O'REILLY®

Site Reliability Engineering

HOW GOOGLE RUNS PRODUCTION SYSTEMS

Edited by Betsy Beyer, Chris Jones,
Jennifer Petoff & Niall Richard Murphy

Praise for *Site Reliability Engineering*

Google's SREs have done our industry an enormous service by writing up the principles, practices and patterns—architectural and cultural—that enable their teams to combine continuous delivery with world-class reliability at ludicrous scale. You owe it to yourself and your organization to read this book and try out these ideas for yourself.

Jez Humble, coauthor of Continuous Delivery and Lean Enterprise

I remember when Google first started speaking at systems administration conferences. It was like hearing a talk at a reptile show by a Gila monster expert. Sure, it was entertaining to hear about a very different world, but in the end the audience would go back to their geckos.

Now we live in a changed universe where the operational practices of Google are not so removed from those who work on a smaller scale. All of a sudden, the best practices of SRE that have been honed over the years are now of keen interest to the rest of us. For those of us facing challenges around scale, reliability and operations, this book comes none too soon.

David N. Blank-Edelman, Director, USENIX Board of Directors, and founding co-organizer of SREcon

I have been waiting for this book ever since I left Google's enchanted castle.
It is the gospel I am preaching to my peers at work.

Björn Rabenstein, Team Lead of Production Engineering at SoundCloud, Prometheus developer, and Google SRE until 2013

A thorough discussion of Site Reliability Engineering from the company that invented the concept. Includes not only the technical details but also the thought process, goals, principles, and lessons learned over time. If you want to learn what SRE really means, start here.

Russ Allbery, SRE and Security Engineer

With this book, Google employees have shared the processes they have taken, including the missteps, that have allowed Google services to expand to both massive scale and great reliability. I highly recommend that anyone who wants to create a set of integrated services that they hope will scale to read this book. The book provides an insider's guide to building maintainable services.

Rik Farrow, USENIX

Writing large-scale services like Gmail is hard. Running them with high reliability is even harder, especially when you change them every day. This comprehensive "recipe book" shows how Google does it, and you'll find it much cheaper to learn from our mistakes than to make them yourself.

Urs Hölzle, SVP Technical Infrastructure, Google

Site Reliability Engineering

How Google Runs Production Systems

Edited by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Site Reliability Engineering

Edited by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safaribooksonline.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editor: Brian Anderson
- Production Editor: Kristen Brown
- Copyeditor: Kim Cofer
- Proofreader: Rachel Monaghan
- Indexer: Judy McConville
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- April 2016: First Edition

Revision History for the First Edition

- 2016-03-21: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491929124 for release details.

978-1-491-92912-4

[LSI]

Foreword

Google's story is a story of scaling up. It is one of the great success stories of the computing industry marking a shift towards IT-centric business. Google was one of the first companies to define what business-IT alignment meant in practice, and went on to inform the concept of DevOps for a wider IT community. This book has been written by a broad cross-section of the very people who made that transition a reality.

Google grew at a time when the traditional role of the system administrator was being transformed. I questioned system administration, as if to say: we can't afford to hold tradition as an authority, we have to think anew, and we don't have time to wait for everyone else to catch up. In the introduction *Principles of Network and System Administration* [Bur99], I claimed that system administration was form of human-computer engineering. This was strongly rejected by some reviewers, who said "we are not yet at the stage where we can call it engineering." At the time, I felt that the field had become lost, trapped in its own wizard culture, and could not see a way forward. Then, Google drew a line in the silicon, forcing that fate into being. The revised role was called SRE, or Site Reliability Engineer Some of my friends were among the first of this new generation of engineer; they formalized it using software and automation. Initially, they were fiercely secretive, and what happened inside and outsid of Google was very different: Google's experience was unique. Over time, information and methods have flowed in both directions. This book shows a willingness to let SRE thinking come out of the shadows.

Here, we see not only how Google built its legendary infrastructure, but also how it studied, learned, and changed its mind about the tools and the technologies along the way. We, too, can face up to daunting challenges with an open spirit. The tribal nature of IT culture often entrenches practitioners in dogmatic positions that hold the industry back. If Google overcame this inertia, so can we.

This book is a collection of essays by one company, with a single common vision. The fact that the contributions are aligned around a single company's goal is what makes it special. There are commo themes, and common characters (software systems) that reappear in several chapters. We see choices from different perspectives, and know that they correlate to resolve competing interests. The articles are not rigorous, academic pieces; they are personal accounts, written with pride, in a variety of personal styles, and from the perspective of individual skill sets. They are written bravely, and with a intellectual honesty that is refreshing and uncommon in industry literature. Some claim "never do this, always do that," others are more philosophical and tentative, reflecting the variety of personalities within an IT culture, and how that too plays a role in the story. We, in turn, read them with the humility of observers who were not part of the journey, and do not have all the information about the myriad conflicting challenges. Our many questions are the real legacy of the volume: Why didn't they do *X*? What if they'd done *Y*? How will we look back on this in years to come? It is by comparing our own ideas to the reasoning here that we can measure our own thoughts and experience

The most impressive thing of all about this book is its very existence. Today, we hear a brazen cultur of "just show me the code." A culture of "ask no questions" has grown up around open source, where community rather than expertise is championed. Google is a company that dared to think about the problems from first principles, and to employ top talent with a high proportion of PhDs. Tools were only components in processes, working alongside chains of software, people, and data. Nothing here

tells us how to solve problems universally, but that is the point. Stories like these are far more valuable than the code or designs they resulted in. Implementations are ephemeral, but the documented reasoning is priceless. Rarely do we have access to this kind of insight.

