

O'REILLY®

Free Sampler



Site Reliability Engineering

HOW GOOGLE RUNS PRODUCTION SYSTEMS

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

Site Reliability Engineering

How Google Runs Production Systems

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

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY[®]

Site Reliability Engineering

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

Copyright © 2016 Google, Inc. All rights reserved.

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.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Site Reliability Engineering*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-92912-4

[LSI]

Table of Contents

Foreword.....	xiii
Preface.....	xv

Part I. Introduction

1. Introduction.....	3
The Sysadmin Approach to Service Management	3
Google’s Approach to Service Management: Site Reliability Engineering	5
Tenets of SRE	7
The End of the Beginning	12
2. The Production Environment at Google, from the Viewpoint of an SRE.....	13
Hardware	13
System Software That “Organizes” the Hardware	15
Other System Software	18
Our Software Infrastructure	19
Our Development Environment	19
Shakespeare: A Sample Service	20

Part II. Principles

3. Embracing Risk.....	25
Managing Risk	25
Measuring Service Risk	26
Risk Tolerance of Services	28

Motivation for Error Budgets	33
4. Service Level Objectives	37
Service Level Terminology	37
Indicators in Practice	40
Objectives in Practice	43
Agreements in Practice	47
5. Eliminating Toil	49
Toil Defined	49
Why Less Toil Is Better	51
What Qualifies as Engineering?	52
Is Toil Always Bad?	52
Conclusion	54
6. Monitoring Distributed Systems	55
Definitions	55
Why Monitor?	56
Setting Reasonable Expectations for Monitoring	57
Symptoms Versus Causes	58
Black-Box Versus White-Box	59
The Four Golden Signals	60
Worrying About Your Tail (or, Instrumentation and Performance)	61
Choosing an Appropriate Resolution for Measurements	62
As Simple as Possible, No Simpler	62
Tying These Principles Together	63
Monitoring for the Long Term	64
Conclusion	66
7. The Evolution of Automation at Google	67
The Value of Automation	67
The Value for Google SRE	70
The Use Cases for Automation	70
Automate Yourself Out of a Job: Automate ALL the Things!	73
Soothing the Pain: Applying Automation to Cluster Turnups	75
Borg: Birth of the Warehouse-Scale Computer	81
Reliability Is the Fundamental Feature	83
Recommendations	84
8. Release Engineering	87
The Role of a Release Engineer	87
Philosophy	88

Continuous Build and Deployment	90
Configuration Management	93
Conclusions	95
9. Simplicity.....	97
System Stability Versus Agility	97
The Virtue of Boring	98
I Won't Give Up My Code!	98
The "Negative Lines of Code" Metric	99
Minimal APIs	99
Modularity	100
Release Simplicity	100
A Simple Conclusion	101

Part III. Practices

10. Practical Alerting from Time-Series Data.....	107
The Rise of Borgmon	108
Instrumentation of Applications	109
Collection of Exported Data	110
Storage in the Time-Series Arena	111
Rule Evaluation	114
Alerting	118
Sharding the Monitoring Topology	119
Black-Box Monitoring	120
Maintaining the Configuration	121
Ten Years On...	122
11. Being On-Call.....	125
Introduction	125
Life of an On-Call Engineer	126
Balanced On-Call	127
Feeling Safe	128
Avoiding Inappropriate Operational Load	130
Conclusions	132
12. Effective Troubleshooting.....	133
Theory	134
In Practice	136
Negative Results Are Magic	144
Case Study	146

Making Troubleshooting Easier	150
Conclusion	150
13. Emergency Response.....	151
What to Do When Systems Break	151
Test-Induced Emergency	152
Change-Induced Emergency	153
Process-Induced Emergency	155
All Problems Have Solutions	158
Learn from the Past. Don't Repeat It.	158
Conclusion	159
14. Managing Incidents.....	161
Unmanaged Incidents	161
The Anatomy of an Unmanaged Incident	162
Elements of Incident Management Process	163
A Managed Incident	165
When to Declare an Incident	166
In Summary	166
15. Postmortem Culture: Learning from Failure.....	169
Google's Postmortem Philosophy	169
Collaborate and Share Knowledge	171
Introducing a Postmortem Culture	172
Conclusion and Ongoing Improvements	175
16. Tracking Outages.....	177
Escalator	178
Outalator	178
17. Testing for Reliability.....	183
Types of Software Testing	185
Creating a Test and Build Environment	190
Testing at Scale	192
Conclusion	204
18. Software Engineering in SRE.....	205
Why Is Software Engineering Within SRE Important?	205
Auxon Case Study: Project Background and Problem Space	207
Intent-Based Capacity Planning	209
Fostering Software Engineering in SRE	218
Conclusions	222

