps servers relatively busy throughout the day and the server count is high, you may be able to save money by leasing space in a colocation facility. If server counts are low or servers are not utilized around the clock, it may make sense to run everything (or almost everything) within an IaaS provider.

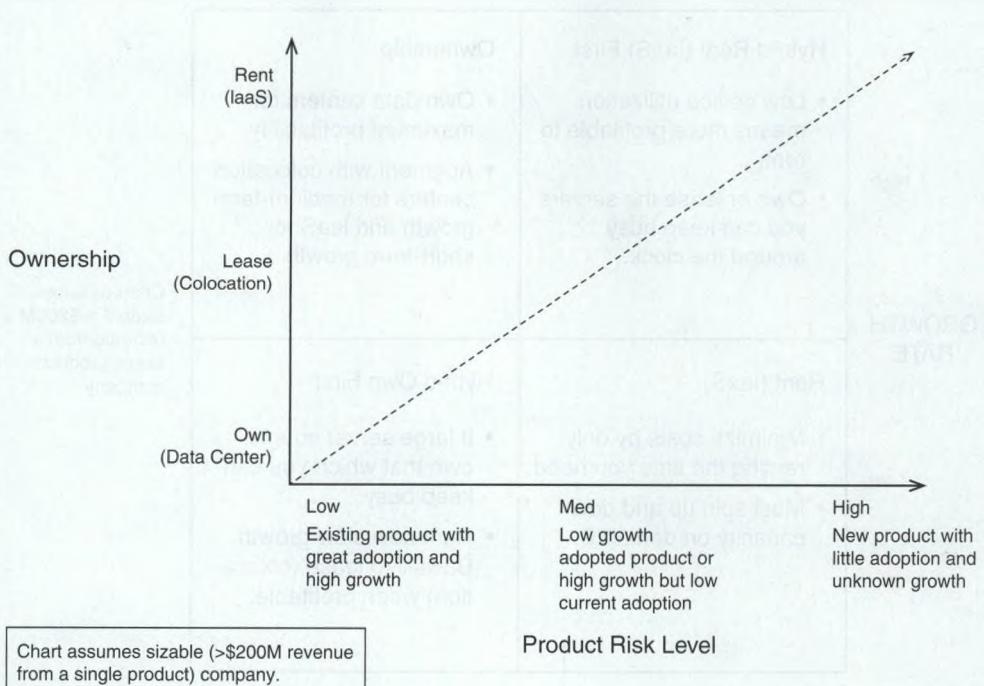


Figure 32.1 Own/Lease/Rent Options Plotted Against Product Risk

As companies grow, and their product risk decreases and their server utilization increases, it becomes more cost-effective to run solutions within a wholly owned data center. The decisions here are primarily financial in nature. Very large companies that can keep servers busy and efficiently use the power and space within a data center usually can do so with a lower cost structure than if they lease or rent the same capacity. Figure 32.2 depicts how we view the own/lease/rent decision as compared to product growth and device utilization.

Figures 32.1 and 32.2 aren't meant to be the final determining factors of what you should do with your data center and device strategy. Rather, they are meant to help foster conversations around the different approaches and the inherent tradeoffs in those approaches. The right decision, as always, should be based on your company's financial objectives from both growth and profitability perspectives.

The two right quadrants of Figure 32.2 suggest owning only that which you can keep busy. The implication here is that you should "rent" any portion that your company cannot keep busy. Figure 33.3 depicts what this may mean. The x-axis of Figure 33.3 shows time and the y-axis shows rate of requests to a given service or product. The graph is equally valuable in discussing activity level through a 24-hour period and activity level

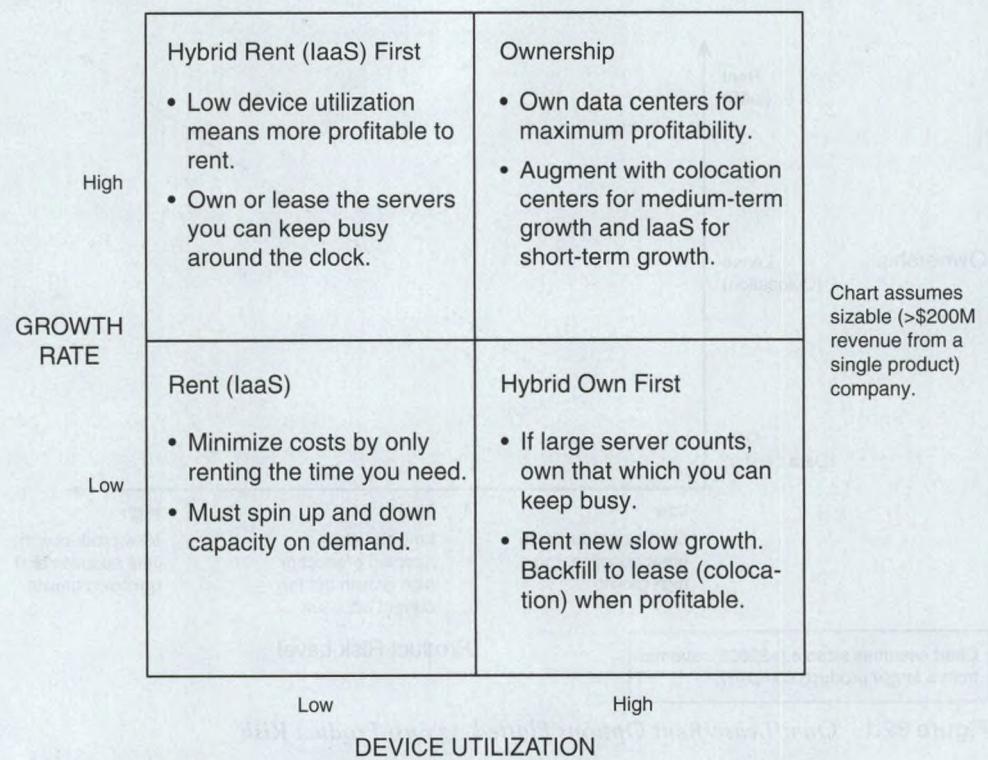


Figure 32.2 Own/Lease/Rent Options Plotted Against Growth and Utilization

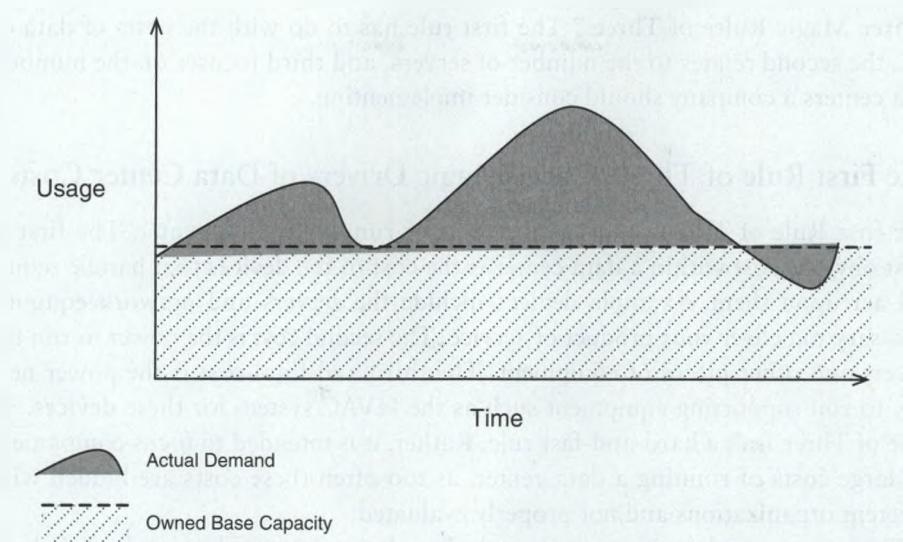


Figure 32.3 Daily/Monthly Usage and Elastic Expansion Considerations

over the course of a year. As such, the x -axis can either be utilization of devices during the course of a day or an indicator of seasonality through the course of a year.

The lightly shaded area below the dotted line at the base of the curve in Figure 33.3 indicates the “mostly busy” area of your service. This is the area where it makes sense to “own” or “lease” (colocation) if you have the capital and your company is of the appropriate size to do so. Clearly, small startups won’t gain an advantage here: They simply can’t buy a data center and keep it busy, and they don’t have the negotiation leverage with a colocation provider. Larger companies, however, can usually see somewhere between marginal and significant improvements in their profit and loss statements when they own the assets and can keep them busy in either a colocation or a wholly owned data center.

The dark area above the dotted line in Figure 33.3 indicates that area where it makes sense to “rent” capacity. Again, this might be for a total of 8 or so hours per day, or over the course of several months of the year. Either way, as long as the rent is less than the cost of owning the asset around the clock, it makes financial sense to do so. An implication here, however, is that we must have the discipline to “spin up” and “spin down” servers when we need them and no longer need them, respectively.

Three Magic Rules of Three

We love simple, easily understood and communicated rules, and that includes our Rules of Three as applied to data centers. There are three of these rules—hence the

“Three Magic Rules of Three.” The first rule has to do with the costs of data centers, the second relates to the number of servers, and third focuses on the number of data centers a company should consider implementing.

The First Rule of Three: Three Magic Drivers of Data Center Costs

Our first Rule of Three concerns the costs of running a data center. The first and most obvious cost within a data center is the cost of the devices that handle requests and act upon them. Example devices include the servers and network equipment necessary to deliver your product or service. The second cost is the power to run these servers and other pieces of equipment. The third and final cost is the power necessary to run supporting equipment such as the HVAC system for these devices. This Rule of Three isn’t a hard-and-fast rule. Rather, it is intended to focus companies on the large costs of running a data center, as too often these costs are hidden within different organizations and not properly evaluated.

These costs tend to increase directly in relationship to the number of devices deployed. Each device obviously has its own cost of acquisition and cost for associated power. The supporting infrastructure (HVAC) needs also typically increase linearly with the number of devices and the power consumption of those devices. More devices drawing more power create more heat that needs to be moved, reduced, and conditioned. In many companies, especially larger companies, this relationship is lost within organizational budget boundaries.

There are other obvious costs not included within this first Rule of Three. For instance, we need headcount to run the data center. This headcount may be in the form of full-time salaried employees, but it may also be built into a colocation contract for services, or implicitly built into the fees our company pays for public cloud usage. Other costs include network transit costs that allow us to communicate to our clients over the Internet. There may also be security costs, the costs of maintaining certain pieces of equipment such as FM-200 fire suppression devices, and so on. These costs, however, tend to be well understood and are often either fixed by the amount of area, as in security and FM-200 maintenance, or clearly within a single organization’s budget, such as network transit costs.

The Second Rule of Three: Three Is the Magic Number for Servers

Many of our clients have included the second Rule of Three as a “magic rule” embodied in an architectural principle. Simply put, it states that the number of servers for any service should never fall below three and that, when planning data center capacity, you should consider all of the existing services and future or planned services and expect that there will be at least three servers for any service. The thought here

is that you build one server for the customer, one server for capacity and growth, and one server to fail. Ideally, the service will be built and the data center planned in such a way that services can expand horizontally per our “Scale out, not up” principle (introduced in Chapter 12, Establishing Architectural Principles).

Taken to its extreme for hyper-growth sites, this rule would be applied to data center capacity planning not only for front-end Web services, but also for data storage services such as a database. If a service requires a database, and if finances allow, the service should be architected such that at the very least it is supported by a write database for writes and load-balanced reads, an additional read database, and a database that can serve as a logical standby in the case of corruption. A fault-isolative architecture or swim lane architecture by service may incorporate several of these database implementations.

It’s important to note that no data center decisions should be made without consulting the architects, product managers, and capacity planners responsible for defining, designing, and planning new and existing systems.

The Third Rule of Three: Three Is the Magic Number for Data Centers

“Whoa, hang on there!” you might say. “We are a young company attempting to become profitable and we simply cannot afford three data centers!” What if we told you that you can run out of three data centers for close to the cost that it takes you to run out of two data centers? Thanks to the egalitarianism of facilities created by IaaS (“public cloud”) providers, the playing field for distribution of data and services has been leveled. Your organization no longer needs to be a *Fortune 500* company to spread its assets around and be close to its customers. If your organization is a public company, you certainly don’t want to make the public disclosure that “Any significant damage to our single data center would significantly hamper our ability to remain a going concern.” If it’s a startup, can you really afford a massive customer departure early in your product life cycle because you didn’t think about how to enable disaster recovery? Of course not.

In Chapter 12, we suggested designing for multiple live sites as an architectural principle. To achieve this goal, you need either stateless systems or systems that maintain state within the browser (say, with a cookie) or pass state back and forth through the same URL/URI. After you establish affinity with a data center and maintain state at that data center, it becomes very difficult to serve the transaction from other live data centers. Another approach is to maintain affinity with a data center through the course of a series of transactions, but allow a new affinity for new or subsequent sessions to be maintained through the life of those sessions. Finally, you can consider segmenting your customers by data center along a z-axis split, and

then replicate the data for each data center, split evenly through the remainder of the data centers. In this approach, should you have three data centers, 50% of the data from data center A would move to data centers B and C. This approach is depicted in Figure 32.4. The result is that you have 200% of the data necessary to run the site in aggregate, but each site contains only 66% of the necessary data—that is, each site contains the copy for which it is a master (33% of the data necessary to run the site) and 50% of the copies of each of the other sites (16.5% of the data necessary to run the site, for a total of an additional 33%).

Let's discuss the math behind our assertion. We will first assume that you agree with us that you need at least two data centers to ensure that you can survive any disaster. If these data centers were labeled A and B, you might decide to operate 100% of your traffic out of data center A and leave data center B as a warm standby. Back-end databases might be replicated using native database replication or a third-party tool and may be several seconds behind the primary database. You would need 100% of your computing and network assets in both data centers—that is, 100% of your Web and application servers, 100% of your database servers, and 100% of your network equipment. Power needs would be similar and Internet connectivity would be similar. You probably keep slightly more than 100% of the capacity necessary to serve your peak demand in each location so that you could handle surges in demand. So, let's say that you keep 110% of your needs in both locations. Whenever

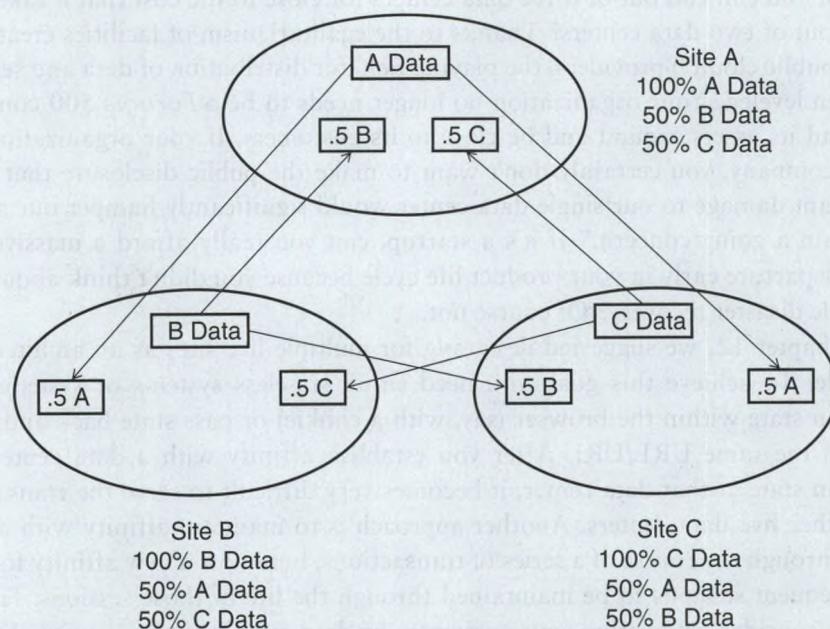


Figure 32.4 Split of Data Center Replication

you buy additional servers for one place, you have to buy the same servers for the other data center. You may also decide to connect the data centers with your own dedicated circuits for the purposes of secure replication of data. Running live out of both sites would help you in the event of a major catastrophe, as only 50% of your transactions would initially fail until you transfer that traffic to the alternate site, but it won't help you from a budget or financial perspective. A high-level diagram of the data centers might look like Figure 32.5.

However, if we have three sites and we run live out of all three sites at once, the cost of those systems goes down. In this scenario, for all non-database systems, we really need only 150% of our capacity in each location to run 100% of our traffic in the event of a site failure. For databases, we definitely need 200% of the storage as compared to one site, but we really need only 150% of the processing power if we are smart about how we allocate our database server resources. Power and facilities consumption should also be roughly 150% of the need for a single site, although obviously we will need slightly more people and probably slightly more overhead than 150% to handle three sites versus one. The only area that increases disproportionately are the network interconnections, as we need two additional connections for three sites. Such a data center configuration is depicted in Figure 32.6; Table 32.2 shows the relative costs of running three sites versus two. In Table 32.2, note that we have figured each site has 50% of the server capacity necessary to run everything, and 66% (66.66, but we've made it a round number and rounded down rather than up in the figure) of the storage per Figure 32.6. You would need 300% of the storage if you were to locate 100% of the data in each of the three sites.