This, then, is the story of how one company did it. The fact that it is many overlapping stories shows us that scaling is far more than just a photographic enlargement of a textbook computer architecture. It is about scaling a business process, rather than just the machinery. This lesson alone is worth its weight in electronic paper.

We do not engage much in self-critical review in the IT world; as such, there is much reinvention and repetition. For many years, there was only the USENIX LISA conference community discussing IT infrastructure, plus a few conferences about operating systems. It is very different today, yet this book still feels like a rare offering: a detailed documentation of Google's step through a watershed epoch. The tale is not for copying—though perhaps for emulating—but it can inspire the next step for all of us. There is a unique intellectual honesty in these pages, expressing both leadership and humility. These are stories of hopes, fears, successes, and failures. I salute the courage of authors and editors in allowing such candor, so that we, who are not party to the hands-on experiences, can also benefit from the lessons learned inside the cocoon.

Mark Burgess
***author of* In Search of Certainty**
Oslo, March 2016

Preface

Software engineering has this in common with having children: the labor *before* the birth is painful and difficult, but the labor *after* the birth is where you actually spend most of your effort. Yet softwa engineering as a discipline spends much more time talking about the first period as opposed to the second, despite estimates that 40–90% of the total costs of a system are incurred after birth.[1] The popular industry model that conceives of deployed, operational software as being "stabilized" in production, and therefore needing much less attention from software engineers, is wrong. Through th lens, then, we see that if software engineering tends to focus on designing and building software systems, there must be another discipline that focuses on the *whole* lifecycle of software objects, fro inception, through deployment and operation, refinement, and eventual peaceful decommissioning. This discipline uses—and needs to use—a wide range of skills, but has separate concerns from other kinds of engineers. Today, our answer is the discipline Google calls Site Reliability Engineering.

So what exactly is Site Reliability Engineering (SRE)? We admit that it's not a particularly clear name for what we do—pretty much every site reliability engineer at Google gets asked what exactly that is, and what they actually do, on a regular basis.

Unpacking the term a little, first and foremost, SREs are *engineers*. We apply the principles of computer science and engineering to the design and development of computing systems: generally, large distributed ones. Sometimes, our task is writing the software for those systems alongside our product development counterparts; sometimes, our task is building all the additional pieces those systems need, like backups or load balancing, ideally so they can be reused across systems; and sometimes, our task is figuring out how to apply existing solutions to new problems.

Next, we focus on system *reliability*. Ben Treynor Sloss, Google's VP for 24/7 Operations, originato of the term SRE, claims that reliability is the most fundamental feature of any product: a system isn' very useful if nobody can use it! Because reliability[2] is so critical, SREs are focused on finding ways to improve the design and operation of systems to make them more scalable, more reliable, and more efficient. However, we expend effort in this direction only up to a point: when systems are "reliable enough," we instead invest our efforts in adding features or building new products.[3]

Finally, SREs are focused on operating *services* built atop our distributed computing systems, whethe those services are planet-scale storage, email for hundreds of millions of users, or where Google began, web search. The "site" in our name originally referred to SRE's role in keeping the *google.co* website running, though we now run many more services, many of which aren't themselves websites —from internal infrastructure such as Bigtable to products for external developers such as the Googl Cloud Platform.

Although we have represented SRE as a broad discipline, it is no surprise that it arose in the fast-moving world of web services, and perhaps in origin owes something to the peculiarities of our infrastructure. It is equally no surprise that of all the post-deployment characteristics of software tha we could choose to devote special attention to, reliability is the one we regard as primary.[4] The domain of web services, both because the process of improving and changing server-side software is

comparatively contained, and because managing change itself is so tightly coupled with failures of al kinds, is a natural platform from which our approach might emerge.

Despite arising at Google, and in the web community more generally, we think that this discipline ha lessons applicable to other communities and other organizations. This book is an attempt to explain how we do things: both so that other organizations might make use of what we've learned, and so tha we can better define the role and what the term means. To that end, we have organized the book so th general principles and more specific practices are separated where possible, and where it's appropria to discuss a particular topic with Google-specific information, we trust that the reader will indulge us in this and will not be afraid to draw useful conclusions about their own environment.

We have also provided some orienting material—a description of Google's production environment and a mapping between some of our internal software and publicly available software—which should help to contextualize what we are saying and make it more directly usable.

Ultimately, of course, more reliability-oriented software and systems engineering is inherently good. However, we acknowledge that smaller organizations may be wondering how they can best use the experience represented here: much like security, the earlier you care about reliability, the better. This implies that even though a small organization has many pressing concerns and the software choices you make may differ from those Google made, it's still worth putting lightweight reliability support place early on, because it's less costly to expand a structure later on than it is to introduce one that is not present. Part IV contains a number of best practices for training, communication, and meetings that we've found to work well for us, many of which should be immediately usable by your organization.

But for sizes between a startup and a multinational, there probably already is someone in your organization who is doing SRE work, without it necessarily being called that name, or recognized as such. Another way to get started on the path to improving reliability for your organization is to formally recognize that work, or to find these people and foster what they do—reward it. They are people who stand on the cusp between one way of looking at the world and another one: like Newton, who is sometimes called not the world's first physicist, but the world's last alchemist.

And taking the historical view, who, then, looking back, might be the first SRE?

We like to think that Margaret Hamilton, working on the Apollo program on loan from MIT, had all the significant traits of the first SRE.[5] In her own words, "part of the culture was to learn from everyone and everything, including from that which one would least expect."

