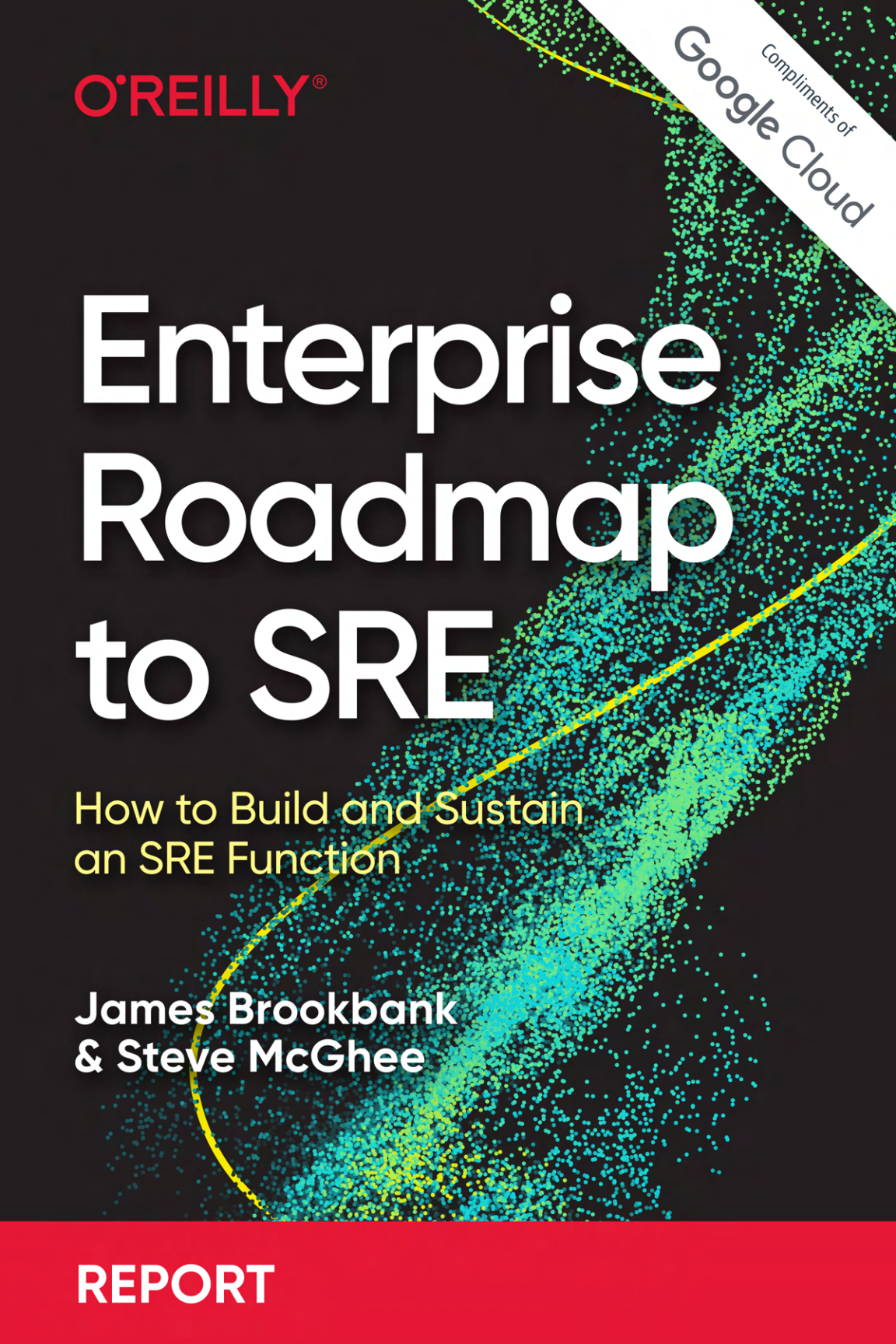
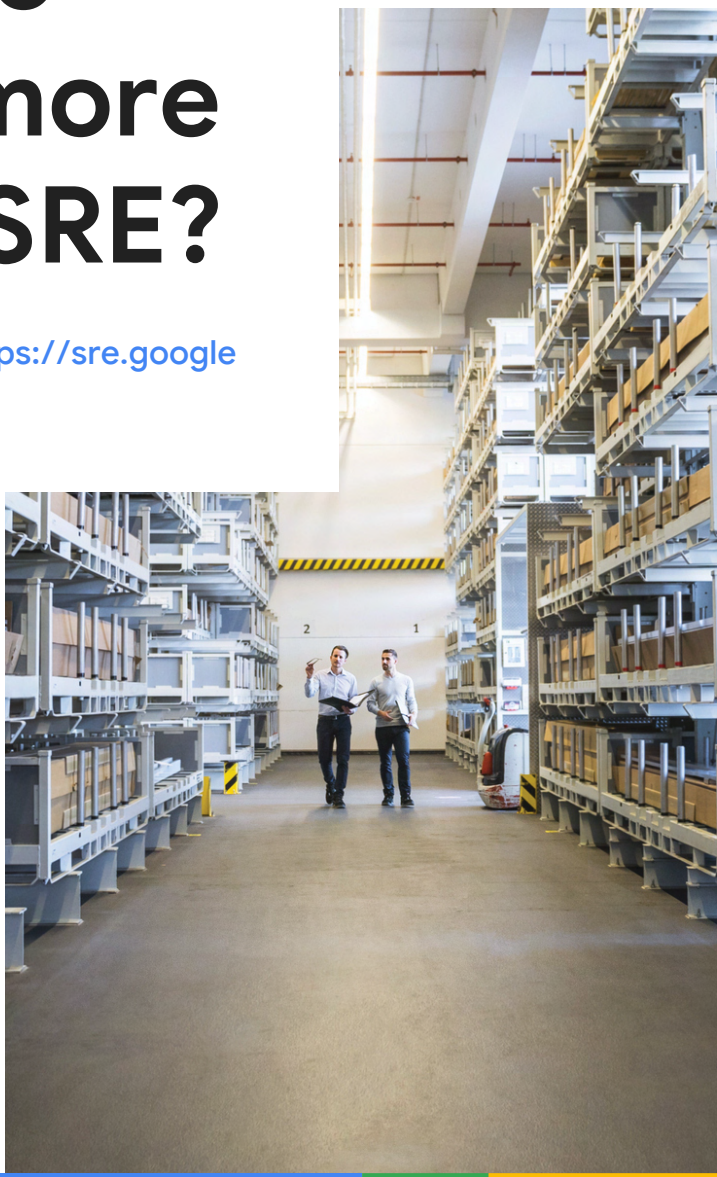