Note that we get this leverage in data centers because we expect the data centers to be sufficiently far apart so as not to have two data centers simultaneously eliminated as a result of any geographically isolated event. You might decide to stick one near the West Coast of the United States, one in the center of the country, and another near the East Coast. Remember, however, that you still want to reduce your data center power costs and reduce the risks to each of the three data centers; thus you

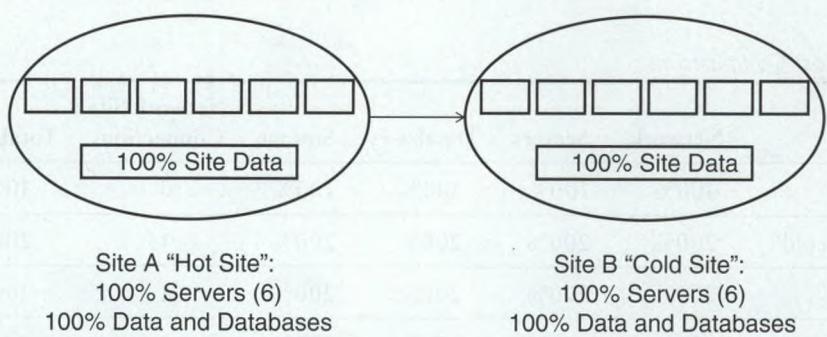


Figure 32.5 Configuration with Two Data Centers: Hot and Cold Sites

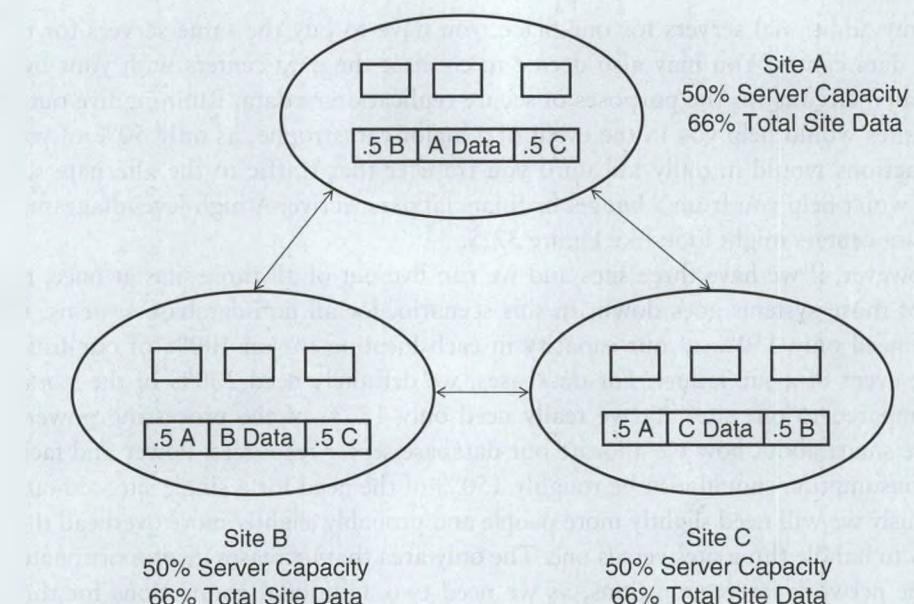


Figure 32.6 Configuration with Three Data Centers: Three Hot Sites

still want the data centers to be located in areas with a low relative cost of power and a low geographic risk.

Maybe now you are a convert to our three-site approach and you immediately jump to the conclusion that more is better! Why not four sites, or five, or 20? Well, more sites are better, and you can certainly play all sorts of games to further reduce your capital costs. But at some point, unless your organization is a very large company, the management overhead of a large number of data centers becomes cost prohibitive. Each additional data center will provide for some reduction in the amount of equipment that you need for complete redundancy, but will increase the management overhead and network connectivity costs. To arrive at the “right number” for

Table 32.2 Cost Comparisons

Site Configuration	Network	Servers	Databases	Storage	Network Site Connections	Total Cost
Single site	100%	100%	100%	100%	0	100%
2 sites: “hot”/“cold”	200%	200%	200%	200%	1	200%
2 sites: live/live	200%	200%	200%	200%	1	200%
3 sites: live/live/live	150%	150%	150%	200%	3	~166%

your company, you should take the example in Table 32.2 and add in the costs to run and manage the data centers to determine the appropriate answers.

Recall our discussion about the elasticity benefits of IaaS: You need not “own” everything. Depending on your decisions based on Figures 32.1 and 32.2, you might decide to “rent” each of your three data centers in various geographic locations (such as Amazon AWS’s “regions”). Alternatively, you might decide to employ a hybrid approach in which you keep a mix of colocation facilities and owned data centers, then augment them with dynamic scaling into the public cloud on a seasonal or daily basis (recall Figure 32.3). While you are performing your cost calculations, remember that use of multiple data centers has other benefits, such as ensuring that those data centers are close to end-customer concentrations to reduce customer response times. Our point is that you should plan for at least three data centers to both prevent disasters and reduce your costs relative to a two-site implementation.

Multiple Active Data Center Considerations

We hinted at some of the concerns of running multiple active data centers in our earlier discussion of why three is the magic number for data centers. In this section, we first cover some of the benefits and concerns of running multiple live data centers, and then we discuss three flavors of approaches and the concerns unique to each of those approaches.

There are three great benefits of running multiple live data centers. First, you get the benefit of disaster recovery—or as we prefer to call it, disaster prevention. Second, a cost reduction is associated with running multiple live data centers relative to having only a primary data center and one backup data center. Third, running multiple data centers gives you the flexibility of putting data centers closer to your customers, thereby reducing response times to their requests.

A multi-data-center approach does not eliminate the benefits you would realize by deploying a content delivery network as described in Chapter 25, Caching for Performance and Scale, but it does benefit those calls that are forced to go directly to the data center due to their dynamic nature. If you are leasing data center space, you also get the benefit of being able to multisource your colocation partners and, as a result, can use the market to drive down the negotiated price for your space. Should you ever need or desire to leave one location, you can run live out of two data centers and move to a lower-cost or higher-quality provider for your third data center. If your organization is a SaaS company, you may find it easier to roll out or push updates to your site by moving traffic between data centers and upgrading sites one at a time during off-peak hours. Finally, when you run live out of multiple data centers, you don’t find yourself questioning the viability of your warm or cold “disaster

recovery site”; instead, your daily operations prove that each of the sites is capable of handling requests from your customers.

Multiple live data centers do add some complexity and will likely increase your head-count needs as compared to running a single data center and maybe even two data centers. The increase in headcount should be moderate, potentially adding one to a few people to manage the contracts and space depending on the size of your company. Some of your processes will need to change, such as how and when you roll out code and how you ensure that multiple sites are roughly consistent with respect to configuration. In addition, members of your team may need to travel more often to visit and inspect sites should you not have full-time employees dedicated to each of those centers. Network costs are also likely to be higher as you add network links between sites for intersite communication.

From an architecture perspective, to gain the full advantages of a multisite configuration, you should consider moving to a near stateless system with no affinity to a data center. You may, of course, decide to route customers based on proximity using a geo-locator service, but you will want the flexibility of determining when to route which traffic to which data center. In this configuration, where data is present at all three data centers and no state or session data is held solely within a single data center, a failure of a service or the failure of an entire data center allows the end user to almost seamlessly fail over to the next available data center. This results in the highest possible availability for any potential configuration.

In some cases, you might determine that maintaining state or data center affinity is required, or the cost of architecting out state and affinity is simply too high. If so, you’ll need to prepare for a service or data center failure by deciding whether to fail all transactions or sessions, force them to restart, or replicate the state and session to at least one more data center. This approach increases the costs slightly, because now you need to either build or buy a replication engine for the user state information and obtain additional systems or storage to handle it.

Multiple Live Site Considerations

Running multiple live sites offers the following benefits:

- Higher availability as compared to a hot-and-cold site configuration
- Lower costs compared to a hot-and-cold site configuration
- Faster customer response times if customers are routed to the closest data center for dynamic calls
- Greater flexibility in rolling out products in a SaaS environment
- Greater confidence in operations versus a hot-and-cold site configuration

Drawbacks or concerns associated with a multiple live site configuration include the following:

- Greater operational complexity
- Small increase in headcount needs
- Increase in travel and network costs
- Increase in operational complexity

Architectural considerations in moving to a multiple live site environment include these issues:

- Eliminate the need for state and affinity wherever possible
- Route customers to the closest data center if possible to reduce dynamic call times
- What must your organization own or lease (wholly owned data center or colocation facility) versus what can it rent from an IaaS provider?
- Investigate replication technologies for databases and state if necessary

Conclusion

This chapter discussed the unique constraints that data centers create for hyper-growth companies, data center location considerations, and the benefit of designing for and operating out of multiple live data centers. When considering options for where to locate data centers, you should consider areas that provide the lowest cost of power with high quality and availability of power. Another major location-based criterion is the geographic risk in any given area. Companies should ideally locate data centers in areas with low geographic risk, low cost of power, and high power efficiency for air-conditioning systems.

Data center growth and capacity need to be evaluated and planned out months or even years in advance based on whether you lease or purchase data center space and how much space you need. Finding yourself in a position in which you need to immediately enter into contracts and occupy space at worst puts your business at risk and at best reduces your negotiating leverage and causes you to pay more money for space. A failure to plan for data center space and power needs well in advance could hinder your organization's ability to grow.

Infrastructure as a Service ("public cloud") companies can help small, medium, and large companies with their data center and server capacity needs. For small companies, these solutions may limit the risk of exploration for a product that may not be adopted; the low capital outlay means it is easy to walk away from mistakes. For medium-size and large companies, these solutions can improve financial

performance by allowing the organization to rent that which it need not own—namely, the server capacity that it does not keep running around the clock.

When planning data centers, remember to apply the three magic Rules of Three. The first rule is that there are three drivers of cost: the cost of the server, the cost of power, and the cost of HVAC services. The second rule is to always plan for at least three servers for any service. The third rule is to plan for three or more live data centers. Remember to consider IaaS options in your planning.

Multiple active data centers may provide a number of advantages for some companies. Your organization may benefit from higher availability and lower overall costs relative to the typical hot-and-cold site disaster recovery configuration. They also give the company greater flexibility in product rollouts and greater negotiation leverage with leased space. Operational confidence in facilities increases as compared to the lack of faith most organizations have in a cold or warm disaster recovery facility. Finally, customers' perceived response times tend to go down for dynamic calls when they are routed to the closest data center.

Drawbacks of the multiple live data center configuration include increased operational complexity, increases in headcount and network costs, and an increase in travel cost. That said, our experience is that the benefits far outweigh the negative aspects of such a configuration.

When considering a multiple live data center configuration, you should attempt to eliminate state and affinity wherever possible. Affinity to a data center closest to the customer is preferred to reduce customer-perceived response times, but ideally you want the flexibility of seamlessly moving traffic. You will need to implement some method of replication for databases. Moreover, should you need to maintain state for any reason, you should consider using that technology for state replication.

Key Points

- Power is typically the constraining factor within most data centers today.
- Cost of power, quality and availability of power, geographic risk, an experienced labor pool, and cost and quality of network transit are all location-based considerations for data centers.
- Data center planning has a long time horizon. It needs to be done months and years in advance.
- An organization should leverage IaaS capacity when its product's adoption risk is high, or when it makes sense to rent capacity that it cannot keep busy through the course of a day or year.
- The three magic drivers of data center costs are servers, power, and HVAC.

- Three is the magic number for servers: Never plan for a service having fewer than three servers initially.
- Three is the magic number for data centers: Always attempt to design for at least three live sites.
- Multiple live sites offer higher availability, greater negotiating leverage, higher operational confidence, lower costs, and faster customer response times than traditional hot-and-cold disaster recovery configurations.
- Multiple live sites tend to increase operational complexity, costs associated with travel and network connectivity, and headcount needs.
- Attempt to eliminate affinity and state in a design that includes multiple live sites.

Chapter 33

Putting It All Together

The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable.

—Sun Tzu

This book began with a discussion of how scalability is a combination of *art* and *science*. The art aspect of scaling is seen in the dynamic and fluid interactions between products, organizations, and processes. The science of scalability is embodied within the method by which we measure our efforts and in the application of the scientific method. A particular company's approach to scaling must be crafted around the ecosystem fashioned by the intersection of the product technology, the uniqueness of the organization, and the maturity and capabilities of the existing processes. Because a one-size-fits-all implementation or answer does not exist, we have focused this book on providing skills and lessons regarding approaches.

But everything begins with people. You can't get better or even sustain your current level of excellence without the right team, the right leadership, the right management, and the right organizational structure. People are central to establishing and following processes as well as designing and implementing the technology. Having the right people in terms of skill set, motivation, and cultural fit is an essential building block. On top of this must be placed the right roles. Even the best people must be placed in the right job that appropriately utilizes their skills. Additionally, these roles must be organized in an appropriate organizational structure, and personnel must receive strong leadership and management to perform at an optimal level.

Although people develop and maintain the processes within an organization, the processes control how the individuals and teams behave. Processes are essential because they allow your teams to react quickly to crises, determine the root causes of failures, determine the capacity of systems, analyze scalability needs, implement scalability projects, and address many more fundamental needs for a scalable system. There is no single right answer when it comes to processes, but there are many wrong answers. Each and every process must be evaluated first for general fit within the organization

in terms of its rigor or repeatability, and then specifically for which steps are right for your particular team in terms of complexity. Too much process can stifle innovation and strangle shareholder value. Conversely, if you are missing the processes that allow you to learn from both your past mistakes and failures, your organization will very likely at least underachieve and potentially even fail as a company.

Last but not least is the technology that drives the business, either as the product itself or as the infrastructure supporting the product's development. Many methodologies must be understood and considered when implementing your scalable solution, such as splitting by multiple axes, caching, using asynchronous calls, developing a data management approach, and many others that we've covered in this book. The key is to develop expertise in these approaches so that you can use them appropriately when necessary. The right people, who understand these approaches, and the right processes, which insist on the evaluation of these approaches, are all put together to develop scalable technology.

In the beginning of this book, we introduced the concept of virtuous and vicious cycles (refer to Figure I.1 in the Introduction). The lack of attention to the people and processes can cause poor technical decisions that create what we term a *vicious cycle*. After you start down this path, teams are likely to ignore the people and process more because of the demands on energy and resources to fix the technology problems. The exact opposite, what we called the *virtuous cycle*, occurs when the right people and the right process feed each other and produce excellent, scalable technology, thereby freeing up resources to continue to improve the overall ecosystem within the organization. After you start down the path of the vicious cycle, it is difficult to stop—but with focused intensity you can do it. In Figure 33.1, we have depicted this concept of stopping the downward spiral and turning it back in the other direction.

What to Do Now?

The question that you might have now that you have made your way through all this information about people, processes, and technology is “What do I do now?” We have a simple four-step process that we recommend for putting all this information into practice. Note that this process is appropriate whether you are making technical, process, organizational, or even personal changes (as in changes in you). These steps are as follows:

1. Assess your situation.
2. Determine where you need or want to be.
3. Annotate the difference between the two.
4. Make a plan.

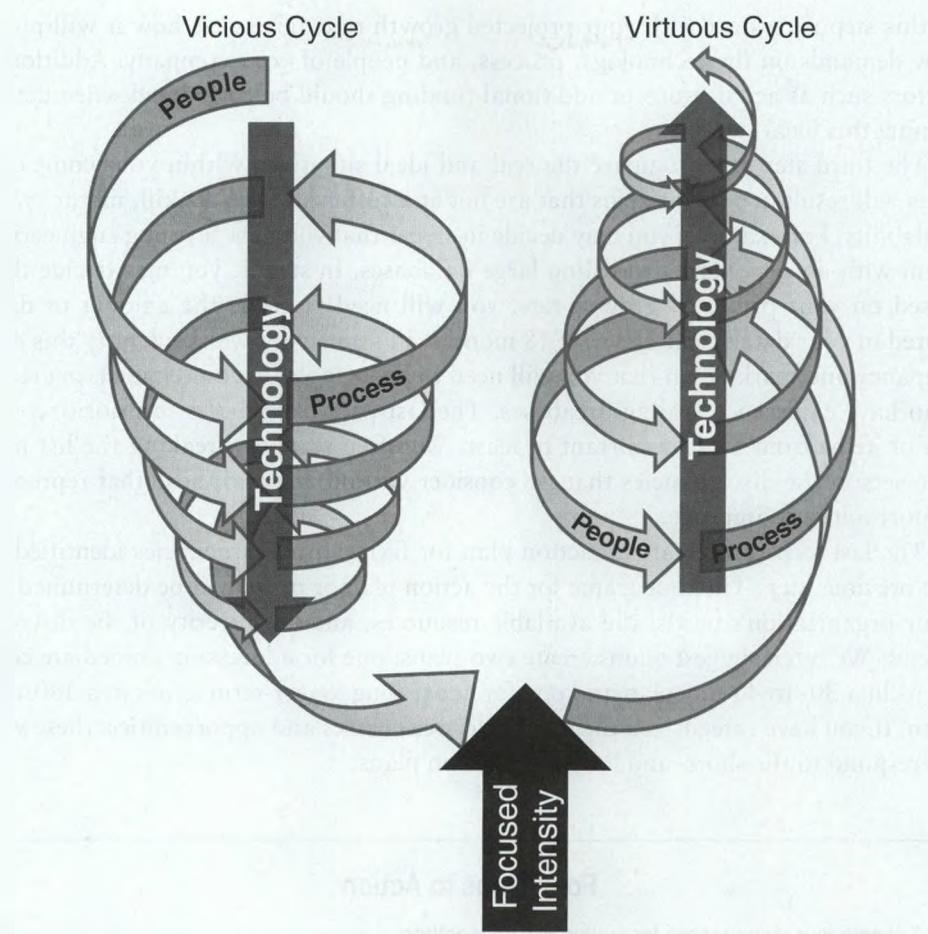


Figure 33.1 *Vicious and Virtuous Technology Cycles*

The first step is to assess your current situation. Look at all three aspects of people, process, and technology and make an honest assessment of where you are with each. Answer questions such as the following: What is the caliber of your people? Is the structure of your organization optimal? Which parts of your technology scale? Are your processes adequately supporting your goals? These assessments are difficult to do from inside the organization. Sometimes, you are just too close to the situation. Our firm, AKF Partners, is often asked to perform these assessments because we can provide an objective third-party perspective.