A case in point was when her young daughter Lauren came to work with her one day, while some of the team were running mission scenarios on the hybrid simulation computer. As young children do, Lauren went exploring, and she caused a "mission" to crash by selecting the DSKY keys in an unexpected way, alerting the team as to what would happen if the prelaunch program, P01, were inadvertently selected by a real astronaut during a real mission, during real midcourse. (Launching P01 inadvertently on a real mission would be a major problem, because it wipes out navigation data, and the computer was not equipped to pilot the craft with no navigation data.)

With an SRE's instincts, Margaret submitted a program change request to add special error checking code in the onboard flight software in case an astronaut should, by accident, happen to select P01

during flight. But this move was considered unnecessary by the "higher-ups" at NASA: of course, tha could never happen! So instead of adding error checking code, Margaret updated the mission specifications documentation to say the equivalent of "Do not select P01 during flight." (Apparently the update was amusing to many on the project, who had been told many times that astronauts would not make any mistakes—after all, they were trained to be perfect.)

Well, Margaret's suggested safeguard was only considered unnecessary until the very next mission, Apollo 8, just days after the specifications update. During midcourse on the fourth day of flight with the astronauts Jim Lovell, William Anders, and Frank Borman on board, Jim Lovell selected P01 by mistake—as it happens, on Christmas Day—creating much havoc for all involved. This was a critical problem, because in the absence of a workaround, no navigation data meant the astronauts were neve coming home. Thankfully, the documentation update had explicitly called this possibility out, and wa invaluable in figuring out how to upload usable data and recover the mission, with not much time to spare.

As Margaret says, "a thorough understanding of how to operate the systems was not enough to preve human errors," and the change request to add error detection and recovery software to the prelaunch program P01 was approved shortly afterwards.

Although the Apollo 8 incident occurred decades ago, there is much in the preceding paragraphs directly relevant to engineers' lives today, and much that will continue to be directly relevant in the future. Accordingly, for the systems you look after, for the groups you work in, or for the organizations you're building, please bear the SRE Way in mind: thoroughness and dedication, belief in the value of preparation and documentation, and an awareness of what could go wrong, coupled with a strong desire to prevent it. Welcome to our emerging profession!

HOW TO READ THIS BOOK

This book is a series of essays written by members and alumni of Google's Site Reliability Engineering organization. It's much more like conference proceedings than it is like a standard book by an author or a small number of authors. Each chapter is intended to be read as a part of a coherent whole, but a good deal can be gained by reading on whatever subject particularly interests you. (If there are other articles that support or inform the text, we reference them so you can follow up accordingly.)

You don't need to read in any particular order, though we'd suggest at least starting with Chapters 2 and 3, which describe Google's production environment and outline how SRE approaches risk, respectively. (Risk is, in many ways, the key quality of our profession.) Reading cover-to-cover is, of course, also useful and possible; our chapters are grouped thematically, into Principles (Part II), Practices (Part III), and Management (Part IV). Each has a small introduction that highlights what the individual pieces are about, and references other articles published by Google SREs, covering specific topics in more detail. Additionally, the companion website to this book, *https://g.co/SREBook*, has a number of helpful resources.

We hope this will be at least as useful and interesting to you as putting it together was for us.

— The Editors

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

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

`Constant width`

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

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material is available at *https://g.co/SREBook*.

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Site Reliability Engineering*, edited by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (O'Reilly). Copyright 2016 Google, Inc., 978-1-491-92912-4."

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

Safari® Books Online

NOTE

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/site-reliability-engineering*.

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

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

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

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

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

Acknowledgments

This book would not have been possible without the tireless efforts of our authors and technical writers. We'd also like thank the following internal reviewers for providing especially valuable

feedback: Alex Matey, Dermot Duffy, JC van Winkel, John T. Reese, Michael O'Reilly, Steve Carstensen, and Todd Underwood. Ben Lutch and Ben Treynor Sloss were this book's sponsors within Google; their belief in this project and sharing what we've learned about running large-scale services was essential to making this book happen.

We'd like to send special thanks to Rik Farrow, the editor of *;login:*, for partnering with us on a number of contributions for pre-publication via USENIX.

While the authors are specifically acknowledged in each chapter, we'd like to take time to recognize those that contributed to each chapter by providing thoughtful input, discussion, and review.

Chapter 3: Abe Rahey, Ben Treynor Sloss, Brian Stoler, Dave O'Connor, David Besbris, Jill Alvidrez, Mike Curtis, Nancy Chang, Tammy Capistrant, Tom Limoncelli

Chapter 5: Cody Smith, George Sadlier, Laurence Berland, Marc Alvidrez, Patrick Stahlberg, Peter Duff, Pim van Pelt, Ryan Anderson, Sabrina Farmer, Seth Hettich

Chapter 6: Mike Curtis, Jamie Wilkinson, Seth Hettich

Chapter 8: David Schnur, JT Goldstone, Marc Alvidrez, Marcus Lara-Reinhold, Noah Maxwell, Peter Dinges, Sumitran Raghunathan, Yutong Cho

Chapter 9: Ryan Anderson

Chapter 10: Jules Anderson, Max Luebbe, Mikel Mcdaniel, Raul Vera, Seth Hettich

Chapter 11: Andrew Stribblehill, Richard Woodbury

Chapter 12: Charles Stephen Gunn, John Hedditch, Peter Nuttall, Rob Ewaschuk, Sam Greenfield

Chapter 13: Jelena Oertel, Kripa Krishnan, Sergio Salvi, Tim Craig

Chapter 14: Amy Zhou, Carla Geisser, Grainne Sheerin, Hildo Biersma, Jelena Oertel, Perry Lorier, Rune Kristian Viken