19. Load Balancing at the Frontend.....	223
Power Isn't the Answer	223
Load Balancing Using DNS	224
Load Balancing at the Virtual IP Address	227
20. Load Balancing in the Datacenter.....	231
The Ideal Case	232
Identifying Bad Tasks: Flow Control and Lame Ducks	233
Limiting the Connections Pool with Subsetting	235
Load Balancing Policies	240
21. Handling Overload.....	247
The Pitfalls of "Queries per Second"	248
Per-Customer Limits	248
Client-Side Throttling	249
Criticality	251
Utilization Signals	253
Handling Overload Errors	253
Load from Connections	257
Conclusions	258
22. Addressing Cascading Failures.....	259
Causes of Cascading Failures and Designing to Avoid Them	260
Preventing Server Overload	265
Slow Startup and Cold Caching	274
Triggering Conditions for Cascading Failures	276
Testing for Cascading Failures	278
Immediate Steps to Address Cascading Failures	280
Closing Remarks	283
23. Managing Critical State: Distributed Consensus for Reliability.....	285
Motivating the Use of Consensus: Distributed Systems Coordination Failure	288
How Distributed Consensus Works	289
System Architecture Patterns for Distributed Consensus	291
Distributed Consensus Performance	296
Deploying Distributed Consensus-Based Systems	304
Monitoring Distributed Consensus Systems	312
Conclusion	313
24. Distributed Periodic Scheduling with Cron.....	315
Cron	315
Cron Jobs and Idempotency	316

Cron at Large Scale	317
Building Cron at Google	319
Summary	326
25. Data Processing Pipelines.....	327
Origin of the Pipeline Design Pattern	327
Initial Effect of Big Data on the Simple Pipeline Pattern	328
Challenges with the Periodic Pipeline Pattern	328
Trouble Caused By Uneven Work Distribution	328
Drawbacks of Periodic Pipelines in Distributed Environments	329
Introduction to Google Workflow	333
Stages of Execution in Workflow	335
Ensuring Business Continuity	337
Summary and Concluding Remarks	338
26. Data Integrity: What You Read Is What You Wrote.....	339
Data Integrity's Strict Requirements	340
Google SRE Objectives in Maintaining Data Integrity and Availability	344
How Google SRE Faces the Challenges of Data Integrity	349
Case Studies	360
General Principles of SRE as Applied to Data Integrity	367
Conclusion	368
27. Reliable Product Launches at Scale.....	369
Launch Coordination Engineering	370
Setting Up a Launch Process	372
Developing a Launch Checklist	375
Selected Techniques for Reliable Launches	380
Development of LCE	384
Conclusion	387

Part IV. Management

28. Accelerating SREs to On-Call and Beyond.....	391
You've Hired Your Next SRE(s), Now What?	391
Initial Learning Experiences: The Case for Structure Over Chaos	394
Creating Stellar Reverse Engineers and Improvisational Thinkers	397
Five Practices for Aspiring On-Callers	400
On-Call and Beyond: Rites of Passage, and Practicing Continuing Education	406
Closing Thoughts	406

29. Dealing with Interrupts.....	407
Managing Operational Load	408
Factors in Determining How Interrupts Are Handled	408
Imperfect Machines	409
30. Embedding an SRE to Recover from Operational Overload.....	417
Phase 1: Learn the Service and Get Context	418
Phase 2: Sharing Context	420
Phase 3: Driving Change	421
Conclusion	423
31. Communication and Collaboration in SRE.....	425
Communications: Production Meetings	426
Collaboration within SRE	430
Case Study of Collaboration in SRE: Viceroy	432
Collaboration Outside SRE	437
Case Study: Migrating DFP to F1	437
Conclusion	440
32. The Evolving SRE Engagement Model.....	441
SRE Engagement: What, How, and Why	441
The PRR Model	442
The SRE Engagement Model	443
Production Readiness Reviews: Simple PRR Model	444
Evolving the Simple PRR Model: Early Engagement	448
Evolving Services Development: Frameworks and SRE Platform	451
Conclusion	456

Part V. Conclusions

33. Lessons Learned from Other Industries.....	459
Meet Our Industry Veterans	460
Preparedness and Disaster Testing	462
Postmortem Culture	465
Automating Away Repetitive Work and Operational Overhead	467
Structured and Rational Decision Making	469
Conclusions	470
34. Conclusion.....	473

A. Availability Table.....	477
B. A Collection of Best Practices for Production Services.....	479
C. Example Incident State Document.....	485
D. Example Postmortem.....	487
E. Launch Coordination Checklist.....	493
F. Example Production Meeting Minutes.....	497
Bibliography.....	501
Index.....	511

Introduction

*Written by Benjamin Treynor Sloss¹
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 system—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 components and deploying them to work together to produce a service. Sysadmins are then tasked with running the 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 to absorb the additional work. Because the sysadmin role requires a markedly different skill set than that 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

¹ Vice President, Google Engineering, founder of Google SRE

emulate. A relevant talent pool is already widely available. An array of existing tools, software components (off the shelf or otherwise), and integration companies are available to help run those assembled systems, so a novice sysadmin team doesn't have 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/or 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 different 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 type 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 change 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 “flag flips,” “incremental updates,” or “cherry-picks.” 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 to 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. When 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 life-long software 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 that 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 their 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 opera-

tional 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 in order to launch them quickly. This basic premise describes the whole model of error budgets. As soon 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 no 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 alert 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 require 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. The 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 avoid 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:

² See “Disaster Role Playing” on page 401.

- 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 ultimately 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 in a service's performance. SREs and product developers will (and should) monitor and modify a service 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 the larger software engineering discipline. The rest of the book explores the SRE Way in detail.

³ For further discussion of how this collaboration can work in practice, see “[Communications: Production Meetings](#)” on page 426.