After you have the assessment in hand, the second step is to determine where you need to be along those three aspects of people, process, and technology. Do you need to focus on infusing more experience into the engineering team, or do you need to dedicate time and energy to developing more robust processes? One approach

to this step is to start with your projected growth rate and assess how it will place new demands on the technology, process, and people of your company. Additional factors such as acquisitions or additional funding should be considered when determining this ideal state.

The third step is to compare the real and ideal situations within your company. This will result in a list of items that are not at a sufficient level of skill, maturity, or scalability. For example, you may decide in step 1 that you have a young engineering team with no experience in scaling large databases. In step 2, you may decide that based on your projected growth rate, you will need to triple the amount of data stored in your database in the next 18 months. In step 3, you would identify this discrepancy and mark down that you will need to add people on your engineering team who have experience scaling databases. The last part of step 3 is to prioritize this list of items from most important to least. We often start by breaking the list into two sets: those discrepancies that we consider weaknesses and those that represent opportunities to improve.

The last step is to create an action plan for fixing the discrepancies identified in the previous step. The time frame for the action plan or plans will be determined by your organization's needs, the available resources, and the severity of the discrepancies. We often suggest teams create two plans: one for addressing immediate concerns in a 30- to 45-day plan and one for addressing longer-term issues in a 180-day plan. If you have categorized the items into weaknesses and opportunities, these will correspond to the short- and long-term action plans.

Four Steps to Action

A simple four-step process for putting this all together:

1. *Perform an assessment.* Rate your company in terms of its organization, processes, and architecture. Use an outside agent if necessary.
2. *Define where you need to be or your ideal state.* How large will your company grow in 12 to 24 months? What does that growth mean for your organization, processes, and architecture?
3. *List the differences between your actual and ideal situations.* Rank order these differences from most to least severe.
4. *Make an action plan.* Put together an action plan to resolve the issues identified in the previous steps. This can be in the form of a single prioritized plan, short-term and long-term plans, or any other planning increments that your company uses.

Further Resources on Scalability

We have covered a lot of material in this book. Because of space limitations, we have often been able to address some topics only in a summary fashion. Following are a few of the many resources that can be consulted for more information on concepts related to scalability. Not all of these necessarily share our viewpoints on many issues, but that does not make them or our positions any less valid. Healthy discussion and disagreement are the backbone of scientific advancement. Awareness of different views on topics will give you a greater knowledge of the concept and a more appropriate decision framework.

Blogs

- AKF Partners Blog: <http://www.akfpartners.com/techblog>
- Silicon Valley Product Group by Marty Cagan: <http://www.svpg.com/blog/files/svpg.xml>
- All Things Distributed by Werner Vogels: <http://www.allthingsdistributed.com>
- High Scalability Blog: <http://highscalability.com>
- Joel on Software by Joel Spolsky: <http://www.joelonsoftware.com>
- Signal vs. Noise by 37signals: <http://feeds.feedburner.com/37signals/beMH>
- Scalability.org: <http://scalability.org>

Books

- *Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications* by Cal Henderson
- *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services* by Neil J. Gunther
- *The Art of Capacity Planning: Scaling Web Resources* by John Allspaw
- *Scalability Rules: 50 Principles for Scaling Websites* by Marty Abbott and Michael Fisher (the authors of this book)
- *Scalable Internet Architectures* by Theo Schlossnagle
- *The Data Access Handbook: Achieving Optimal Database Application Performance and Scalability* by John Goodson and Robert A. Steward
- *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley Object Technology Series) by Bruce Powel Douglass
- *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide* (Addison-Wesley Information Technology Series) by David S. Linthicum
- *Inspired: How to Create Products Customers Love* by Marty Cagan

Part V

Appendices

Appendix A: Calculating Availability	539
Appendix B: Capacity Planning Calculations	547
Appendix C: Load and Performance Calculations	555

Det Kgl.
Bibliotek

Appendices

A

Appendix A: Classification of the Arabic Manuscripts

B

Appendix B: Classification of the Manuscripts

C

Appendix C: Classification of the Manuscripts

Appendix A

Calculating Availability

There are many ways of calculating a site's availability. Included in this appendix are five ways that this can be accomplished. In Chapter 6, Relationships, Mindset, and the Business Case, we made the argument that knowing your availability is extremely important to make the business case that you need to undertake scalability projects. Downtime equals lost revenue. Moreover, the more scalability projects you postpone or neglect to accomplish, the worse your outages and brownouts will be. If you agree with all of that, then why does it matter how you calculate outages or downtime or availability? It matters because the better job you do and the more everyone agrees that your method is the standard way of calculating the measurement, the more credibility your numbers have. You want to be the final authority on this measurement; your Agile teams need to own it and be the custodians of it. Imagine how the carpet could be pulled out from under your scalability projects if someone disputed your availability numbers in the executive staff meeting.

Another reason that a proper and auditable measurement should be put in place is that for Software as a Service (SaaS) offerings, there is no more important metric than being available to your customers when they need your service. Everyone in the organization should have this metric and goal as part of his or her personal goals. Every member of the technology organization should know the impact on availability that every outage causes. People should question one another about outages and work together to ensure they occur as infrequently as possible. With availability as part of the company's goals, affecting employees' bonuses, salaries, promotions, and so on, this should be a huge motivator to care about this metric.

Before we talk about the five different methods of calculating availability, we need to make sure we are all on the same page with the basic definition of availability. In our vernacular, *availability* is how often the site is available over a specific duration. It is simply the amount of time the site can be used by customers divided by the total time. For example, if we are measuring availability over one week, we have 10,080 minutes of possible availability, $7 \text{ days} \times 24 \text{ hr/day} \times 60 \text{ min/day}$. If our site is available 10,010 minutes during that week, our availability is $10,010 / 10,080 = 0.9935$. Availability is normally stated as a percentage, so our availability would be 99.35% for the week.

Hardware Uptime

The simplest and most straightforward measurement of availability is calculating it based on device (or hardware) uptime. Using simple monitoring tools that rely on SNMP traps for catching when devices are having issues, organizations can monitor the hardware infrastructure as well as keep track of when the site's hardware is having issues. On whatever time period availability is to be calculated, the team can look back through the monitoring log and identify how many servers had issues and for what duration.

A simple method would be to take the total time of the outage and multiply it by a percentage of the site that was impacted. The percentage would be generated by taking the number of servers having issues and dividing by the total number of servers hosting the site. As an example, let's assume an access switch failed and the hosts were not dual homed, so the failure took out 12 Web servers that were attached to it for 1½ hours until someone was able to get in the cage and swap the network device. The site is hosted on 120 Web servers. Therefore, the total downtime would be 9 minutes, calculated as follows:

$$\text{Outage duration} = 1\frac{1}{2} \text{ hours}$$

$$\text{Servers impacted} = 12$$

$$\text{Total servers} = 120$$

$$90 \text{ min} \times 12/120 = 9 \text{ min}$$

With the downtime figured, the availability can be calculated. Continuing our example, let's assume that we want to measure availability over a week and this was our only outage during that week. During a week, we have 10,080 minutes of possibly availability, $7 \text{ days} \times 24 \text{ hr/day} \times 60 \text{ min/hr}$. Because this is our only downtime of the week, we have $10,080 - 9 = 10,071$ minutes of uptime. Availability is simply the ratio of uptime to total time expressed as a percentage, so we have $10,071 / 10,080 = 99.91\%$.

As we mentioned, this is a very simplistic approach to availability. The performance of a Web server, of course, is not necessarily the experience that your customers have. Just because a server was unavailable does not mean that the site was unavailable for the customers; in fact, if you have architected your site properly, a single failure will likely not cause any customer-impacting issues. The best measure of availability will have a direct relation to the maximization of shareholder value; this maximization in turn likely considers the effects on customer experience and the resulting impacts on revenue or cost for the company.

This is not meant to imply that you should not measure your servers and other hardware's availability. You should, however, refer back to the goal tree in Chapter 5, Management 101, shown in Figure 5.2. Device or hardware availability would likely be a leaf

on this tree beneath the availability of the ad-serving systems and the registration systems. In other words, the device availability impacts the availability of these services, but the availability of the services themselves is the most important metric. Although you should use device or hardware availability as a key indicator of your system's health, you also need a more sophisticated and customer-centric measurement for availability.

Customer Complaints

The next approach to determining availability involves using the customers as a barometer or yardstick for your site's performance. This measurement might be in the form of the number of inbound calls or emails to your customer support center or the number of posts on your forums. Often, companies with very sophisticated customer support services will have real-time tracking metrics on support calls and emails. Call centers measure this data every day and keep tabs on how many calls and emails they receive as well as how many they can service. If there is a noticeable spike in such service requests, it is often the fault of an issue with the application.

How could we turn the number of calls into an availability measurement? There are many ways to create a formula for doing this, but they are all inaccurate. One simple formula might be to take the number of calls received on a normal day and the number received during a complete outage; these would serve as your 100% available and 0% available rates. As the number of calls increases beyond the normal day rate, you start subtracting availability until you reach the amount indicating a total site outage; at that point, you count the time as the site being completely unavailable.

As an example, suppose we normally get 200 calls per hour from customers. When the site is completely down in the middle of the day, the call volume goes to 1,000 per hour. Today, we start seeing the call volume go to 400 per hour at 9:00 a.m. and remain there until noon, when it drops to 150 per hour. We assume that the site had some issues during this time, a suspicion that is confirmed by the operations staff. We mark the period from 9:00 a.m. to noon as an outage. The percentage of downtime is 25%, calculated as follows:

$$\text{Outage duration} = 3 \text{ hours} = 180 \text{ min}$$

$$\text{Normal volume} = 200 \text{ calls/hr}$$

$$\text{Max volume} = 1,000 \text{ calls/hr}$$

$$\text{Diff (Max - Norm)} = 800 \text{ calls/hr}$$

$$\text{Amount of calls above normal} = 400 - 200 = 200 \text{ calls/hr}$$

$$\text{Percentage above normal} = 200 / 800 = 1 / 4 = 25\%$$

$$180 \text{ min} \times 25\% = 45 \text{ min}$$

Although this is certainly closer to a real user experience metric, it is also fraught with problems and inaccuracies. For starters, customers are not likely to call in right away. Most service centers require people to stay on the phone for several minutes or longer waiting before they are able to speak with a real person. Therefore, many customers will not bother calling in because they don't want to be put on hold. Not all customers will call; in fact, probably only your most vocal customers will call. While at eBay, for instance, we measured that the customer contact rate would be somewhere in the vicinity of 1% to 5% of the customers actually impacted. This fact skews the metrics toward the functionality that is used by your most vocal customers, who are often your most advanced users. Another major issue with this measurement is that a lot of Web 2.0 or SaaS companies do not have customer support centers. As a consequence, they have very little direct contact with their customers; in turn, the delay in understanding if there is a real problem, the significance of it, and the duration of it are extremely difficult to detect. Another issue with this measurement is that customer calls vary dramatically depending on the time of the day. To compensate for this factor, you must have a scale for each hour to compare against.

Similar to the hardware measurement discussed earlier, the measurement of customer contacts is a good metric to keep track of but not a good one to solely rely upon on to gauge your availability. The pulse of the customer or customer temperature—whatever you wish to call this factor—is a great way to judge how your customer base is responding to a new layout or feature set or payment model. This feedback is invaluable for the product managers to ensure they are focused on the customers' needs and listening to their feedback. For a true availability measurement, we again recommend something more sophisticated.

Portion of Site Down

A third way of measuring availability is monitoring the availability of services on your site. This is obviously more easily accomplished if your site has fault isolation lanes (swim lanes) created to keep services separated. Perhaps you might monitor the ability of a simulated user, usually in the form of a script, to perform certain tasks such as logon, run reports, and so on. This simulated user is then the measure of your availability. As an example, if you want to monitor five services—login, report, pay, post, and logout—you could create five scripts that run every five minutes. If any script fails, it notifies a distribution list. After the service is restored, the test script stops sending failure notices. This way, you can track through email exactly when the downtime occurred and which services were affected.

As an example, suppose we have this monitoring method set up for our five services. We receive problem emails for our login service starting at 9:45 a.m. and they

stop at 11:15 a.m. This gives us 1½ hours of downtime on one of our services. A simple method of calculating the availability is to take 1/5 of the downtime, because one of the five services had the problem. This would result in 18 minutes of downtime, calculated as follows:

$$\text{Outage duration} = 1\frac{1}{2} \text{ hours}$$

$$\text{Services impacted} = 1$$

$$\text{Total services} = 5$$

$$90 \text{ min} \times 1/5 = 18 \text{ min}$$

This method does have some limitations and downsides, but it can be a fairly accurate way of measuring the impact of downtime upon customers. One of its major limitations is that this method monitors only services for which you have scripts built. If you either don't build the scripts or can't accurately simulate real users, your monitoring will be less effective. Obviously, you need to monitor the most important services that you provide in your application. It's likely not realistic to monitor every single service, but the major ones should absolutely be covered.

Another limitation is that not all users use all the services equally. For example, a signup flow is used only by new users, whereas a login flow is used by all of your existing customers. Should each flow be weighted equally? Perhaps you could add weighting by importance or volume of usage to each flow to help more accurately calculate the impact on your customers for the availability of each flow.

A final limitation of this method is that if you monitor your application from inside of your network, you are not necessarily experiencing the same customer impact as outside your network. This is especially true if the outage is caused by your Internet service provider (ISP).

Even though this approach of monitoring the availability of services on your site does have some limitations, it does offer a pretty good customer-centric availability measurement.

Third-Party Monitoring Service

The fourth measurement that we want to present for determining availability is using a third-party monitoring service. This is very similar to the previous measurement except that it overcomes the limitation of monitoring within your own network and potentially includes more sophisticated scripting to achieve a more realistic user experience. The principal concepts are very similar: You set up services that you want to monitor and have the third-party service alert a distribution list when a problem occurs.

A wide variety of vendors offer such monitoring services, including Keynote, Gomez, and Montastic, among many others. Some of these services are free and others are fairly costly depending on the sophistication and diversity of monitoring that you require. For example, some of the premium monitoring services have the capability of monitoring from many different peering networks as well as providing user simulation from users' computers, which is nearly as realistic a user experience as you can achieve.

The key with using a third-party monitoring service is to first determine your requirements for monitoring. Issues to consider include how difficult your application or services are to monitor because of their dynamic nature and how many different geographic locations your site is monitored from. Some services are capable of monitoring from almost any Internet peering service globally. Some vendors offer special monitoring for dynamic pages. Others offer dynamic alerts that don't need thresholds set but instead "learn" what is normal behavior for your application's pages and alert when they are "out of control," statistically speaking.

Business Graph

The last measurement that we will present was provided as an example in Chapter 6. The business graph is our preferred method of availability calculation because it couches availability in terms of something meaningful to the business—revenue. It entails using key performance indicators (like real-time revenue) to determine the impact of availability on the business and its stakeholders. To accomplish this, you must implement monitors for each of the key performance indicators (KPIs) that you believe are closely related to business performance. Then, each time there is an outage, you can compare a normal day with the outage day and determine the degree of impact. The way to do this is determine the area between the graphs, which is representative of the amount of downtime that should be registered.

In Figure A.1, the solid line is a normal day's revenue (here revenue is the KPI being monitored) and the dashed line is the traffic from the day with an outage. The outage began at 4:00 p.m. and lasted until approximately 6:40 p.m., when the site was fully recovered. The area between the lines for this interval is calculated as the outage percentage and could be used in the calculation of downtime.