Chapter 15: Dan Wu, Heather Sherman, Jared Brick, Mike Louer, Štěpán Davidovič, Tim Craig

Chapter 16: Andrew Stribblehill, Richard Woodbury

Chapter 17: Isaac Clerencia, Marc Alvidrez

Chapter 18: Ulric Longyear

Chapter 19: Debashish Chatterjee, Perry Lorier

Chapters 20 and 21: Adam Fletcher, Christoph Pfisterer, Lukáš Ježek, Manjot Pahwa, Micha Riser, Noah Fiedel, Pavel Herrmann, Paweł Zuzelski, Perry Lorier, Ralf Wildenhues, Tudor-Ioan Salomie, Witold Baryluk

Chapter 22: Mike Curtis, Ryan Anderson

Chapter 23: Ananth Shrinivas, Mike Burrows

Chapter 24: Ben Fried, Derek Jackson, Gabe Krabbe, Laura Nolan, Seth Hettich

Chapter 25: Abdulrahman Salem, Alex Perry, Arnar Mar Hrafnkelsson, Dieter Pearcey, Dylan Curley, Eivind Eklund, Eric Veach, Graham Poulter, Ingvar Mattsson, John Looney, Ken Grant, Michelle Duffy, Mike Hochberg, Will Robinson

Chapter 26: Corey Vickrey, Dan Ardelean, Disney Luangsisongkham, Gordon Prioreschi, Kristina Bennett, Liang Lin, Michael Kelly, Sergey Ivanyuk

Chapter 27: Vivek Rau

Chapter 28: Melissa Binde, Perry Lorier, Preston Yoshioka

Chapter 29: Ben Lutch, Carla Geisser, Dzevad Trumic, John Turek, Matt Brown

Chapter 30: Charles Stephen Gunn, Chris Heiser, Max Luebbe, Sam Greenfield

Chapter 31: Alex Kehlenbeck, Jeromy Carriere, Joel Becker, Sowmya Vijayaraghavan, Trevor Mattson-Hamilton

Chapter 32: Seth Hettich

Chapter 33: Adrian Hilton, Brad Kratochvil, Charles Ballowe, Dan Sheridan, Eddie Kennedy, Erik Gross, Gus Hartmann, Jackson Stone, Jeff Stevenson, John Li, Kevin Greer, Matt Toia, Michael Haynie, Mike Doherty, Peter Dahl, Ron Heiby

We are also grateful to the following contributors, who either provided significant material, did an excellent job of reviewing, agreed to be interviewed, supplied significant expertise or resources, or had some otherwise excellent effect on this work:

Abe Hassan, Adam Rogoyski, Alex Hidalgo, Amaya Booker, Andrew Fikes, Andrew Hurst, Ariel Gol Ashleigh Rentz, Ayman Hourieh, Barclay Osborn, Ben Appleton, Ben Love, Ben Winslow, Bernhard Beck, Bill Duane, Bill Patry, Blair Zajac, Bob Gruber, Brian Gustafson, Bruce Murphy, Buck Clay, Cedric Cellier, Chiho Saito, Chris Carlon, Christopher Hahn, Chris Kennelly, Chris Taylor, Ciara Kamahele-Sanfratello, Colin Phipps, Colm Buckley, Craig Paterson, Daniel Eisenbud, Daniel V. Klein, Daniel Spoonhower, Dan Watson, Dave Phillips, David Hixson, Dina Betser, Doron Meyer, Dmitry Fedoruk, Eric Grosse, Eric Schrock, Filip Zyzniewski, Francis Tang, Gary Arneson, Georgina Wilcox, Gretta Bartels, Gustavo Franco, Harald Wagener, Healfdene Goguen, Hugo Santos, Hyrum Wright, Ian Gulliver, Jakub Turski, James Chivers, James O'Kane, James Youngman, Jan Monsch, Jason Parker-Burlingham, Jason Petsod, Jeffry McNeil, Jeff Dean, Jeff Peck, Jennifer Mace, Jerry Cen, Jess Frame, John Brady, John Gunderman, John Kochmar, John Tobin, Jordyn Buchanan, Josepl Bironas, Julio Merino, Julius Plenz, Kate Ward, Kathy Polizzi, Katrina Sostek, Kenn Hamm, Kirk Russell, Kripa Krishnan, Larry Greenfield, Lea Oliveira, Luca Cittadini, Lucas Pereira, Magnus Ringman, Mahesh Palekar, Marco Paganini, Mario Bonilla, Mathew Mills, Mathew Monroe, Matt D. Brown, Matt Proud, Max Saltonstall, Michal Jaszczyk, Mihai Bivol, Misha Brukman, Olivier Oansaldi, Patrick Bernier, Pierre Palatin, Rob Shanley, Robert van Gent, Rory Ward, Rui Zhang-Sher Salim Virji, Sanjay Ghemawat, Sarah Coty, Sean Dorward, Sean Quinlan, Sean Sechrest, Shari Trumbo-McHenry, Shawn Morrissey, Shun-Tak Leung, Stan Jedrus, Stefano Lattarini, Steven Schirripa, Tanya Reilly, Terry Bolt, Tim Chaplin, Toby Weingartner, Tom Black, Udi Meiri, Victor Terron, Vlad Grama, Wes Hertlein, and Zoltan Egyed.

We very much appreciate the thoughtful and in-depth feedback that we received from external reviewers: Andrew Fong, Björn Rabenstein, Charles Border, David Blank-Edelman, Frossie Economou, James Meickle, Josh Ryder, Mark Burgess, and Russ Allbery.