O'REILLY®
Compliments of
Google Cloud
Enterprise Roadmap to SRE
How to Build and Sustain an SRE Function
James Brookbank & Steve McGhee
REPORT

Google Cloud

Want to know more about SRE?

To learn more, visit https://sre.google



Enterprise Roadmap to SRE

How to Build and Sustain an SRE Function

James Brookbank and Steve McGhee

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Enterprise Roadmap to SRE
by James Brookbank and Steve McGhee

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://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Christopher Faucher
Copyeditor: Tom Sullivan
Proofreader: Stephanie English
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

January 2022: First Edition

Revision History for the First Edition
2022-01-20: First Release

978-1-098-11771-9

[LSI]

Table of Contents

Preface

Two previous O'Reilly books from Google—*Site Reliability Engineering* by Betsy Beyer, Chris Jones, Niall Richard Murphy, and Jennifer Petoff (Eds.) and *The Site Reliability Workbook* by Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara, and Stephen Thorne (Eds.)—demonstrated how and why a commitment to the entire service lifecycle enables organizations to successfully build, deploy, monitor, and maintain software systems.

This report is designed to build on the foundation of those books and delve a little deeper into the challenges of adopting site reliability engineering (SRE) in large and complex organizations (which we refer to as *enterprises*). Despite the popularity of SRE over the past few years, we have feedback from numerous enterprises that there is a gap between the enthusiasm for SRE and the level of adoption.

We think this is an important gap to close because reliability is increasingly a major differentiator for enterprises. The pace and scale of technology change triggered by both cloud adoption and the COVID-19 pandemic often requires different techniques to handle this increased complexity.

These topics will be of more interest to you if you are involved in (or depend on) reliability for production systems and need to know more about SRE adoption. This includes executive and leadership roles but also individual contributors (cloud architects, site reliability engineers [SREs], platform developers, etc.) Regardless of role, if you design, implement, or maintain technology systems, there is likely something here for you.

CHAPTER 1

Getting Started with Enterprise SRE

Introducing SRE into an existing enterprise can seem daunting, so we've gathered suggestions here to help you. Get started by evaluating your existing environment, setting expectations, and ensuring you take steps in the right direction as you assess SRE and how it might work within your organization.

Evolution Is Better Than Revolution

One of the defining features of enterprises is that there will always be a history of previous IT/management information system (MIS) methodologies and principles, and we will discuss a few common methodologies in detail. Regardless of the current state, we've seen the most success in adopting SRE when choosing to evolve and complement existing frameworks rather than fighting them head-on. Also, SRE is not immune to the fact that with any technology adoption process, *history matters* (see the Wikipedia page on path dependence). In short, this means that in a complex system such as an enterprise, applying the same changes from different places will lead to divergent and not convergent outcomes. We'll start with some examples of how to succeed with different popular frameworks.