As you can see, measuring availability is not straightforward and can be very complex. The purpose of these examples is not to say which method is right or wrong, but rather to give you several options that you can choose from or combine to develop the best overall availability metric for your organization. The importance of a reliable and agreed-upon availability metric should not be understated, as it

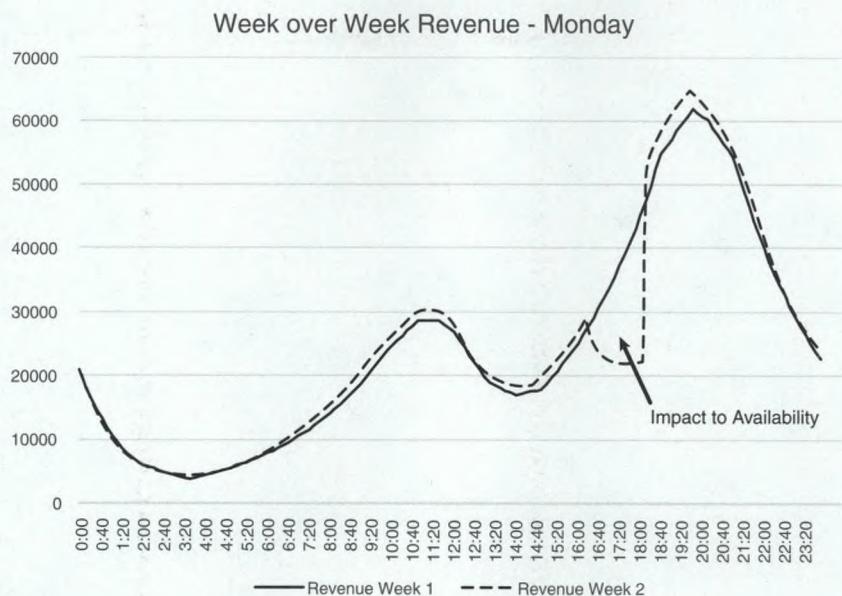


Figure A.1 Business Availability Graph

will be the basis for many recommendations and scalability projects as well as a metric that should be tied directly to people's goals. Spend the time necessary to come up with the most accurate metric possible that will become the authoritative measurement of availability.

Appendix B

Capacity Planning Calculations

In Chapter 11, Determining Headroom for Applications, we covered how to determine the headroom or free capacity that was available for your application. In this appendix, we will walk through a larger example of capacity planning for an entire site, but we will follow the process outlined in Chapter 11. The steps to be followed are:

1. Identify the components.
2. Determine the actual and maximum usage rates.
3. Determine the growth rate.
4. Determine seasonality.
5. Compute the amount of headroom gained through projects.
6. State the ideal usage percentage.
7. Perform the calculations.

For an example, let's consider a typical business-to-business (B2B) Software as a Service (SaaS) product that is becoming very popular and growing rapidly. The growth is seen in bursts; as new companies sign up for the service, the load increases based on the number of users at each new company. So far, 25 client companies with a total of 1,500 users have accounts on the SaaS product offering. The CTO of the SaaS offering needs to perform a capacity planning exercise because she is planning for next year's budget and wants accurate cost projections.

Step 1 is to identify the components within the application that we care about sufficiently to include in the analysis. The SaaS application is very straightforward, with a Web server tier, an application server tier, and a single database with standbys for failover. We will skip the network devices in this capacity planning exercise, but periodically they should be reanalyzed to ensure that they have enough headroom to continue to scale.

Step 2 is to determine the actual and maximum usage rates for each component. Let's assume we have good records or logs of these items and we know the actual peak and average usage for all our components. We also perform load and

performance testing before each new code release, and we know the maximum requests per second for each component based on the latest code version.

In Figure B.1, the Web server and application server requests are being tracked and monitored. You can see that there are around 125 requests per second at peak for the Web servers. There are also around 80 requests per second at peak for the application servers. The difference occurs because there are a lot of preprocessed static pages on the Web servers that do not require any business logic computations to be performed. These pages include corporate pages, landing pages, images, and so on. You could make an argument that different types of pages scale differently and should be put on a different set of Web servers or at a minimum be analyzed differently for capacity planning. For simplicity of this example, we will continue to group them together as a total number of requests.

From the graphs, we put together a summary in Table B.1 of the Web servers, application servers, and the database server. For each component, we have the peak total requests, the number of hosts in the pool, the peak request per host, and the maximum allowed on each host. The maximum allowed was determined through load and performance testing with the latest code base and is the number at which we begin to see diminished response times that are outside of our internal service level agreements.

Step 3 is to determine the growth rate. To determine this, we turn to our traffic usage, which we monitor to show how much traffic the SaaS offering has each day. This data is displayed in Figure B.2. We also use this graph to show a percentage growth rate on

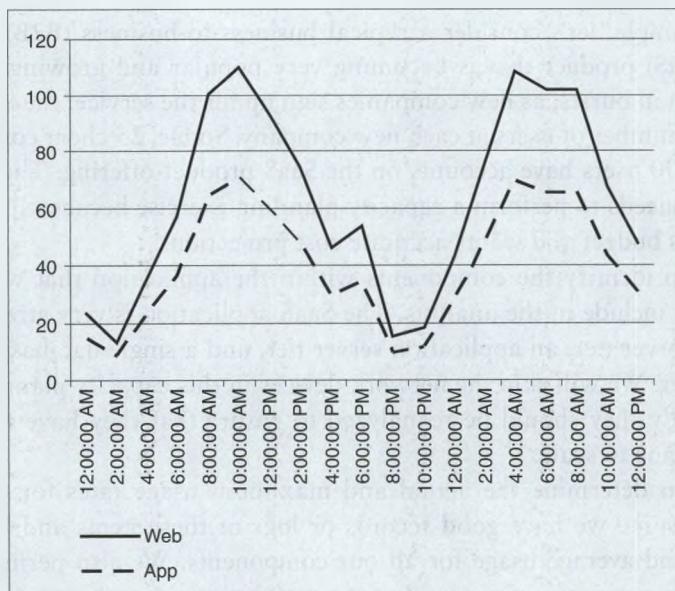


Figure B.1 Web Server and Application Server Requests

Table B.1 Web Server and Application Server Requests Summary

(a) Web Server

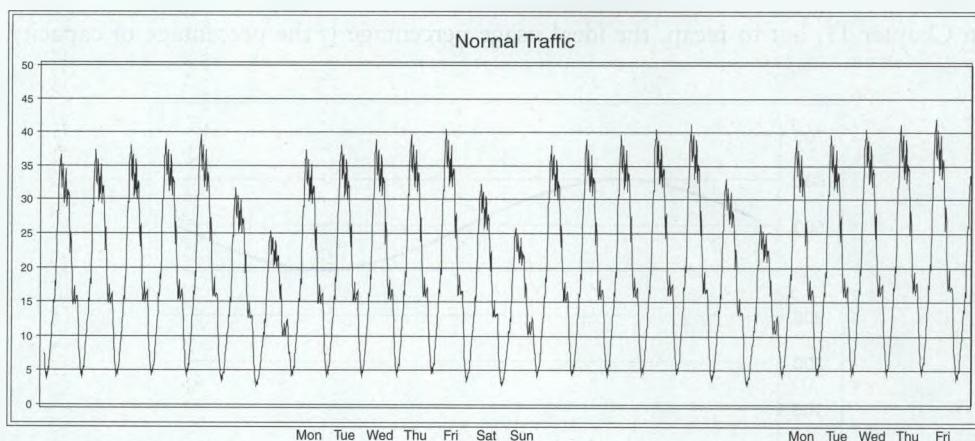
Peak requests per second total	125
Number of servers	5
Peak requests per second per server	25
Maximum requests per second per server	75

(b) Application Server

Peak requests per second total	80
Number of servers	4
Peak requests per second per server	20
Maximum requests per second per server	50

(c) Database

Peak SQL per second total	35
Number of nodes	1
Peak requests per second per server	35
Maximum SQL per second per node	65

**Figure B.2** Normal Traffic Graph

a week-over-week basis. For our traffic, we have a 2% week-over-week growth rate on average throughout the year. This equates to a 280% growth rate annually, or approximately 3x growth in traffic each year. This combined growth comes from existing clients using the application more (i.e., natural growth) as well as the growth caused by our sales team signing up new clients (i.e., human-made growth). These growth rates are sometimes calculated separately if the sales team can provide accurate estimates for the number of clients who will be brought on board each year, or they can be extrapolated from previous year growth rates for new and existing customers.

Step 4 is to determine the seasonality effect on the SaaS offering. Because of the nature of B2B product offerings, a lot of tasks are accomplished during the early part of the year, as this is when budgets are renewed. Figure B.3 shows the seasonality graph of our example SaaS offering for existing customers but excluding new users. This way, we can eliminate the growth from new users and just see the seasonality effect on existing users. This capacity planning exercise is conducted in August, which is typically a lower-use month; we expect to see a 50% increase in traffic by January based on our seasonality effect.

Step 5 is to compute the headroom that we expect to gain through scalability or headroom projects. Perhaps one project that has been planned for the fall of this year is to split the database by creating a write master with two read copies—an x-axis split according to the AKF Database Scale Cube. This split increases the number of nodes to three and, therefore, distributes the existing requests among the three nodes. The write requests are more CPU intensive, but that factor is offset by the smaller number of write requests as compared to the number of read requests. For our capacity planning purposes, this will effectively drop the number of requests per node from 35 to 12.

Step 6 is to state the ideal usage percentage. We covered this topic in great detail in Chapter 11, but to recap, the ideal usage percentage is the percentage of capacity

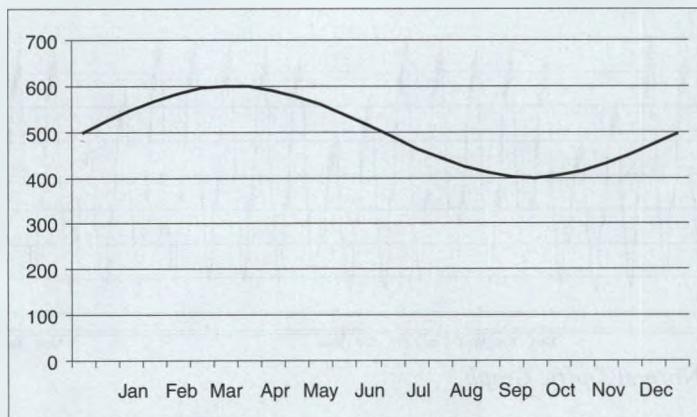


Figure B.3 Seasonality Traffic Graph

Headroom = (Ideal Usage Percentage × Maximum Capacity) – Current Usage

$$-\sum_{t=1}^{12} (\text{Growth}(t) - \text{Optimization Projects}(t))$$

Figure B.4 Headroom Equation

that you feel comfortable using on a particular component. The reasons for not using 100% are twofold. First, our data and calculations inevitably contain some degree of error. As hard as we try and as much data as we collect, there are sure to be inaccuracies in our capacity plans. For this reason, we need a buffer to accommodate the inaccuracies. Second, as you approach using 100% capacity of any hardware, the system's behavior may become erratic and unpredictable. We want to operate in the range of perfectly predictable behavior. For each component, there may be a different ideal usage percentage depending on how accurate you believe your estimates are and how much buffer you need. For our exercise, we will use 90% for the database and 75% for the Web and application servers.

Step 7, our final step, is to perform the calculations. You may recall the formula shown in Figure B.4. This formula states that the capacity or headroom of a particular component is equal to the ideal usage percentage multiplied by the maximum capacity of that component minus the current usage minus the sum over the time period of the growth rate, which also includes seasonality, minus projected gains from scalability projects.

As you can see in Table B.2, we have calculated the headroom or capacity for each component. Let's walk through the Web tier in detail.

Table B.2 Capacity Calculation

(a) Web Server

Peak requests per second total	125
Number of servers	5
Peak requests per second per server	25
Maximum requests per second per server	75
Ideal usage percentage	75%
Scalability projects	0
Growth rate over 12 months	437.5
Headroom = (IUP × max) – usage – sum (growth – projects)	-281.25

(Continues)

Table B.2 Capacity Calculation (Continued)

(b) App Server	
Peak requests per second total	80
Number of servers	4
Peak requests per second per server	20
Maximum requests per second per server	50
Ideal usage percentage	75%
Scalability projects	0
Growth rate over 12 months	280
Headroom = (IUP × max) – usage – sum (growth – projects)	-210
(c) Database	
Peak SQL per second total	35
Number of nodes	3
Peak requests per second per server	11.67
Maximum SQL per second per node	65
Ideal usage percentage	90%
Scalability projects	0
Growth rate over 12 months	122.5
Headroom = (IUP × max) – usage – sum (growth – projects)	18

The calculation starts with the ideal usage percentage multiplied by the maximum usage; for the Web server tier, this is $75\% \times 75$ requests per second per server $\times 5$ servers, which equals 281.25 requests per second (req/s). This is the total capacity that our example SaaS offering can handle. The next part of the calculation is to subtract the current usage of 125 req/s, which yields 156.25 req/s. Thus we currently have 156 req/s extra capacity. Now we need to add in the future needs. Sum the growth rate minus any projects planned over the time period to improve scalability. For the Web servers, we have a 3x growth rate annually in traffic and a 50% growth rate from seasonality, resulting in a 3.5x total current usage, which equals 437.5 req/s. This final equation is

$$(75\% \times 75 \times 5) - 125 - (3.5 \times 125) = -281.25 \text{ req/s}$$

Because this is a negative number, we know that we do not have enough current capacity based on the number of servers in the pool. We can take several actions to correct this deficiency. We could add capacity projects over the next year that would increase our capacity on the existing servers to handle the growing traffic. We could also purchase more servers and grow our capacity through a continued *x*-axis split. The way to calculate how many servers you need is to simply divide the capacity number, 281.25, by the sum of the ideal usage percentage and the maximum usage per server ($75\% \times 75$) = 56.25. The result of this is $281.25 / 56.25 = 5$, which means you need five additional servers to handle the expected traffic growth.

Appendix C

Load and Performance Calculations

In Chapter 17, Performance and Stress Testing, we covered the definition, purpose, variations on themes, and the steps to complete performance testing. We discussed that performance testing covers a broad range of testing evaluations, with each sharing the focus on the necessary characteristics of the system rather than the individual materials, hardware, or code. Focusing on ensuring the software meets or exceeds the specified requirements or service level agreements is what performance testing is all about. We emphasized that an important aspect of performance testing is using a methodical approach; from the very beginning, we argued for establishing benchmarks and success criteria; and at the very end, we suggested repeating the tests as often as possible for validation purposes. We believe that such a structured and repetitive approach is critical to achieve results in which you can be confident and upon which you can base sound decisions. Here, we want to present an example performance test to demonstrate how the tests are defined and analyzed.

Before we begin the example, we want to review the seven steps that we presented for a performance test in Chapter 17:

1. *Establish success criteria.* Establish which criteria or benchmarks are expected from the application, component, or system that is being tested.
2. *Establish the appropriate environment.* The testing environment should be as close to the production environment as possible to ensure that your results are accurate.
3. *Define the tests.* There are many different categories of tests that you should consider for inclusion in the performance test, including endurance, load, most used, most visible, and component tests.
4. *Execute the tests.* Perform the actual tests and gather all data possible.
5. *Analyze the data.* Analyze the data by such methods as comparing it to previous releases and stochastic models to identify which factors are causing variations.

6. *Report to engineers.* Provide the analysis results to the engineers for them to either take action or confirm that the results are as expected.
7. *Repeat the tests and analysis.* As necessary and possible, validate bug fixes and continue testing.

Let's use a business-to-business (B2B) Software as a Service (SaaS) offering as an example. Our made-up SaaS company has a new release of its code in development. This code base, known internally as release 3.1, is expected out early next month. The engineers have just completed development of the final features, and the code base is now in the quality assurance testing stage. We are joining just as it is beginning the performance testing phase, which in this company occurs during the later phases of quality assurance after the functional testing has been completed.

The first step that we need to accomplish is to identify the benchmarks for the performance test. If the company has performance tested all of its releases for the past year, it will have a good set of benchmarks against which comparisons can be made. In the past, it has used the criterion that the new release must be within 5% of the previous release's performance. This specification ensures that the service has sufficient hardware in production to run the application without scaling issues as well as helps control the cost of hardware for new features. The team confirms with its management that the criteria will remain the same for this release.

The second step is to establish the environment. The team does not have a dedicated performance testing environment and must requisition the use of a shared staging environment from the operations team when it needs to perform testing. The team is required to give one week's notice to schedule the environment and is given three days in which to perform the tests. Occasionally, if the environment is being heavily used for multiple purposes, the testing is required to be performed during off hours, but this time the team has from Monday at 9:00 a.m. through Wednesday at midnight to perform the tests. The staging environment that is used is a scaled-down version of production; all of the physical and logical hosts are present, but they are much smaller in size and total numbers than in the production environment. For each component or service in production, there is a single server that represents a pool of servers in production. Although such a structure is not ideal and we would prefer to have two servers in a test environment representing a larger pool in production, the configuration is what our example SaaS company can afford today, and it's certainly better than nothing.

The third step is to define the tests. For the SaaS product offering, the most important performance test to conduct is a load test on the most critical components—perhaps a report page and an upload page for getting data into the system. If the team completes the tests for these critical services and has extra capacity, it will often perform a load test on the landing page because it is the most visible component in the application.

The fourth step is to execute or conduct the tests and gather the data. Our fictional SaaS company has automated scripts that run the tests and capture the data simultaneously. These scripts run a fixed number of simultaneous executions of a standard data set or set of instructions. This amount has been determined as the maximum amount of simultaneous executions that a particular service will need to be able to handle on a single server. For the upload, the number is 10 simultaneous uploads with data sets ranging from 5,000 to 10,000 pieces of data. For the reports, the number is 20 simultaneous requests per report server. The scripts capture the mean or average response time, the standard deviation of the response times, the number of SQL executions, and the number of errors reported by the application.