We would like to extend special thanks to Cian Synnott, original book team member and co-conspirator, who left Google before this project was completed but was deeply influential to it, and

Margaret Hamilton, who so graciously allowed us to reference her story in our preface. Additionally, we would like to extend special thanks to Shylaja Nukala, who generously gave of the time of her technical writers and supported their necessary and valued efforts wholeheartedly.

The editors would also like to personally thank the following people:

Betsy Beyer: To Grandmother (my personal hero), for supplying endless amounts of phone pep talks and popcorn, and to Riba, for supplying me with the sweatpants necessary to fuel several late nights. These, of course, in addition to the cast of SRE all-stars who were indeed delightful collaborators.

Chris Jones: To Michelle, for saving me from a life of crime on the high seas and for her uncanny ability to find manzanas in unexpected places, and to those who've taught me about engineering over the years.

Jennifer Petoff: To my husband Scott for being incredibly supportive during the two year process of writing this book and for keeping the editors supplied with plenty of sugar on our "Dessert Island."

Niall Murphy: To Léan, Oisín, and Fiachra, who were considerably more patient than I had any right to expect with a substantially rantier father and husband than usual, for years. To Dermot, for the transfer offer.

---

[1] The very fact that there is such large variance in these estimates tells you something about software engineering as a discipline, but see, e.g., [Gla02] for more details.

[2] For our purposes, reliability is "The probability that [a system] will perform a required function without failure under stated conditions for a stated period of time," following the definition in [Oco12].

[3] The software systems we're concerned with are largely websites and similar services; we do not discuss the reliability concerns that face software intended for nuclear power plants, aircraft, medical equipment, or other safety-critical systems. We do, however, compare our approaches with those used in other industries in Chapter 33.

[4] In this, we are distinct from the industry term DevOps, because although we definitely regard infrastructure as code, we have *reliability* as our main focus. Additionally, we are strongly oriented toward removing the necessity for operations—see Chapter for more details.

[5] In addition to this great story, she also has a substantial claim to popularizing the term "software engineering."

Part I. Introduction

This section provides some high-level guidance on what SRE is and why it is different from more conventional IT industry practices.

Ben Treynor Sloss, the senior VP overseeing technical operations at Google—and the originator of th term “Site Reliability Engineering”—provides his view on what SRE means, how it works, and how i compares to other ways of doing things in the industry, in Chapter 1.

We provide a guide to the production environment at Google in Chapter 2 as a way to help acquaint you with the wealth of new terms and systems you are about to meet in the rest of the book.

Chapter 1. Introduction

Written by Benjamin Treynor Sloss[1]
Edited by Betsy Beyer

Hope is not a strategy.

Traditional SRE saying

It is a truth universally acknowledged that systems do not run themselves. How, then, *should* a syste —particularly a complex computing system that operates at a large scale—be run?

The Sysadmin Approach to Service Management

Historically, companies have employed systems administrators to run complex computing systems.

This systems administrator, or sysadmin, approach involves assembling existing software componen and deploying them to work together to produce a service. Sysadmins are then tasked with running th service and responding to events and updates as they occur. As the system grows in complexity and traffic volume, generating a corresponding increase in events and updates, the sysadmin team grows absorb the additional work. Because the sysadmin role requires a markedly different skill set than tha required of a product's developers, developers and sysadmins are divided into discrete teams: "development" and "operations" or "ops."

The sysadmin model of service management has several advantages. For companies deciding how to run and staff a service, this approach is relatively easy to implement: as a familiar industry paradigm there are many examples from which to learn and emulate. A relevant talent pool is already widely available. An array of existing tools, software components (off the shelf or otherwise), and integratio companies are available to help run those assembled systems, so a novice sysadmin team doesn't hav to reinvent the wheel and design a system from scratch.

The sysadmin approach and the accompanying development/ops split has a number of disadvantages and pitfalls. These fall broadly into two categories: direct costs and indirect costs.

Direct costs are neither subtle nor ambiguous. Running a service with a team that relies on manual intervention for both change management and event handling becomes expensive as the service and/o traffic to the service grows, because the size of the team necessarily scales with the load generated by the system.

The indirect costs of the development/ops split can be subtle, but are often more expensive to the organization than the direct costs. These costs arise from the fact that the two teams are quite differe in background, skill set, and incentives. They use different vocabulary to describe situations; they carry different assumptions about both risk and possibilities for technical solutions; they have different assumptions about the target level of product stability. The split between the groups can easily become one of not just incentives, but also communication, goals, and eventually, trust and respect. This outcome is a pathology.

Traditional operations teams and their counterparts in product development thus often end up in conflict, most visibly over how quickly software can be released to production. At their core, the development teams want to launch new features and see them adopted by users. At *their* core, the ops teams want to make sure the service doesn't break while they are holding the pager. Because most outages are caused by some kind of change—a new configuration, a new feature launch, or a new typ of user traffic—the two teams' goals are fundamentally in tension.

Both groups understand that it is unacceptable to state their interests in the baldest possible terms ("We want to launch anything, any time, without hindrance" versus "We won't want to ever change anything in the system once it works"). And because their vocabulary and risk assumptions differ, both groups often resort to a familiar form of trench warfare to advance their interests. The ops team attempts to safeguard the running system against the risk of change by introducing launch and chang gates. For example, launch reviews may contain an explicit check for *every* problem that has *ever* caused an outage in the past—that could be an arbitrarily long list, with not all elements providing equal value. The dev team quickly learns how to respond. They have fewer "launches" and more "fla flips," "incremental updates," or "cherrypicks." They adopt tactics such as sharding the product so that fewer features are subject to the launch review.

Google's Approach to Service Management: Site Reliability Engineering

Conflict isn't an inevitable part of offering a software service. Google has chosen to run our systems with a different approach: our Site Reliability Engineering teams focus on hiring software engineers run our products and to create systems to accomplish the work that would otherwise be performed, often manually, by sysadmins.

What exactly is Site Reliability Engineering, as it has come to be defined at Google? My explanation is simple: SRE is what happens when you ask a software engineer to design an operations team. Whe I joined Google in 2003 and was tasked with running a "Production Team" of seven engineers, my entire life up to that point had been software engineering. So I designed and managed the group the way *I* would want it to work if I worked as an SRE myself. That group has since matured to become Google's present-day SRE team, which remains true to its origins as envisioned by a lifelong softwar engineer.

A primary building block of Google's approach to service management is the composition of each SRE team. As a whole, SRE can be broken down two main categories.

50–60% are Google Software Engineers, or more precisely, people who have been hired via the standard procedure for Google Software Engineers. The other 40–50% are candidates who were very close to the Google Software Engineering qualifications (i.e., 85–99% of the skill set required), and who *in addition* had a set of technical skills that is useful to SRE but is rare for most software engineers. By far, UNIX system internals and networking (Layer 1 to Layer 3) expertise are the two most common types of alternate technical skills we seek.

Common to all SREs is the belief in and aptitude for developing software systems to solve complex problems. Within SRE, we track the career progress of both groups closely, and have to date found no

practical difference in performance between engineers from the two tracks. In fact, the somewhat diverse background of the SRE team frequently results in clever, high-quality systems that are clearly the product of the synthesis of several skill sets.

The result of our approach to hiring for SRE is that we end up with a team of people who (a) will quickly become bored by performing tasks by hand, and (b) have the skill set necessary to write software to replace their previously manual work, even when the solution is complicated. SREs also end up sharing academic and intellectual background with the rest of the development organization. Therefore, SRE is fundamentally doing work that has historically been done by an operations team, but using engineers with software expertise, and banking on the fact that these engineers are inherently both predisposed to, and have the ability to, design and implement automation with software to replace human labor.

By design, it is crucial that SRE teams are focused on engineering. Without constant engineering, operations load increases and teams will need more people just to keep pace with the workload. Eventually, a traditional ops-focused group scales linearly with service size: if the products supported by the service succeed, the operational load will grow with traffic. That means hiring more people to do the same tasks over and over again.

To avoid this fate, the team tasked with managing a service needs to code or it will drown. Therefore, Google places *a 50% cap on the aggregate "ops" work for all SREs*—tickets, on-call, manual tasks, etc. This cap ensures that the SRE team has enough time in their schedule to make the service stable and operable. This cap is an upper bound; over time, left to their own devices, the SRE team should end up with very little operational load and almost entirely engage in development tasks, because the service basically runs and repairs itself: we want systems that are *automatic*, not just *automated*. In practice, scale and new features keep SREs on their toes.

Google's rule of thumb is that an SRE team must spend the remaining 50% of its time actually doing development. So how do we enforce that threshold? In the first place, we have to measure how SRE time is spent. With that measurement in hand, we ensure that the teams consistently spending less than 50% of their time on development work change their practices. Often this means shifting some of the operations burden back to the development team, or adding staff to the team without assigning the team additional operational responsibilities. Consciously maintaining this balance between ops and development work allows us to ensure that SREs have the bandwidth to engage in creative, autonomous engineering, while still retaining the wisdom gleaned from the operations side of running a service.

We've found that Google SRE's approach to running large-scale systems has many advantages. Because SREs are directly modifying code in their pursuit of making Google's systems run themselves, SRE teams are characterized by both rapid innovation and a large acceptance of change. Such teams are relatively inexpensive—supporting the same service with an ops-oriented team would require a significantly larger number of people. Instead, the number of SREs needed to run, maintain, and improve a system scales sublinearly with the size of the system. Finally, not only does SRE circumvent the dysfunctionality of the dev/ops split, but this structure also improves our product development teams: easy transfers between product development and SRE teams cross-train the entire group, and improve skills of developers who otherwise may have difficulty learning how to build a

million-core distributed system.

Despite these net gains, the SRE model is characterized by its own distinct set of challenges. One continual challenge Google faces is hiring SREs: not only does SRE compete for the same candidates as the product development hiring pipeline, but the fact that we set the hiring bar so high in terms of both coding and system engineering skills means that our hiring pool is necessarily small. As our discipline is relatively new and unique, not much industry information exists on how to build and manage an SRE team (although hopefully this book will make strides in that direction!). And once an SRE team is in place, their potentially unorthodox approaches to service management require strong management support. For example, the decision to stop releases for the remainder of the quarter once an error budget is depleted might not be embraced by a product development team unless mandated by their management.

DEVOPS OR SRE?

The term "DevOps" emerged in industry in late 2008 and as of this writing (early 2016) is still in a state of flux. Its core principles—involvement of the IT function in each phase of a system's design and development, heavy reliance on automation versus human effort, the application of engineering practices and tools to operations tasks—are consistent with many of SRE's principles and practices. One could view DevOps as a generalization of several core SRE principles to a wider range of organizations, management structures, and personnel. One could equivalently view SRE as a specific implementation of DevOps with some idiosyncratic extensions.