Table C.1 shows the response time results of the upload tests. The results are from 10 separate runs of the test, each with 10 simultaneous executions of the upload service and their corresponding response times.

Table C.2 gives the response times for the report tests. The execution time results are from 10 separate runs of the test, each with 20 simultaneous executions of the report. You can see that the testing scripts provided means and standard deviations for each run as well as for the overall data.

Table C.1 Upload Test Response Time Results

1	2	3	4	5	6	7	8	9	10	Overall	
8.4	2.2	5.7	5.7	9.7	9.2	5.4	7.9	6.1	5.8		
6.4	7.9	4.4	8.6	10.6	10.4	3.9	8.3	7.3	9.3		
2.8	10	3	8.5	2	10.8	9.4	2.4	7.1	10.8		
8.8	5.9	10.2	2.3	10.5	2.6	6	7.1	10.4	8.2		
9	3.4	7.7	4	4.8	2.7	6.8	7.5	4.5	2.6		
10.4	3.7	2	7.4	7.5	2.4	9	9.7	5	2.5		
5.8	9.6	7.9	4.8	8.8	7.9	4.1	2.5	8	8.1		
6.5	6.2	6.5	9.5	2.4	2.4	10.6	6.6	2.2	5.7		
6.5	6.2	6.5	9.5	2.4	2.4	10.6	6.6	2.2	5.7		
5.7	4.1	8.2	7.3	9.7	4.8	3.3	9.1	2.8	7.9		
Mean	7.0	5.9	6.2	6.8	6.8	5.6	6.9	6.8	5.6	6.7	6.4
StDev	2.2	2.6	2.5	2.5	3.6	3.6	2.8	2.5	2.7	2.7	2.7

Table C.2 Report Test Response Time Results

	1	2	3	4	5	6	7	8	9	10	Overall
	4.2	5.2	3.2	6.9	5.3	3.6	3.2	3.3	2.4	4.7	
	4.4	6.5	1.1	6.7	3.1	4.8	4.6	1.4	2	6.5	
	1.4	3	6.5	2.7	6.2	5.4	1.3	3.7	1.8	2.6	
	3.8	6.9	2.7	2.6	5.8	6.8	1	3.5	1.8	4.9	
	2	6.7	2	4.9	4.1	6	2.3	3.9	6.7	1.3	
	1.3	4.3	2.7	1.4	3.3	3.7	1.7	3.7	6.2	3.9	
	4	6.5	1.4	3.8	5.2	6	5.3	5.5	5.8	5.9	
	6.3	5.7	5.7	6.3	2	7	4.6	1.9	2.9	5.1	
	4.1	6.5	1.2	3.2	4.4	3.6	7	2.5	8.4	4.5	
	1.3	3.9	3.6	4.3	6.5	4.4	3.2	5.1	7.1	7.4	
	9	4.9	2.4	1.8	8.7	7.5	7.8	6.2	7	2.8	
	1.5	3.7	5.5	4	6.8	8.4	2.1	8.3	1.4	8.9	
	4.9	5.2	6.6	6.8	4.6	6.7	1.2	5.4	8	9	
	2.5	5.3	8.3	2.6	8.1	7.7	2	1.9	5.9	2.2	
	7.8	4.8	6	6.4	4.2	8.5	4.5	6.8	7.5	6.5	
	7.8	3.6	8.8	2.9	8.6	3.8	2.6	4.8	5.6	4.1	
	6.7	1.8	2.6	5.5	3.4	8.9	5.2	7.2	6.5	1.5	
	4.3	7.1	3.4	4.1	3.8	2	1.5	5	7.5	2.3	
	4.3	7.1	3.4	4.1	3.8	2	1.5	5	7.5	2.3	
	4.5	1.9	4	2.8	4.3	6.8	4.4	2.9	6.2	3.2	
Mean	4.3	5.0	4.1	4.2	5.1	5.7	3.4	4.4	5.4	4.5	4.6
StDev	2.3	1.7	2.3	1.7	1.9	2.1	2.0	1.9	2.4	2.3	2.1

Table C.3 provides the summary results of the upload and report tests for both the current version 3.1 as well as the previous version of the code base 3.0. For the rest of the example, we will stick with these two tests in order to cover them in sufficient detail and not have to continue repeating the same thing about all the other tests that could be performed. The summary results include the overall mean and standard deviation, the number of SQL queries that were executed for each test, and the number of

Table C.3 Summary Test Results

	Upload Test			Report Test		
	v 3.1	v 3.0	Diff	v 3.1	v 3.0	Diff
Simultaneous Executions	10	10		20	20	
Mean Response Time	6.4	6.2	3%	4.6	4.5	3%
Standard Dev Response Time	2.7	2.4	14%	2.1	2.4	-11%
SQL Executions	72	69	4%	240	220	9%
Number of Errors	14	15	-7%	3	2	50%

errors that the application reported. The third column for each test is the difference between the old and new versions' performance. This is where the analysis begins.

The fifth step is to perform the analysis on the data gathered during testing. As we mentioned, the third column in Table C.3 shows the difference between the versions of code for the upload and report tests. Our SaaS company's analysis of the data starts with this comparison.

As you can see, both of the tests had a 3% increase in the mean response time. Our stated guidelines were that no more than a 5% increase was allowed in a new code version. This performance is therefore acceptable.

The standard deviation, which measures the variability within the data, thereby showing how large or small the spread of individual data points is from the mean, indicates that variability increased on the upload test but decreased on the report test. Thus there is more variability in the upload than there was previously, which is something that we should probably investigate further.

The number of SQL query executions increased on both tests. This information should be addressed with the engineers to ensure that they did, indeed, add database queries to both services. If they did not, this result should be investigated to determine why the count increased.

The number of application reported errors has declined on the upload, which is a positive trend indicating that the engineers might have added application logic to handle different data sets. On the report, the number has increased significantly from a percentage standpoint, but the actual number is low, increasing from 2 to 3, so this is likely not a problem. Nevertheless, it should be pointed out to the engineers and possibly product managers to determine the severity of an error on this report.

To continue the analysis of the variability of the upload data as seen in the increased standard deviation, we can perform some simple statistical tests. One of the first tests that can be performed is a paired *t*-test, which is a hypothesis test for the mean difference between paired observations. The paired *t*-test calculates the

difference for each pair of measurements; for the upload test, it is each test of the data sets. Next, the paired *t*-test determines the mean of these weight changes and determines whether this is statistically significant. We won't go into the details of how to perform this analysis other than to say that you formulate a hypothesis and a null hypothesis. The hypothesis would be that the means of the data sets (v3.1 to v3.0) are the same, and the null or alternative hypothesis would be that they are not the same. The result is that these two data sets are statistically different.

Continuing the investigation of the increased variability in v3.1, the SaaS company might subject the data to a control chart. A control chart is a statistical test that measures for special-cause variation. Two types of variation exist: common cause and special cause. Common-cause variation, as the name implies, is normal variation in a process that exists because of common causes, such as the "noise" inherent in any process. Special-cause variation is caused by noncommon causes, such as differences in hardware or software. If a process has only common-cause variation, it is considered to be "in control." When special-cause variation is present, the process is "out of control."

To create a control chart, plot the data points on a graph and mark the mean as well as an upper control limit (UCL) and a lower control limit (LCL). The UCL is determined by calculating three standard deviations above the mean. The LCL is determined by calculating three standard deviations below the mean. Many different tests can be performed to look for special-cause variation. The most basic test is to look for any point that is above the UCL or below the LCL—in other words, look for any point that is more or less than three standard deviations from the mean. In Figure C.1, the version 3.1 upload response time is plotted in a control chart. As you

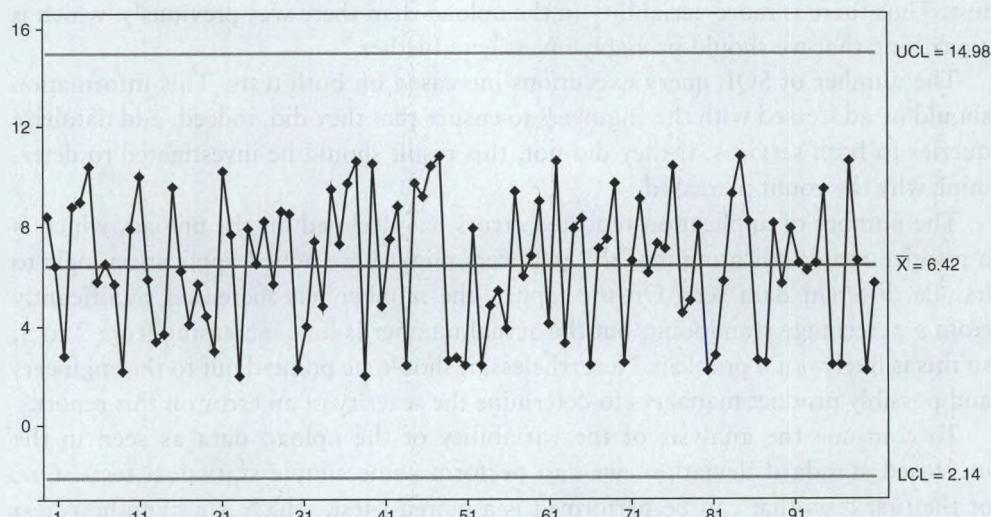


Figure C.1 Upload Response Time Control Chart

can see, no point is more than three standard deviations away. There are many other tests to determine how the new version of code should perform compared to the old version, but at this point, the company in the example decides to continue to the next step by getting the engineers involved.

The sixth step is to report the results of the tests and analysis. Perhaps the Agile teams responsible for the upload and report services decide that the increased variation is not concerning, but the increase in the number of database queries might be an issue, so they decide to file a bug report and investigate further.

The seventh step is to repeat the tests and analysis as necessary. Over the next few weeks, the SaaS company's Agile teams investigate and test the new code repeatedly. They end up running another set of performance tests on version 3.1 of the code before releasing the code to production. Because of their diligence, they feel confident that this version code will perform in a manner that is acceptable.

This example has covered the steps of performance testing. We have shown how you might decide which tests to perform, how to gather the data, and how to perform some basic analysis on it. A wide spectrum of sophistication can be demonstrated with performance testing. Much of that depends on the resources, in the form of both people and time, that your organization can commit to this step in the software development life cycle.

Index

A

- About this book, 5, 9–10
- “Above the Clouds” (UC Berkeley), 475–476
- Abrams, Jonathan, 71
- ACA (Affordable Care Act), 331–332
- Accountability of CEO, 24–25
- ACID database properties, 400, 401, 441
- Acronyms
 - DRIER, 148–149, 157, 158
 - RASCI, 36–39, 40, 156, 214, 223
 - SMART goals, 89–90, 96, 97, 156
- AdSense, 91
- AdSonar, 91, 92
- Affective conflict
 - about, 54–55
 - defined, 13
 - influencing innovation, 63
- Affordable Care Act (ACA), 331–332
- Agile Organizations
 - about, 66–68, 70, 240–241, 247
 - aligning to architecture, 62
 - ARB process in, 246–247
 - autonomy of teams in, 241–242, 247
 - evolution of, 59–61
 - illustrated, 61
 - incorporating barrier conditions in, 293–295
 - limited resources in, 242–243, 247
 - maintaining standards across teams, 243–246, 248
 - scaling processes for, 16
 - team ownership of architecture, 241–242
 - tradeoffs in, 307–308, 312, 313
- AKF
 - architectural principles of, 214–222
 - definition of management, 101
 - 5–95 Rule for, 104, 105, 117
 - risk model, 260

- AKF Application Scale Cube
 - implementing, 357–358
 - summary of, 365–367
 - using, 367–371
 - x-axis, 357–358, 359, 361, 365–367, 371–372
 - y-axis, 357–358, 361–362, 371, 372–373
 - z-axis, 357–358, 359, 361–362, 363–364, 365–367, 371–373
- AKF Database Scale Cube
 - applying, 375–376
 - business-to-business SaaS solutions with, 391–392
 - concerns about replication delays, 377
 - ecommerce implementation for, 388–389
 - employing for search implementation, 389–391
 - illustrated, 376
 - summary of, 385–388
 - timeline for employing splits, 393, 394
 - when and how to use, 392
 - x-axis, 376–381, 386, 387, 393
 - y-axis, 381–383, 386, 387, 388, 393–394
 - z-axis, 383–386, 387, 388, 394
- AKF Scale Cube. *See also* AKF Application Scale Cube; AKF Database Scale Cube
 - application state and, 351
 - with cloud environments, 485–487, 494
 - data segmentation scenarios for, 434
 - defined, 343–345
 - illustrated, 344, 350
 - implementing, 357–358
 - overview, 353–355
 - summary of, 350–352
 - uses for, 355
 - using for database splits, 375–376

- AKF Scale Cube (*continued*)
 when and where to use, 352–353
x-axis of, 344–346, 351, 352, 354
y-axis of, 346–348, 351, 352, 354
z-axis of, 349–350, 351, 352, 354
- Allchin, Jim, 43
- Allen, Paul, 256
- Allspaw, John, 159–160, 239
- Amazon, 16–17, 44, 123, 126, 281, 461, 462
- Amazon Web Services, 331, 488, 492, 516
- Amdahl's law, 448–449
- Amelio, Gil, 11, 257
- Apple Computer, 11, 257–258
- Application caches, 401
- Application servers
 capacity planning calculations for, 552
 monitoring requests for, 548–549
- Application service providers (ASPs), 461
- Applications. *See also* Preparing cloud applications; Splitting applications
 AKF Scale Cube and, 351
 cache hits/misses for, 397, 409
 caching software, 406–407
 calculating server capacities for, 552
 designing for monitoring, 495, 507
 grid computing and monolithic, 452–453, 458
 portability between clouds, 472, 474
 rapid development of, 296–297
 stateful and stateless, 218–219, 418–419
 user sessions for stateful, 420–422, 424
y-axis splits for complexity and growth in, 371
- Arao, Karl, 206
- ARB (Architecture Review Board)
 as barrier condition, 292, 294, 296, 302
 checklist for, 236
 considering members of, 230–232
 defined, 225
 entry/exit criteria for, 234–235
 feature approval by, 230
 following TAD rules, 324
 implementing JAD and, 237
 meetings of, 232–234
 overview, 237, 238
 process in Agile Organizations, 246–247
 using AKF Scale Cube, 353
- Architects, 30–31
- Architectural principles. *See also* Technology-agnostic design
 asynchronous design, 217–218, 222
 automation over people, 221–222
 building small, releasing small, failing fast, 221, 222
 buy when non-core, 220, 222
 D-I-D matrix, 306–307
 design for disabling systems, 215, 222, 300–301, 302
 designing for rollback, 215, 222, 302
 developing, 209–211, 223
 engendering ownership of, 213–214
 following RASCI principle, 214, 223
 illustrated, 213
 isolating faults, 221, 222
 mature technologies, 217, 222
 monitoring as, 215–216, 222
 most adopted AKF, 214–222
 multiple live sites, 216, 222
 N + 1 design, 213–215, 222
 providing two axes of scale, 219–220, 222
 scaling out, not up, 219, 222
 scope of, 223
 selecting, 212–213
 SMART characteristics of, 211, 223
 stateless systems, 218–219, 222
 team development of, 223
 using commodity hardware, 220, 222
- Architecture. *See also* Fault isolation
 aligning Agile Organizations to, 62
 designing for any technology, 317–318
 fault isolation terminology, 327–329
 implementing fault isolation, 339–341
 multiple live data centers, 527
 object cache, 401
 OSI Model, 460
 ownership by Agile teams, 241–242
 technology-agnostic, 318–319, 323–325
- Architecture Review Board. *See* ARB
- Art of Capacity Planning, The* (Allspaw), 159
- Art of scalability, 4, 531–532
- Artificial intelligence, 461
- ASPs (application service providers), 461