Tenets of SRE

While the nuances of workflows, priorities, and day-to-day operations vary from SRE team to SRE team, all share a set of basic responsibilities for the service(s) they support, and adhere to the same core tenets. In general, an SRE team is responsible for the *availability*, *latency*, *performance*, *efficiency*, *change management*, *monitoring*, *emergency response*, *and capacity planning* of their service(s). We have codified rules of engagement and principles for how SRE teams interact with the environment—not only the production environment, but also the product development teams, the testing teams, the users, and so on. Those rules and work practices help us to maintain our focus on engineering work, as opposed to operations work.

The following section discusses each of the core tenets of Google SRE.

Ensuring a Durable Focus on Engineering

As already discussed, Google caps operational work for SREs at 50% of their time. Their remaining time should be spent using their coding skills on project work. In practice, this is accomplished by monitoring the amount of operational work being done by SREs, and redirecting excess operational work to the product development teams: reassigning bugs and tickets to development managers, [re]integrating developers into on-call pager rotations, and so on. The redirection ends when the operational load drops back to 50% or lower. This also provides an effective feedback mechanism, guiding developers to build systems that don't need manual intervention. This approach works well when the entire organization—SRE and development alike—understands why the safety valve mechanism exists, and supports the goal of having no overflow events because the product doesn't

generate enough operational load to require it.

When they are focused on operations work, on average, SREs should receive a maximum of two events per 8–12-hour on-call shift. This target volume gives the on-call engineer enough time to handle the event accurately and quickly, clean up and restore normal service, and then conduct a postmortem. If more than two events occur regularly per on-call shift, problems can't be investigated thoroughly and engineers are sufficiently overwhelmed to prevent them from learning from these events. A scenario of pager fatigue also won't improve with scale. Conversely, if on-call SREs consistently receive fewer than one event per shift, keeping them on point is a waste of their time.

Postmortems should be written for all significant incidents, regardless of whether or not they paged; postmortems that did not trigger a page are even more valuable, as they likely point to clear monitoring gaps. This investigation should establish what happened in detail, find all root causes of the event, and assign actions to correct the problem or improve how it is addressed next time. Google operates under a *blame-free postmortem culture*, with the goal of exposing faults and applying engineering to fix these faults, rather than avoiding or minimizing them.

Pursuing Maximum Change Velocity Without Violating a Service's SLO

Product development and SRE teams can enjoy a productive working relationship by eliminating the structural conflict in their respective goals. The structural conflict is between pace of innovation and product stability, and as described earlier, this conflict often is expressed indirectly. In SRE we bring this conflict to the fore, and then resolve it with the introduction of an *error budget*.

The error budget stems from the observation that *100% is the wrong reliability target for basically everything* (pacemakers and anti-lock brakes being notable exceptions). In general, for any software service or system, 100% is not the right reliability target because no user can tell the difference between a system being 100% available and 99.999% available. There are many other systems in the path between user and service (their laptop, their home WiFi, their ISP, the power grid…) and those systems collectively are far less than 99.999% available. Thus, the marginal difference between 99.999% and 100% gets lost in the noise of other unavailability, and the user receives no benefit from the enormous effort required to add that last 0.001% of availability.

If 100% is the wrong reliability target for a system, what, then, is the right reliability target for the system? This actually isn't a technical question at all—it's a product question, which should take the following considerations into account:

- What level of availability will the users be happy with, given how they use the product?
- What alternatives are available to users who are dissatisfied with the product's availability?
- What happens to users' usage of the product at different availability levels?

The business or the product must establish the system's availability target. Once that target is established, the error budget is one minus the availability target. A service that's 99.99% available is 0.01% unavailable. That permitted 0.01% unavailability is the service's *error budget*. We can spend the budget on anything we want, as long as we don't overspend it.

So how do we want to spend the error budget? The development team wants to launch features and attract new users. Ideally, we would spend all of our error budget taking risks with things we launch i

order to launch them quickly. This basic premise describes the whole model of error budgets. As soo as SRE activities are conceptualized in this framework, freeing up the error budget through tactics such as phased rollouts and 1% experiments can optimize for quicker launches.

The use of an error budget resolves the structural conflict of incentives between development and SRE. SRE's goal is no longer "zero outages"; rather, SREs and product developers aim to spend the error budget getting maximum feature velocity. This change makes all the difference. An outage is n longer a "bad" thing—it is an expected part of the process of innovation, and an occurrence that both development and SRE teams manage rather than fear.

Monitoring

Monitoring is one of the primary means by which service owners keep track of a system's health and availability. As such, monitoring strategy should be constructed thoughtfully. A classic and common approach to monitoring is to watch for a specific value or condition, and then to trigger an email aler when that value is exceeded or that condition occurs. However, this type of email alerting is not an effective solution: a system that requires a human to read an email and decide whether or not some type of action needs to be taken in response is fundamentally flawed. Monitoring should never requir a human to interpret any part of the alerting domain. Instead, software should do the interpreting, and humans should be notified only when they need to take action.

There are three kinds of valid monitoring output:

Alerts

> Signify that a human needs to take action immediately in response to something that is either happening or about to happen, in order to improve the situation.

Tickets

> Signify that a human needs to take action, but not immediately. The system cannot automatically handle the situation, but if a human takes action in a few days, no damage will result.

Logging

> No one needs to look at this information, but it is recorded for diagnostic or forensic purposes. Th expectation is that no one reads logs unless something else prompts them to do so.

Emergency Response

Reliability is a function of mean time to failure (MTTF) and mean time to repair (MTTR) [Sch15]. The most relevant metric in evaluating the effectiveness of emergency response is how quickly the response team can bring the system back to health—that is, the MTTR.