- Asynchronous design
 as architectural principle, 217–218, 222
 coordination and communication in, 415
 data synching methods for, 411–412, 423
 HTTP's stateless protocol, 418, 424
 initiating new threads, 413, 423
 scaling synchronously or
 asynchronously, 414–415
 synchronous vs. asynchronous calls,
 412–413
 systems using, 416–418
- ATM (Asynchronous Transfer Mode)
 networks, 460
- Atomicity of databases, 401
- Automating processes, 221–222
- Autonomic Computing Manifesto (IBM),
 461, 482
- Availability
 calculating Web site, 539
 customer complaints as metric for,
 541–542
 determining with third-party monitoring
 services, 543–544
 fault isolation and, 329–333, 334, 342
 graphing calculations of, 544–545
 guaranteeing transaction, 378
 increasing with fault isolation, 334, 342
 measuring, 112
 monitoring portion of site down,
 542–543
 TAD and, 323, 326
- Avoidance in user sessions, 420–421, 422
- Axes of scale, 219–220, 222
- B**
- Back-office grids, 456–457
- Bank of America, 320
- Barrier conditions
 ARB process as, 292, 294, 302
 creating for hybrid development models,
 296–297
 establishing performance testing for, 292
 including in Agile development,
 293–295
 JAD process as, 294
 overview, 301, 302
- uses for, 291–293
 waterfall development and, 295–296
- Batch cache refresh, 397
- Behavior
 evaluating employee, 109–110
 leaders and selfless, 79–80
 leadership influencing, 96
- Bezos, Jeff, 16–17
- Blackouts, 112
- Blah* as a Service offerings, 461–462
- Blink* (Gladwell), 262
- Blogs on scalability, 535
- “Blue-eyed/brown-eyed” exercise, 54,
 55–56
- Board of directors, 37
- Books on scalability, 535
- Boundaries
 fault isolation and swim lane, 338, 342
 finding optimal team, 44
 team, 65
- Boyatzis, Richard, 76–77
- Brewer, Eric, 378
- Brewer’s theorem, 378
- Brooks, Jr., F. P., 14, 16, 46, 448
- Brownouts, 112
- Budgets for headroom, 198, 208
- Buffers vs. caches, 396, 409
- Build grids, 455–456
- Build small, release small, fail fast, 221,
 222
- Build vs. buy decisions
 checklist of questions for, 255, 258,
 321–322
 considering strategic competitive
 differentiation, 253, 255
 cost-effectiveness of component
 building, 254–255
 cost-focused approaches to, 250–251
 developing and maintaining components
 in, 253–254, 255
 estimating competition for component,
 254, 255
 failures in build-it-yourself decisions,
 256–258
 making good buy decisions, 255–256
 merging cost and strategy approaches in,
 252–253, 258

- Build vs. buy decisions (*continued*)
 Not Built Here phenomenon and, 252
 overview, 258
 scalability and, 249–250
 strategy-focused approaches to, 251, 258
 TAD and, 323, 325
- Bureaucracy, 140
- Business change calendar, 187
- Business unit owners, 27–28
- Buy when non-core, 220, 222
- C**
- Cabrera, Felipe, 492
- Caches
 buffers vs., 396, 409
 cache hits, 397
 cache misses, 397, 398
 cache ratio, 397
 object, 399–402
 proxy, 402–403
 refreshing batch, 397
 reverse proxy, 403–405
 types of application, 401
- Caching
 content delivery networks, 407–408
 defined, 395–396, 409
 HTML headers vs. meta tags for controlling, 405
 LRU algorithm for, 397–398, 408
 MRU algorithm for, 398, 408
 software, 406–407
 structures for, 396, 409
- Calculating
 hardware uptime, 540–541
 headroom, 205, 206, 200–201
 headroom capacity, 547–553
 load and performance, 555–561
 load and performance for SaaS, 555–561
 tradeoffs using decision matrix, 309
 Web site availability, 539–545
- Callbacks, 415
- CAP theorem, 378
- Capability levels, 134–135
- Capability Maturity Model Integration (CMMI) project, 132, 134–135, 136–137, 140
- Capacity
 calculating headroom, 547–553
 maximizing with grid computing, 450, 451
 planning, 33–34, 203–205, 208
- Carnegie Mellon Software Engineering Institute, 134
- Case methods
 about, 9–10
 airline pricing models, 368–370
 Amazon, 16–17, 44, 123, 126, 281, 461, 462
 Amazon Web Services, 331, 488, 492, 516
 Apple, 11, 257–258
 eBay, 34–35, 123, 162–163, 388–389
 Etsy, 159–160, 239–240
 FAA's Air Traffic Control system, 181
 Friendster, 71–72
 Google, 91, 123, 462
 Google MapReduce, 435–438, 444, 457
 Intuit, 99–100, 102–105, 491–493
 Microsoft, 43, 99, 256, 462
 Netflix, 281
 PayPal, 123
 Quigo, 91, 92–94, 114–115, 210
 Salesforce, 330–331, 391–392
 Spotify, 68–69, 244–245
 Wooga, 244, 245–246
- Causal roadmap to success, 94–95, 97
- CD. *See* Continuous delivery systems
- CDNs (content delivery networks), 407–408, 409–410
- Centralization in user sessions, 421, 422
- CEOs (chief executive officers). *See also* Leaders
 accountability of, 24–25
 RASCI matrix and, 36–38
 role of, 25–26, 40
- CFOs (chief financial officers), 27–28
- Change control
 change control meetings, 191–192, 195
 performance/stress testing and, 288
- Change identification
 about, 179–180, 194
 change management vs., 181
- Change log, 179

- Change management
 - about, 180–183, 193–195
 - approving changes, 186–187, 194
 - change control meetings, 191–192, 195
 - checklist for, 193
 - continuous process improvement, 192, 195
 - FAA's Air Traffic Control system, 181
 - identifying change, 179–180, 181, 194
 - implementing and logging changes, 189, 194
 - ITIL goals of, 183
 - proposing changes, 183–186, 194
 - reviewing changes, 191, 194, 195
 - rollback plans for, 190
 - scheduling changes, 187–189, 194
 - validating changes, 189–190, 194
- Changes. *See also* Change management
 - about, 177–178
 - approving, 186–187, 194
 - defining, 178
 - implementing and logging, 189, 194
 - postmortems for, 153–156, 158, 172–173, 175
 - proposing, 183–186, 194
 - reviewing, 191, 194, 195
 - rollback plans for, 190
 - scheduling, 187–189, 194
 - using crises as catalyst for, 162–163
 - validating, 189–190, 194
- Chaos in crisis, 163–164, 174
- Chapters, 69, 245
- Chat channel, 166, 168–169, 175
- Checklists
 - Architecture Review Board review, 236
 - build vs. buy questions, 255, 258, 321–322
 - change management, 193
 - fast or right, 310–311
 - fault isolation design, 340–341
 - headroom calculation, 205
 - joint architecture design sessions, 227–228, 236
 - markdown, 301
 - performance testing steps, 280
 - risk assessment steps, 268
- rollbacks, 298
- team size, 50–51
- Chief executive officers. *See* CEOs
- Chief financial officers (CFOs), 27–28
- Chief technology officers. *See* CTOs
- Chipsoft, 99
- Chunk, 342
- CIOs (chief information officers). *See* CTOs
- Citibank, 320
- Cloud, 460
 - Cloud computing. *See also* IaaS; Preparing cloud applications
 - application portability in, 472, 474
 - benefits of, 468–471, 481
 - Blah as a Service* offerings, 461–462
 - common characteristics of, 466
 - control issues in, 472–473, 475
 - cost of, 469–470, 471, 474, 475
 - decision making steps for, 478–481, 482, 483
 - decision matrix, 479–480
 - drawbacks of, 471–476, 482–483
 - fitting to infrastructures, 476–478, 482, 483
 - flexibility of, 470–471
 - grids vs., 467–468
 - history of, 460–461, 481, 482
 - multiple tenants, 465
 - overview, 459, 481–483
 - pay by usage for, 463
 - performance of, 473–474, 475
 - public vs. private clouds, 462–463
 - scale on demand, 464–465
 - security liabilities of, 471–472, 474
 - skill sets needed for, 478
 - speed benefits of, 470, 471
 - UC Berkeley's assessment of, 475–476
 - using with production environments, 476–478
 - virtualization, 466–467
 - Clusters, 328, 329
 - COBIT (Control Objectives for Information and Related Technology), 143, 144
 - Code reviews
 - introducing, 292
 - using with RAD method, 296
 - waterfall development and, 296

- Coding. *See* Source code
- Cognitive conflict
- about, 54–55
 - defined, 13
 - influencing innovation, 63
- Collins, Jim, 79
- Communications
- during crises and escalations, 168–169, 171–173, 175
 - effects of experience on, 119–120
 - making customer apologies, 173
 - organizational influences in, 41–42
 - team size and poor, 47
 - within functional organizations, 53–54
- Companies. *See* Organizations
- Complexity
- data splitting for growth and, 371, 383
 - grid computing, 453, 458
 - of processes, 137–139
- Components
- considering scalability and, 321–322
 - cost-effectiveness of building, 254–255
 - developing and maintaining assets or, 253–254, 255
 - estimating competition for, 254, 255
 - example of good buy decisions, 255–256
 - failures in build-it-yourself decisions, 256–258
 - strategic competitive differentiation for, 253, 255
- Computers. *See also* Servers
- decreasing size of, 509–510
 - processing large data sets with distributed, 434–438, 444
- Conflicts
- “blue-eyed/brown-eyed” exercise, 54, 55–56
 - cognitive or affective, 54–55
 - incident and problem resolution, 150
 - types of, 13–14
 - when process not fitted to culture, 139–140, 141, 142
 - within organizations, 54, 66, 67
- Consistency
- database, 401
 - node, 378
- Content delivery networks (CDNs), 407–408, 409–410
- Continuous delivery (CD) systems
- about, 182
 - change approvals in, 187
 - change control meetings for, 192
 - unit testing in, 293
- Continuous process improvement, 192, 195
- Control in cloud computing, 472–473, 475
- Control Objectives for Information and Related Technology (COBIT), 143, 144
- Cook, Scott, 99
- Costs. *See also* Tradeoffs
- benefits using grid computing, 451
 - cloud computing, 469–470, 471, 474, 475
 - cost-value data dilemma, 430–431
 - data center, 509–511, 520, 521–525
 - data storage, 427–429, 432–433, 444
 - factoring in project triangle, 303–306, 313
 - focus in build vs. buy decisions, 250–251, 258
 - measuring cost of scale, 111–112
 - projecting data center, 515
 - reducing with fault isolation, 335–336
 - rollback, 299–300
 - technology-agnostic design and, 319–320, 326
 - y-axis splits, 348
- Cowboy coding, 295
- Creativity, 133–134
- Crises and escalations
- about escalations, 170–171, 175
 - characteristics of crises, 160–161
 - communications and control in, 168–169, 175
 - crises vs. incidents, 161–162, 174
 - eBay scalability crisis, 162–163
 - engineering lead’s role in, 167–168, 174
 - Etsy’s approach to, 159–160
 - from incident to crisis, 161
 - individual contributor’s role in, 167–168, 174
 - managing, 163–168
 - postmortems and communications about, 172–173, 175
 - problem manager’s role, 164–166, 174
 - status communications in, 171–172

team manager's role, 166–167
 using as catalyst for change, 162–163
 war rooms for, 169–170, 175

Crisis managers, 168, 174

Crisis threshold, 161

CTOs (chief technology officers)
 experiential chasm with, 121–122, 128
 problems with, 121–122
 RASCI matrix and, 37, 38
 responsibility for scalability, 2–3
 role of, 28–30, 40

Cultures
 candidates' fit into, 108, 116
 clashes with processes, 139–140, 141, 142
 productivity and behavior in, 11

Customers
 apologizing to, 173
 handling growing customer base, 371
 impact of downtime on, 543
 unprofitable, 431
 using complaints as metric, 541–542

D

D-I-D matrix, 307

Daily incident meetings, 152–153, 157, 158

Data. *See also* Data storage; Source code;
 Splitting databases
 analyzing performance test, 278–279,
 280, 555, 559–561
 assessing stress test, 285, 286
 caching, 395–399
 collecting repeat test, 279–280
 collecting to identify problems, 498–499
 cost-value dilemma for storing, 430–431
 costs of, 427–429, 444
 ETL concept for, 434, 439, 457
 high computational rates of grid
 computing, 450, 451, 458
 methods for synching, 411–412, 423
 NoSQL solutions for scalability,
 440–443, 445
 plotting on control charts, 560
 processing large data sets, 434–438, 444
 separating by meaning, function, or
 usage, 381–383
 transforming, 433–434
 y - and z -axis Big Data splits, 438–440,
 445

Data centers
 considering multiple, 525–527
 costs of, 509–511, 520, 521–525
 IaaS strategies vs., 516–519
 location of, 511–514
 projecting growth for, 514–516
 Rules of Three for, 519–525
 splitting, 521–525

Data sets
 processing large, 434–438, 444
 using y - and z -axis Big Data splits,
 438–440, 445

Data storage
 cost-value dilemma for, 430–431
 costs of, 427–429, 432–433, 444
 issues in, 427
 matching value to costs of, 431–434
 option value of data, 431–432
 overview, 444–445
 reducing data set size, 434–438, 444
 strategic competitive differentiation, 432
 tiered storage solutions, 432–433, 444
 transforming data, 433–434
 types of costs in, 429

Data warehouse grids, 456

Database servers, 548–549, 552

Databases. *See also* Splitting databases
 ACID properties of, 400, 401, 441
 calculating capacity for, 552
 cloning data without bias, 375, 376–377
 replication delays in, 377, 380
 using axes splits for, 387–388

Datum, 396, 399, 409

DDOS (distributed denial-of-service)
 attacks, 490

Deadlock, 412

Decentralized user sessions, 421, 422

Decision making. *See also* Build vs. buy
 decisions
 analyzing tradeoffs, 303–313
 ARB, 233–234
 evaluating management's, 106
 leadership and, 81–82
 mistakes in, 81
 snap judgements and, 262
 steps for cloud computing, 478–481

Decision matrix
 calculating tradeoffs with, 308, 309, 313
 for cloud computing, 479–480

- Delegation
 - by CEO, 26
 - by CTO/CIO, 29
 - guidelines for, 24–25
- Destructive interference, 120
- Development life cycle, 302
- DevOps responsibilities, 31–32
- Disabling systems
 - architectural design for, 215, 222, 300–301, 302
 - markdown checklist for, 301
- Distributed denial-of-service (DDOS)
 - attacks, 490
- Diversity
 - experiential, 63–64
 - network, 64, 242
- Documentation, 230, 234, 235
- DRIER process, 148–149, 157, 158
- Dunning, David, 76
- Dunning-Kruger effect, 76, 77
- Durability of databases, 401

- E**
- eBay, 34–35, 123, 162–163, 388–389
- Ecommerce scalability, 388–389
- Economies of scale, 244, 248
- Efficiency in organizations, 41–44
- 80/20 rule, 276
- Elliott, Jane, 55–56
- Employees
 - cultural and behavioral fit of, 108
 - recruiting for data centers, 514
 - signs of under- and overworked, 47–48
 - team size and experience of, 45
 - terminating, 109–110
- Empowerment
 - JAD entry/exit criteria for team, 229
 - team, 64–65
- Engineers
 - cowboy coding by, 295
 - escalating crises to managers, 171
 - fostering technology agnosticism in, 325
 - individual contributions by, 31
 - infrastructure, 32–33
 - measuring productivity of, 113–114
 - reporting performance test results to, 279, 280, 556, 561
 - role in crises, 167–168, 174

- Entry/exit criteria
 - joint architecture design, 228–230
 - used for waterfall implementations, 295–296
- Environments. *See* Preparing cloud applications; Production environments
- Equations for headroom, 201, 202, 551
- Escalations. *See* Crises and escalations
- Ethics
 - leadership and, 78–79, 96
 - managerial, 100–101
- ETL (extract, transform, and load) concept, 434, 439, 457
- Etsy, 159–160, 239–240
- Everything as a Service (XaaS), 461, 462, 483
- Executing performance tests, 277–278, 280, 555, 557–559
- Executive interrogation, 26
- Executives
 - business unit owners, general managers, and P&L owners, 27–28
 - CEOs, 25–26, 40
 - CFOs, 27–28
 - CTO/CIO, 28–30
 - experiential chasm with, 120–121, 128
- Experiential chasm, 119–120, 128
- Experiential diversity, 63–64
- Extract, transform, and load (ETL) concept, 434, 439, 457

- F**
- FAA's Air Traffic Control system, 181
- Facebook, 71, 72
- Fail Whale, 197
- Failure domains, 49, 50, 147
- Failure mode and effects analysis (FMEA), 264–267, 268, 270–271
- Failures
 - build small, release small, fail fast design principle, 221, 222
 - cloud environment outages, 487–489, 494
 - communication, 41–42
 - effects of, 21–23
 - leadership, 71–72
 - Microsoft's Longhorn, 43