Humans add latency. Even if a given system experiences more *actual* failures, a system that can avoi emergencies that require human intervention will have higher availability than a system that requires hands-on intervention. When humans are necessary, we have found that thinking through and recording the best practices ahead of time in a "playbook" produces roughly a 3x improvement in MTTR as compared to the strategy of "winging it." The hero jack-of-all-trades on-call engineer does work, but the practiced on-call engineer armed with a playbook works much better. While no

playbook, no matter how comprehensive it may be, is a substitute for smart engineers able to think on the fly, clear and thorough troubleshooting steps and tips are valuable when responding to a high-stakes or time-sensitive page. Thus, Google SRE relies on on-call playbooks, in addition to exercises such as the "Wheel of Misfortune,"[2] to prepare engineers to react to on-call events.

Change Management

SRE has found that roughly 70% of outages are due to changes in a live system. Best practices in this domain use automation to accomplish the following:

- Implementing progressive rollouts
- Quickly and accurately detecting problems
- Rolling back changes safely when problems arise

This trio of practices effectively minimizes the aggregate number of users and operations exposed to bad changes. By removing humans from the loop, these practices avoid the normal problems of fatigue, familiarity/contempt, and inattention to highly repetitive tasks. As a result, both release velocity and safety increase.

Demand Forecasting and Capacity Planning

Demand forecasting and capacity planning can be viewed as ensuring that there is sufficient capacity and redundancy to serve projected future demand with the required availability. There's nothing particularly special about these concepts, except that a surprising number of services and teams don't take the steps necessary to ensure that the required capacity is in place by the time it is needed. Capacity planning should take both organic growth (which stems from natural product adoption and usage by customers) and inorganic growth (which results from events like feature launches, marketing campaigns, or other business-driven changes) into account.

Several steps are mandatory in capacity planning:

- An accurate organic demand forecast, which extends beyond the lead time required for acquiring capacity
- An accurate incorporation of inorganic demand sources into the demand forecast
- Regular load testing of the system to correlate raw capacity (servers, disks, and so on) to service capacity

Because capacity is critical to availability, it naturally follows that the SRE team must be in charge of capacity planning, which means they also must be in charge of provisioning.

Provisioning

Provisioning combines both change management and capacity planning. In our experience, provisioning must be conducted quickly and only when necessary, as capacity is expensive. This exercise must also be done correctly or capacity doesn't work when needed. Adding new capacity often involves spinning up a new instance or location, making significant modification to existing

systems (configuration files, load balancers, networking), and validating that the new capacity performs and delivers correct results. Thus, it is a riskier operation than load shifting, which is often done multiple times per hour, and must be treated with a corresponding degree of extra caution.

Efficiency and Performance

Efficient use of resources is important any time a service cares about money. Because SRE ultimatel controls provisioning, it must also be involved in any work on utilization, as utilization is a function of how a given service works and how it is provisioned. It follows that paying close attention to the provisioning strategy for a service, and therefore its utilization, provides a very, very big lever on the service's total costs.

Resource use is a function of demand (load), capacity, and software efficiency. SREs predict demand provision capacity, and can modify the software. These three factors are a large part (though not the entirety) of a service's efficiency.

Software systems become slower as load is added to them. A slowdown in a service equates to a loss of capacity. At some point, a slowing system stops serving, which corresponds to infinite slowness. SREs provision to meet a capacity target *at a specific response speed*, and thus are keenly interested a service's performance. SREs and product developers will (and should) monitor and modify a servic to improve its performance, thus adding capacity and improving efficiency.[3]

The End of the Beginning

Site Reliability Engineering represents a significant break from existing industry best practices for managing large, complicated services. Motivated originally by familiarity—"as a software engineer, this is how I would want to invest my time to accomplish a set of repetitive tasks"—it has become much more: a set of principles, a set of practices, a set of incentives, and a field of endeavor within th larger software engineering discipline. The rest of the book explores the SRE Way in detail.

---

[1] Vice President, Google Engineering, founder of Google SRE

[2] See "Disaster Role Playing".

[3] For further discussion of how this collaboration can work in practice, see "Communications: Production Meetings".

Chapter 2. The Production Environment at Google, from the Viewpoint of an SRE

Written by JC van Winkel
Edited by Betsy Beyer

Google datacenters are very different from most conventional datacenters and small-scale server farms. These differences present both extra problems and opportunities. This chapter discusses the challenges and opportunities that characterize Google datacenters and introduces terminology that is used throughout the book.

Hardware

Most of Google's compute resources are in Google-designed datacenters with proprietary power distribution, cooling, networking, and compute hardware (see [Bar13]). Unlike "standard" colocation datacenters, the compute hardware in a Google-designed datacenter is the same across the board.[1] To eliminate the confusion between server hardware and server software, we use the following terminology throughout the book:

Machine

: A piece of hardware (or perhaps a VM)

Server

: A piece of software that implements a service

Machines can run any server, so we don't dedicate specific machines to specific server programs. There's no specific machine that runs our mail server, for example. Instead, resource allocation is handled by our cluster operating system, *Borg*.

We realize this use of the word *server* is unusual. The common use of the word conflates "binary that accepts network connection" with *machine*, but differentiating between the two is important when talking about computing at Google. Once you get used to our usage of *server*, it becomes more apparent why it makes sense to use this specialized terminology, not just within Google but also in the rest of this book.

Figure 2-1 illustrates the topology of a Google datacenter:

- Tens of machines are placed in a *rack*.
- Racks stand in a *row*.
- One or more rows form a *cluster*.
- Usually a *datacenter* building houses multiple clusters.
- Multiple datacenter buildings that are located close together form a *campus*.