- Fannie Mae, 320
 Fast or right checklist, 310–311
 Fault isolation
 along swim lane boundary, 338, 342
 approaches to, 336–367
 architectural terms for, 327–329
 benefits of, 329–336
 challenges for cloud applications, 487–489, 494
 costs and, 335–336
 design checklist for, 340–341
 examples of, 327
 implementing, 339–341
 increasing availability with, 329–333, 334, 342
 no shared components or data, 337
 scalability and, 334
 testing designs for, 341
 time to market and, 334–335
 with transactions along swim lanes, 338
 Fault isolation zones, 221, 222, 338
 Features
 analyzing tradeoffs for, 307–310
 documenting tradeoffs for, 230, 234, 235
 JAD entry/exit criteria for, 228–230
 requiring approval by ARB, 230
 Federal Aviation Administration (FAA) Air Traffic Control system, 181
 FeedPoint, 91
 5–95 Rule, 104, 105, 117
 Flexibility
 cloud computing, 470–471
 NoSQL solution and query, 442
 FMEA (failure mode and effects analysis), 264–267, 268, 270–271
 Ford, Henry, 347
 Forward proxy cache, 402–403
 Foster, Ian, 447
 Freddie Mac, 320
 Friendster, 71–72
 Functional organizational structure
 characteristics of, 51–56, 70
 communications within, 53–54
 conflicts within, 54, 66, 67
 illustrated, 52
 matrix organizations vs., 58
- G**
- Galai, Yaron, 91
 Gates, Bill, 26, 43, 256
 Gateway caches, 404, 409
 General managers, 27–28
 Gladwell, Malcolm, 262
 Globus Toolkit, 447, 448
 Go/no-go processes. *See* Barrier conditions
 Goal trees, 114–115, 118, 209–210
 Goals
 applying to scalability solutions, 93–94
 change management, 183
 creating goal tree, 114–115
 D-I-D matrix for project, 307
 defined, 89, 97
 developing architectural principles from, 209–211, 223
 ineffectiveness in shared, 22–23
 SMART, 89–90, 96, 97, 156, 211
Good to Great (Collins), 79
 Google, 91, 123, 462
 Google MapReduce, 435–438, 444, 457
 Graph databases, 443
 Graphs
 headroom, 207
 Web site availability, 544–545
 Grid computing
 back-office grids, 456–457
 build grids in, 455–456
 cloud computing vs., 467–468
 complexity of, 453, 458
 cons of, 452–453
 data warehouse grids, 456
 high computational rates of, 450, 451, 458
 history of, 447–449
 implemented in MapReduce, 457
 maximizing capacity used, 450, 451
 monolithic applications and, 452–453, 458
 overview, 457–458
 production grids in, 454
 pros of, 449–451, 458
 shared infrastructure for, 450, 451, 452, 458
Grid, The (Foster and Kesselman), 447

Growth

- dealing with too much data, 427
 - own/lease/rent options plotted against, 518–519
 - projecting data center, 514–516
 - projecting in headroom calculations, 200–201
 - using AKF Database Scale Cube for, 392
 - y-axis application splits for, 371
 - y-axis database splits for, 383
 - z-axis database splits for, 383–385
 - z-axis splits for customer base, 371
- Guilds, 69, 245
- Gut-feeling risk assessment, 261–263, 271, 308, 313

H

- Half racks, 510
- Hardware, 220, 222
- Headroom
 - calculating, 205
 - determining, 199–203
 - equations for, 201, 202, 551
 - ideal usage percentages, 203–205
 - overview, 207–208
 - performance/stress testing and, 288
 - planning, 198–199
 - spreadsheet for, 206–207
- Healthcare.gov, 331–332, 333
- Heat maps, 498
- Hewlett-Packard, 462
- Hiring
 - cultural interviews before, 108, 116
 - headroom calculations and, 198, 208
 - selecting candidates, 107–108
- Hit ratio, 397, 409
- Hoare, Sir Charles Anthony Richard, 412
- Homo homini lupo* strategies, 65
- Hotlinks, 71
- HTTP (Hyper-Text Transfer Protocol)
 - headers in, 405
 - stateless protocol for, 418, 424
- Human factors in risk management, 270–271
- HVAC services, 510, 512, 513, 528, 529

I

- IaaS (Infrastructure as a Service)
 - concept of, 461, 462, 482
 - PaaS vs., 489
 - scaling features for, 219
 - shifting from data centers to, 516–519, 527–528
- IBM, 256, 461, 482
- Incident monitors, 500–501
- Incidents. *See also* Crises and escalations
 - assessing data for, 503–504
 - assigning swim lanes to sources of, 339–340
 - conflicts in managing, 150
 - crises vs., 161–162, 174
 - daily meetings about, 152–153, 157, 158
 - defined, 144–145, 158
 - DRIER process for managing, 148–149, 157, 158
 - escalating to crisis, 160–161
 - finding, 496–503, 507
 - life cycles for, 150–151
 - management components of, 146–149
 - monitoring with failure domains, 147
 - overlooking, 495–496
 - postmortems for, 153–156, 158
 - quarterly reviews of, 153, 157, 158
 - resolving, 147
- Individual contributors
 - architects, 30–31
 - capacity planning, 33–34
 - DevOps responsibilities, 31–32
 - engineers, 31
 - infrastructure engineers, 32–33
 - quality assurance, 33
- Information Technology Infrastructure Library (ITIL), 143, 144, 145, 146, 148, 149, 183
- Infrastructure as a Service. *See* IaaS
- Infrastructure engineers, 32–33
- Infrastructures. *See also* IaaS
 - fitting cloud computing to, 476–478, 482, 483
 - sharing for grid computing, 450, 451, 452, 458
- Initiating new threads, 413, 423

IInnovation

- cognitive/affective conflict and, 63, 66
 - defined, 62–63
 - experiential diversity and, 63–64
 - network diversity, 64, 242
 - sense of empowerment and, 64–65, 66
 - team boundaries and, 63
 - theory of innovation model, 66
- Input/output per second (IOPS), 489, 494
- Internet pipe costs, 512
- Intuit, 99–100, 102–105, 491–493
- IOPS (input/output per second), 489, 494
- IRC channels, 166, 168–169, 175
- Isaacson, Walter, 11, 257–258
- Isolation of databases, 401
- Issue management, defined, 144
- IT organization model, 122–124
- ITIL (Information Technology Infrastructure Library), 143, 144, 145, 146, 148, 149, 183
- ITSM (IT Service Management) framework, 183
- Itzhak, Oded, 91
- Ivarsson, Anders, 68, 244

J

- JAD (joint architecture design)
checklist for, 227–228, 236
defined, 225
designing for Agile teams, 242, 247
entry/exit criteria for, 228–230
fixing organizational dysfunction with, 225–226
following TAD rules, 324
function of, 226–227
implementing ARB and, 237
membership of, 237
overview, 236–237
using AKF Scale Cube with, 353
using as barrier condition, 294
- JavaScript Object Notation (JSON), 443
- Jobs, Steve, 10–11, 257–258
- Joint architecture design. *See* JAD

K

- Keeven, Tom, 34–35, 330
- Kesselman, Carl, 447
- Key performance indicators (KPIs), 496
- King, Jr., Martin Luther, 54, 55
- KLOC (thousands of lines of code), 113–114
- Kniberg, Henrik, 68, 244
- KPIs (key performance indicators), 496
- Kruger, Justin, 76

L

- Law of the Instrument, The, 34

Leaders

- aligning to shareholder value, 83, 96
 - asking questions, 25–26
 - behavior of, 11
 - born or made, 73, 96
 - building team relationships, 119–122, 128
 - dealing with conflicts, 55
 - decision making by, 81–82
 - delegation by, 24–25
 - developing causal roadmap to success, 94–95, 97
 - developing vision statements, 84–87
 - empowering teams and scalability, 82–83
 - implementing scalability, 532–534
 - making build vs. buy decisions, 249–258
 - morality of, 75, 81–82, 96
 - overview, 95–97
 - problems with executives, 120–121
 - resolving crises, 174
 - scalability proficiency of, 26
 - seeking outside help, 26
 - selfless behavior of, 79–80
 - setting examples, 78–79, 96
 - SMART goals developed by, 89–90
 - 360-degree reviews for, 77, 96
 - transformational leadership by, 84, 96
 - valuing people, 80–81
- Leadership. *See also* Leaders
- assessing abilities for, 76–78, 96
 - attributes of, 74–75, 96

- Leadership (*continued*)
creating mission statements, 87–89
decision making and, 81–82
defined, 72–73, 96
developing qualities for, 73, 96
ethics and, 78–79, 96
failures in, 71–72
importance in scalability, 17–19, 20
management vs., 17–18, 73, 101
model of, 74–76
overview, 95–97
selfless behavior and, 79–80
transformational, 84, 96
working with limited Agile resources, 242–243, 247
- Life cycles
development, 302
problem and incident, 150–151
- Live Community, 492–493
- Loads. *See also* Performance testing
calculating performance of, 555–561
performance testing for server, 274
stress testing, 283, 284, 286
- Location of data centers, 511–514
- Lockheed Martin, 259
- LRU (least recently used) caching algorithm, 397–398, 408
- M**
- Management. *See also* Managers;
Managing incidents and problems
building teams, 105–107
contingencies for project, 104–105
creating goal tree, 114–115
creating team success, 115–116, 117, 118
defining, 100–101, 116, 117
developing crisis management process, 163–164
ethics in, 100–101
evaluating measurement metrics and goals, 111–114, 117, 118
experiential chasm with teams, 119–120, 128
5–95 Rule for, 104, 105, 117
implementing scalability, 532–534
importance in scalability, 17–19, 20
interviewing candidates, 108, 116
leadership vs., 17–18, 73, 101
leading postmortems, 153–156, 158
managing chaos in crisis, 163–164, 174
measuring output, 12–13
overview, 116–118
problem managers, 164–166, 174
problems with business leaders, 120–121
problems with CTOs, 121–122
project and task, 102–105
recognizing badly fitted processes, 139–140, 141, 142
similarities with leadership, 117
team managers in crises, 166–167
upgrading teams, 107–111
using IT model for customer product, 122–124
when to implement processes, 137
- Managers
appointing, 49, 51, 52
creating team success, 115–116
crisis, 168, 174
determining crisis threshold, 161
good, 102
measuring performance, 111–114
problem, 164–166
responsibilities of, 45–46
seed, feed, and weed activities for, 107–111, 117
selecting employment candidates, 107–108
team size and experience of, 45
terminating employees, 109–110
working with limited Agile resources, 242–243, 247
- Managing incidents and problems
about incidents, 144–145
components of incident management, 146–149
components of problem management, 149–150
conflicts when, 150
daily incident meetings, 152–153, 157, 158
defining problems, 145–146, 158
DRIER process, 148–149, 157, 158
flow for, 156–157

- incident and problem life cycles, 150–151
monitoring systems for, 147
overview, 143–144
postmortems, 153–156, 158
quarterly incident reviews, 153, 157, 158
Manifesto for Agile Software Development, 59, 293
MapReduce, 435–438, 444, 457
Markdown functionality, 300–301, 302, 333
Marshalling processes, 399
Maslow's Hammer, 34
Matrices
 D-I-D, 307
 decision, 308, 309
 RASCI, 35–39
Matrix organizations
 characteristics of, 56–59, 70
 functional organizations vs., 58
 illustrated, 57
 moving goal lines in, 66
Mature technologies, 217, 222
Maturity levels, 134–135, 140
MBPS (megabytes per second), 489, 494
McCarthy, John, 461
McKee, Annie, 76–77
Mealy machine, 419
Measurements. *See also Metrics; Risks*
 cost of scale, 111–112
 evaluating metrics and goals for, 111–114, 117, 118
 managers' support for, 102
 measuring availability, 124–126
 using as barrier conditions, 293, 295
Meetings
 Architecture Review Board, 232–234
 change control, 191–192, 195
 daily incident, 152–153, 157, 158
Megabytes per second (MBPS), 489, 494
Membership
 Architecture Review Board, 230–232
 joint architecture design, 237
Metrics
 customer complaints used as, 541–542
 deriving from performance testing, 274
 finding incidents using, 499–501
 measuring output with, 12–13
 needed for project, 111–114, 117, 118
Micromanagement, 48
Microsoft, 43, 99, 256, 462
Mission, 97
Mission First, People Always, 80–81, 96
Mission statements
 applying to scalability solutions, 92–93
 creating, 87–89, 97
Mitigating failure, 265–267
Monitoring
 applications, 503
 correlating data size to problem specificity, 498
 designing systems for, 215–216, 222, 495–503, 507
 establishing barrier conditions with, 293
 existing platforms for, 503, 507
 failure domains, 147
 learning what to monitor, 496–499
 overview, 506–507
 processes, 504–507
 stress test processes, 284, 286
 user experience and business metrics for, 499–501
 using system, 501–502
 value of, 503–504
 Web and application server requests, 548–549
Monolithic applications, 452–453, 458
Moore, Gordon, 509
Moore machine, 419
Moore's law, 219, 509
Morale, 22, 47
Morality of leaders, 75, 81–82, 96
MRU (most recently used) caching algorithm, 398, 408
Multiple live sites, 216, 222, 521–528, 529
Multitenant states, 219
Multitenants
 cloud computing with, 465
 using SaaS database splitting for products, 391–392
Mutex synchronization, 411, 423
Mythical Man-Month, The (Brooks, Jr.), 14, 16, 46, 448

N

- NBH (Not Built Here) phenomenon, 252
- Negative stress testing, 182
- Netflix Chaos Monkey, 281
- Network architecture, 460
- Network diversity, 64, 242
- NeXT, 11, 257, 258
- N + 1* design, 213–215, 222
- No shared components or data, 337
- NoSQL solutions
 - implementing, 434–438, 440, 444
 - scalability using, 440–443, 445
 - using multiple nodes in, 440
- Not Built Here (NBH) phenomenon, 252

O

- Object caches, 399–402
- Office of Government Commerce (United Kingdom), 143, 146
- Option value of data, 431–432
- Organizational cost of scale, 14–17
- Organizational design. *See also* Joint architecture design
 - Agile Organizations, 59–70
 - cognitive conflict and, 13–14
 - creating efficiency with, 41–44
 - determining team size, 44–51
 - functional organizational structure, 51–56, 70
 - matrix organizational structure, 56–59, 70
 - overview, 69–70
 - signs of incorrect team size, 47–48
- Organizations. *See also* Agile
 - Organizations; Organizational design
 - choosing application splits for, 370–371
 - clarity of roles in, 21–23, 40
 - costs and build vs. buy decisions, 250–251, 258
 - creating mission statements, 87–89
 - customer apologies by, 173
 - defining team roles, 23–24
 - delegation within, 24–25
 - designing, 20
 - fitting cloud computing to, 476–478, 482, 483

- fostering standards within, 42–43
- introducing processes into, 135–139
- JAD for dysfunctional, 225–226
- mapping goals for, 114–115
- planning scalability for, 532–534
- as scalability element, 11–17, 20
- scalability needs of, 4
- strategy-focused build vs. buy decisions, 251, 258
- vision statements for, 86–87
- OSI Model, 460
- Outage costs, 126–127, 128
- Own/lease/rent options, 517, 518–519
- Ownership
 - assigning for team processes, 140, 142
 - engendering team architectural, 213–214, 241–242
 - product standards and, 44

P

- P&L owners, 27–28
- PaaS (Platform as a Service), 462, 483, 489
- Paging, 490
- Pareto, Vilfredo Federico Damaso, 276
- Partition tolerance in distributed computer systems, 378
- Paterson, Tim, 256
- Patient Protection and Affordable Care Act (PPACA), 331–332
- Pay-as-you-go cloud computing, 463
- PayPal, 123
- People
 - assessing abilities of, 76–78, 96
 - conflict between groups of, 54
 - importance in scalability, 10–11, 20, 61, 531–532
 - leader's valuing of, 80–81
 - leadership attributes in, 74–75, 96
 - managing, 116–118
 - team size and productivity of, 15
- Performance. *See also* Caching; Performance testing
 - buffers and caches for, 409
 - cloud computing, 473–474, 475
 - data thrashing, 203–204

- identifying stress test objectives, 281–282, 285
management measurement of, 111–114, 118
metrics and goals for, 111–114, 117, 118
variability in cloud I/O, 489–491, 494
- Performance testing
analyzing data from, 278–279, 280, 555, 559–561
appropriate environments for, 275–276, 280, 289, 555, 556
defining, 276–277, 289, 555, 556
executing, 277–278, 280, 555, 557–559
goals of, 289
load testing, 274, 289
overview, 288–290
performing, 273–274
relating to scalability, 287–288, 290
repeating and analyzing, 279–280, 556, 561
reporting results of, 279, 280, 556, 561
steps in, 280, 289, 555–561
success criteria for, 274, 280, 555, 556
using as barrier condition, 292
- Pixar, 11
- Planning
capacity, 33–34
headroom, 198–199, 203–205, 208
organization's scalability, 532–534
performance test definition, 276–277, 289
project contingencies, 105
rollbacks, 297
- Platform as a Service (PaaS), 462, 483, 489
- Pods, 328, 329, 330, 342, 391–392
- Pools, 328–329, 344–345, 347–348
- Portability between clouds, 472, 474
- Positive stress testing, 281
- Postmortems
after crises, 172–173, 175
leading, 153–156, 158
noticing incidents too late, 495–496
recognizing early signs of problems in, 496–499
- Power utilization of data centers, 510, 512, 513, 528, 529
- PPACA (Patient Protection and Affordable Care Act), 331–332
- Preparing cloud applications
applying Scale Cube in cloud, 485–487, 493, 494
fault isolation challenges, 487–489, 494
Intuit case study, 491–493
overview, 485, 493–494
variability in input/output, 489–491, 494
- Principles. *See* Architectural principles
- Problems
conflicts managing incidents and, 150
defined, 145–146, 158
detecting, 496–503, 507
developing monitors to detect, 502–503
identifying incidents from, 495–499
life cycles for, 150–151
locating indicators of, 501
management components of, 149–150
postmortems for, 153–156, 158
resolving, 147
- Processes. *See also* Barrier conditions
assigning ownership for team, 140, 142
automating, 221–222
choosing, 138–139, 141
CMMI framework for, 132, 134–135, 136–137
complexity of, 137–139, 141
continuous improvement of, 192, 195
creating crisis management, 163–164
culture clash with, 139–140, 141, 142
defined, 132
developing code without team, 295
identifying stress testing, 282, 286
marshalling and unmarshalling, 399
overview, 132, 140–142
value of, 131–132, 140–141
when to implement, 137, 141
- Proctor & Gamble, 99
- Product organization model, 122–124
- Production environments. *See also* Rollbacks
change identification in, 179–180, 181, 194
cloud computing uses for, 476–478
continuous delivery in, 182
markdown functionality for, 333
mimicking for performance testing, 275–276, 280, 289, 555, 556
simulating for stress testing, 283, 286

- Production grids, 454
- Productivity
- measuring, 12–13
 - organizational cost of scale, 14–17, 20
 - right behaviors and, 11
 - team size and poor, 47
- Products. *See also* Features; Projects; Standards
- automating processes, 221–222
 - determining headroom for, 197–208
 - drawbacks of stateful, 218–219
 - implementing search as own service, 389–391
 - JAD entry/exit criteria for features, 228–230
 - using IT model for customer, 122–124
- Project triangle, 303–306, 313
- Projects
- fast or right tradeoffs in, 303–313
 - 5–95 Rule for, 104, 105, 117
 - management contingencies for, 104–105
 - managing, 102–105, 116
 - triangle of tradeoffs in, 303–306, 313
- Pros-and-cons comparisons, 308–309, 313
- PROS software systems, 368–370
- Proulx, Tom, 99
- Proxy caches, 402–403, 408
- Proxy server, 402–403
- Public vs. private clouds, 462–463
- Pulling activities, 17, 19
- Pushing activities, 17, 18
- Q**
- Quality. *See also* Tradeoffs
- analyzing tradeoffs in, 307–311
 - defining project, 304–305
 - factoring in project triangle, 303–306, 313
 - measuring, 113
- Quality assurance, 33
- Quarterly incident reviews, 153, 157, 158
- Query flexibility for NoSQL solutions, 442
- Questions
- asking, 25–26
 - build vs. buy, 255, 258, 321–322
- evolving for system monitoring, 496–501, 507
- regarding data center location, 513–514
- QuickBooks, 99, 100
- Quicken, 99, 100
- Quicksort algorithm, 412
- Quigo, 91, 92–94, 114–115, 210
- R**
- Rack units (U), 509–510
- RAD (rapid application development), 296–297
- RASCI acronym
- applying to architectural principles, 214, 223
 - defining roles using, 36–39, 40, 156
- Repeating performance tests, 279–280, 556, 561
- Replication delays, 377, 380
- Resonant Leadership* (Boyatzis and McKee), 76–77
- Resources
- example of resource contention, 412
 - further reading on scalability, 535
 - working with limited Agile, 242–243, 247
- Response time measurements, 112
- Reverse proxy caches, 403–405, 408, 409
- Reviews. *See also* ARB
- change, 191, 194, 195
 - code, 292, 296
 - quarterly incident, 153, 157, 158
 - 360-degree, 77, 96
- Richter-Reichhelm, Jesper, 245–246
- Right behaviors, 11
- Right person, 10–11
- Risk management
- assessing risk, 268
 - continuous process improvement and, 192–195
 - human factors in, 270–271
 - importance in scalability, 259–261
 - measuring risk, 261–267
 - overview, 271–272
 - relation of performance and stress testing to, 288

- risk model, 260
rules for acute, 268–270
- Risks
evaluating production tradeoffs, 307–311
FMEA (failure mode and effects analysis), 264–267, 268, 270–271
gut feel method, 261–263, 271, 308, 313
identifying for change, 185–186
managing acute, 268–270
measuring, 261–267
noting high-risk features, 235
plotting own/lease/rent options against, 517
risk assessment steps, 268
technology-agnostic design and, 320, 326
tradeoff rules for, 311, 313
traffic light method, 263–264, 268, 271
when changing schedules, 187–188
- Roadmap to success, 94–95
- Roles
clarity in team, 21–23, 40
defining, 23–24
engineering lead's, 167–168, 174
executives, 25–30
individual contributors, 30–34, 167–168, 174
missing skill sets and, 34–35
problem manager, 164–166, 174
RASCI matrix for defining, 35–39
responsibilities of, 35–39
team managers in crises, 166–167
- Rollbacks
checklist for, 298
costs of, 299–300
designing architecture for, 215, 222, 302
incorporating in change management, 190
planning for, 297
rollback insurance policy, 298, 299, 302
technical considerations for, 298–299
version numbers for, 299
window for, 297–298
- Rules
acute risk management, 268–270
Pareto 80/20 rule, 276
- technology-agnostic design, 323–325
tradeoff, 311, 313
- Rules of Three
about, 215, 528–529
applying to data centers, 519–525
- S
- SaaS (Software as a Service)
capacity planning calculations for, 547–553
evolution of, 59, 60, 461, 462, 482
Intuit's development of, 99–100
load and performance calculations for, 555–561
projects using functional organizations, 53
using database splits with, 391–392
variability in cloud input/output, 489–491, 494
- Saban, Nick, 131
- SABRE (Semi-automated Business Research Environment) reservation system, 367–368
- Salesforce, 330–331, 391–392
- Scalability. *See also* AKF Scale Cube; Data storage; Headroom
Agile Organizations and, 60–61
art vs. science of, 4, 531–532
barrier conditions for, 292–293
build vs. buy decisions and, 249–250
business case for, 124–127, 128
caching and, 395, 409
cloud computing and, 459, 481–483
collaborative design for, 226–227
company needs for, 4
defining direction for, 90–94
eBay crisis in, 162–163
effect of crises on, 161
empowering, 82–83
fault isolation and, 334
grid computing in, 449–451, 458
headroom calculations and, 198–199, 208
implementing, 532–534
incidents related to, 144–145
issues in, 2–3

- Scalability (*continued*)
 management and leadership in, 17–19, 20, 82–83
 managing changes, 177–178
 NoSQL solutions for, 440–443, 445
 organizational cost of scale, 14–17
 organizational factors in, 11–17, 20, 41–44
 people and, 10–11, 20, 531–532
 processes in, 131–132
 resources on, 535
 risk management in, 259–261
 scale on demand cloud computing, 464–465
 scaling agnostically, 250
 scaling out, not up, 219, 222
 sports analogy for, 105–107
 supporting with TAD, 321–323, 326
 synchronous or asynchronous calls in, 414–415, 422–423
 team failures in, 21–23
 tradeoffs in, 306–307
 using multiple axes of, 365–367
 vicious/virtuous cycles in, 3, 532, 533
 x-axis splits and, 359–361
 y-axis splits and, 361–362
 z-axis splits and, 363–364
 Scalability projects, 201
 Scheduling changes, 187–189, 194
 Scope
 effect on project triangle, 305
 factoring in project triangle, 303–306, 313
 scalability and, 306
 Sculley, John, 11
 Search as own service, 389–391
 Search engine marketing, 91
 Seattle Computer Products, 255–256
 Security in cloud computing, 471–472, 474
 Seed, feed, and weed activities, 107–111, 117
 Servers. *See also* Architectural principles; Performance testing; Stress testing
 applying Rule of Three to data center, 520–521, 528, 529
 asynchronous system design for, 217–218, 222
 capacity planning calculations for, 547–553
 decreasing size of computers, 509–510
 Etsy upgrades for, 239–240
 headroom usage percentages for, 203–205
 implementing reverse proxy, 404–405
 monitoring requests for Web and application, 548–549
 proxy cache implementation for, 402–403
 reverse proxy caches for, 409
 simulating environment for stress testing, 283, 286
 ServiceNow, 392
 Shard, 328, 329, 342
 Shared goals, 22–23, 103–104
 Shared infrastructure for grid computing, 450, 451, 452, 458
 Shareholders
 leader alignment to values of, 83, 96
 leader's pursuit of value for, 79–80
 Silo organizations. *See* Functional organizational structure
 Skill sets, 34–35, 478
 Slivers, 328, 329
 Smart, Geoff, 108
 SMART goals, 89–90, 96, 97, 156, 211
 Smith, Adam, 244
 Software, caching, 406–407
 Software as a Service. *See* SaaS
 Software Engineering Institute, 132, 134
 Source code
 compilation steps for build grids, 455–456
 cowboy coding, 295
 introducing reviews of, 292, 296
 KLOC, 113–114
 measuring engineer's production of, 113–114
 splitting into failure domains, 49, 50
 Speed. *See also* Tradeoffs
 analyzing tradeoffs in, 307–311
 benefits of cloud computing, 470, 471
 defining project, 305
 factoring in project triangle, 303–306, 313

- Splits. *See also* AKF Scale Cube; Splitting applications; Splitting databases
Big Data, 438–440, 445
making team, 49–51
service- or resource-oriented, 436–437
using data center live sites, 521–525
within cloud environments, 485–487,
493, 494
- Splitting applications
effects of, 370–371
overview, 357, 371–373
x-axis for AKF Application Scale Cube, 357–358, 359, 361, 365–367,
371–372
y-axis for AKF Application Scale Cube, 357–358, 361–362
z-axis for AKF Application Scale Cube, 358, 363–364
- Splitting databases
computing headroom using *x*-axis splits, 550
using AKF Scale Cube for, 375–376
y- and *z*-axis data splits, 438–440, 445
- Spotify Agile team structure, 68–69,
244–245
- Squads, 68–69, 244–245
- Srivastava, Amitabh, 43
- Stability, defined, 177
- Standards
Agile methodologies and, 293
fostering within organizations, 42–43
maintaining across teams, 243–246, 248
ownership of product, 44
- Stansbury, Tayloe, 100, 102–103
- State machines, 419, 424
- Stateful applications
avoiding, 218–219
user sessions and, 420–422, 424
when to use, 351
- Stateless systems
advantages of, 218–219, 222, 351
using stateless session data, 420–421
- Statistical process control chart (SPCC), 500, 501
- Status communications, 171–172
- Steve Jobs (Isaacson), 11
- Strategic competitive advantage, 430
- Strategic competitive differentiation, 432
- Strategy-focused approaches in build vs.
buy decisions, 251, 258
- Stress testing
about, 281, 289
analyzing data from, 285, 286
creating load for, 284, 286
determining load for, 283, 286
establishing environment for, 283, 286
executing tests, 284–285, 286
key services in, 282, 286
objectives for, 281–282, 285
overview, 288–290
processes for monitoring, 284, 286
relating to scalability, 287–288, 290
steps in, 285–286, 289–290
- Structures for caching, 396, 409
- Success
causal roadmap to, 94–95, 97
managing path to, 115–116, 117, 118
- Success criteria for performance tests, 274,
280, 555, 556
- Sun Grid Engine, 448
- Sun Tzu, 9, 21, 41, 71, 99, 119, 131, 143,
159, 177, 197, 209, 225, 239, 259,
273, 291, 303, 317, 327, 343, 357,
375, 395, 411, 427, 447, 459, 485,
509, 531
- Swim lanes
along natural barriers, 340, 341
defined, 327–328, 329, 342
identifying recurring incidents for own,
339, 341
isolating money-makers within, 339,
340, 342
measuring service availability and,
542–543
no crossing boundaries of, 338, 342
transactions along, 338, 342
- Synchronization. *See also* Asynchronous
design
methods of, 411–412, 423
mutex, 411, 423
using synchronous vs. asynchronous
calls, 412–413, 414–415, 422–423
- Systems monitoring, 501–502

T

TAA (technology-agnostic architecture), 318–319, 323–325

TAD. *See* Technology-agnostic design

Tags

controlling caching with meta, 405
tag-datum cache structure, 396, 399, 409

Tasks, 102–105, 116

Team size

checklist for, 50–51

constraints on, 45–46

employee experience and, 45

factors increasing, 46

growing or splitting teams, 48–51

managers and, 45–46

overview, 70

signs of incorrect, 47–48

Teams. *See also* Team size

autonomy of Agile, 241–242, 247

boundaries of, 65

building, 105–107

business unit owners, general managers, and P&L owners, 27–28

capacity planning for, 33–34

CFOs of, 27–28

choosing processes for, 138–139, 141

collaboration between, 103–104

CTO/CIO of, 28–30

developing and using processes, 132–134

DevOps responsibilities, 31–32

egotist behavior when leading, 79–80, 96

engineers, 31–33

feeding members of, 108–109, 117

growing or splitting, 48–51

infrastructure, 32–33

interviewing new members, 108, 116

IT model for customer product, 122–124

leader's empowering of, 82–83

maintaining standards across, 243–246, 248

management and path to success, 115–116, 117, 118

organization of Agile, 61

ownership of principles by, 213–214

quality assurance, 33

recognizing badly fitted processes,

139–140, 141, 142

relationships with management,

119–122, 128

roles in, 21–24, 35–39

scaling cross-functional designs,

226–227

seeding performance of, 107, 110, 117

selecting architectural principles,

212–213

sense of empowerment for, 64–65

sharing goals, 22–23

size and productivity of, 15

skill sets of, 34–35, 478

Spotify's, 68–69, 244–245

structure of, 14

system architects, 30–31

team manager role, 166–167

tool for defining roles, 35–39

Two-Pizza Team rule, 16–17

upgrading, 107–111

weeding individuals from, 109–110, 117

Wooga cross-functional, 245–246

working with limited resources,

242–243, 247

Technology. *See also* CTOs

calculating hardware uptime, 540–541

missing skill sets in, 34–35

vicious and virtuous cycles in, 3, 532

Technology-agnostic architecture (TAA),

318–319, 323–325

Technology-agnostic design (TAD)

about TAA and, 318–319

availability and, 323, 326

build vs. buy decisions and, 323, 325

costs of, 319–320, 326

risks and, 320, 326

rules for, 323–325

supporting scalability with, 321–323,

326

Terminating employees, 109–110

Testing. *See* Performance testing; Stress testing

Theory of innovation model, 66

Thin slicing, 262

Thousands of lines of code (KLOC),

113–114

Thrashing, 203–204
 Threads
 initiating new, 413, 423
 swapping synchronously, 414
 thread join synchronization, 412
 360-degree review process, 77, 96
 Tiered storage solutions, 432–433, 444
 Time
 summarizing headroom, 202–203
 time to market and fault isolation, 334–335
 Tradeoffs
 assessing feature, 230, 234, 235
 factors in project triangle, 303–306, 313
 fast or right checklist, 310–311
 risks and rules for, 311, 313
 weighing risks in fast or right, 307–311
 Traffic light risk assessment, 263–264, 268, 271
 Transactions
 implementing eBay database splits for, 388–389
 reducing time with *y*-axis database splits, 382
 x-axis splits for growth in, 371
 z-axis database splits for scaling growth of, 383–385
 Transformational leadership, 84, 96
 Transforming data, 433–434
 Tribes, 69, 245
 TurboTax, 99, 100, 492
 Twitter Fail Whale, 197
 “Two-Pizza” Rule, 16–17, 44

U

UNICORE (UNiform Interface to Computing REsources), 448
 Unit testing, 293
 Unprofitable customers, 431
 Upgrading teams, 107–111
 U.S. Constitution
 goals within, 89–90
 mission outlined in, 88
 vision of Preamble, 86
 U.S. Declaration of Independence, 86
 U.S. Pledge of Allegiance, 85–86

User sessions
 approaches to scaling in, 420–422, 424
 avoidance in, 420–421, 422
 centralization in, 421, 422
 decentralization in, 421, 422
 stateful applications and, 420
 Utilization plotted against own/lease/rent options, 518–519

V

Validating changes, 189–190, 194
 Venn diagrams, 212–213
 Vicious/virtuous technology cycles, 3, 532
 Virtualization for cloud computing, 466–467
 Vision, 96
 Vision statements
 applying to scalability solutions, 92
 developing, 84–87, 96–97
 managing teams toward, 103–104
 Von Moltke, Helmut, 104

W

War rooms, 169–170, 175
 Waterfall development models, 295–296
Web Operations (Allspaw), 159
 Web pages, 405, 408
 Web servers, 548–549, 551
 Web site availability, 539, 540–541
Who (Smart), 108
 Wilson, Mike, 389
 Wooga, 244, 245–246

X

X-axis
 AKF Application Scale Cube, 357–358, 359, 361, 365–367, 371–372
 AKF Database Scale Cube, 376–381, 386, 387, 393
 AKF Scale Cube, 344–346, 351, 352, 354
 cloud environment and Scale Cube, 485–486, 493, 494
 when to use database splits, 393, 394
 XaaS (Everything as a Service), 461, 462, 483

Y**Y-axis**

- AKF Application Scale Cube, 357–358, 359, 361–362, 365–367, 371, 372–373
- AKF Database Scale Cube, 381–383, 386, 387, 388, 393–394
- AKF Scale Cube, 346–348, 351, 352, 354
 - cloud environment and Scale Cube, 486–487, 494
 - scalability and Big Data splits, 438–440, 445
 - when to use database splits, 393, 394

Yavonditte, Mike, 91

Z**Z-axis**

- AKF Application Scale Cube, 357–358, 359, 361–362, 363–364, 365–367, 371–373
- AKF Database Scale Cube, 383–386, 387, 388, 394
- AKF Scale Cube, 349–350, 351, 352, 354
 - cloud environment and Scale Cube, 486–487, 494
 - scalability and Big Data splits, 438–440, 445
 - when to use database splits, 393, 394