SRE Practices Can Coexist with the ITIL Framework

The Information Technology Infrastructure Library, or ITIL, is a set of detailed practices for IT activities, such as IT service management (ITSM). Not every enterprise uses ITIL, but if you do have some level of ITIL adoption in your organization, then be prepared for there to be substantial overlap between SRE and ITIL practices. Also, because ITIL is a framework, your particular implementation may have wide variations from the library.

> **Key Point:** ITIL is five core books with thousands of pages on how to build and run IT services, and a lot of these topics aren't reliability-related and intentionally not covered by SRE. ITIL is a framework whereas SRE is a set of practices, so they are definitely compatible, but expect challenges in translating terminology, e.g., "warranty," "utility," and so on. Also, SRE has strong opinions on areas such as change management and service ownership, so be prepared to adjust *how* you do things even if the outcomes are aligned.

There are some common antipatterns for SRE that will prove challenging to reconcile. A change advisory board (CAB) is a common pattern for change control. The SRE approach to continuous delivery means streamlining and making this body strategic: you can read more in Google's DevOps Research & Assessment (DORA)'s article explaining streamlining change approval. Similarly, for a network operations center (NOC), this should move from an event-driven model to a more proactive approach, centered on automation and enablement. In both cases, focus on evolving the model rather than trying to immediately replace it.

DevOps/Agile/Lean

DevOps has a multitude of definitions. For simplicity, we'll assume it includes the relevant parts of other methodologies such as Agile (Scaled Agile Framework [SAFe], Disciplined Agile Delivery [DAD], and Large Scale Scrum [LeSS]) and Lean (Six Sigma, Kanban). Google's DORA research shows that SRE and DevOps are complementary, so if you have some level of DevOps adoption in your

organization, this will usually be beneficial. As with ITIL, expect some overlap with SRE and DevOps practices, and your particular implementation may have wide variations from The DevOps Handbook.

We'll cover specific SRE practices in more detail later, but many capabilities most commonly associated with DevOps (e.g., version control, peer review, etc.) are also generally considered prerequisites for SRE adoption. Whether you choose to build these capabilities through DevOps or SRE initiatives is up to you, but they will still need to be done to ensure adoption success.

> **Key point:** Be pragmatic about reconciling your DevOps and SRE initiatives; successful large-scale change is achieved iteratively and incrementally. It's important to deconstruct the individual activities and focus on enabling people rather than spend unnecessary time and effort getting the perfect "big picture."

Although they are complementary, there are some areas of DevOps and SRE that might prove challenging to reconcile. For example, you may have decided to replace Dev and Ops reporting hierarchies with cross-functional DevOps teams. In this context, reintroducing a dedicated function such as SRE needs careful consideration.

Start Where You Are

Whatever current methodologies and frameworks you currently use, it's important to understand and be honest with yourself about where you are today. As per the SRE book, "Hope is not a strategy!" If you don't think anything is missing from your current environment or there is any opportunity for improvement, then you should ask yourself why you're looking to adopt SRE. Likewise, some of your technologies or employees won't initially seem aligned with your SRE vision. Take the time to understand this before making changes.

Outline Your Expectations and Vision

Next, it's important to understand what outcomes you expect. SRE has a number of technical and cultural components but they all have a common goal of meeting reliability targets. You should expect to spend meaningful time and effort in defining how they interact with your existing frameworks. Simply stating "better reliability" isn't going to work. Similarly, if you are expecting outcomes not related to reliability (e.g., cost, velocity), then be prepared to do some extra work adapting SRE practices to your overall vision.

SRE Starts with People

Processes and technologies come and go over time, whereas people and practices are able to adopt and adapt them. If you start with training and hiring, then you can always add or remove technologies and processes. Building capabilities is an incremental process, so don't try to hire your way to success. Think about hiring as augmenting your training rather than replacing it. *Remember that SRE needs a generative culture to be successful, so ensuring this is critical.*

Embrace Your Uniqueness

There isn't one best practice way to adopt SRE within your specific organization. The only right way is the one that you are successful with. We now have the benefit of substantial study of what did and didn't work at multiple organizations, but you will inevitably make novel mistakes. Use these organizations' experiences as a genuine learning tool to build virtuous improvement loops into your organization.

CHAPTER 2

Why the SRE Approach to Reliability?

Reliability isn't a new concept. Enterprises have always placed their reliability, quality, or uptime as qualities they strive to improve. So what's different about SRE? Why is this happening today, what is different about it, and why should it matter to enterprises?

Setting Reliability as a Key Product Differentiator

Why do enterprises want to build SRE teams or otherwise pursue reliability? What are the outcomes they are hoping to achieve? Popular fads come and go (technology, processes), but there needs to be a substantial business value for them to stay. Consider reliability and security. Neither are initially a clear product differentiator, but both are assumed requirements. It is only in the presence of *problems* coupled with *high expectations* or *reliance* on a product that either become more pronounced. For example, years ago, security exploits and hacks were relatively rare, so security was present, but not on the marketing materials of a consumer or business-facing product. Now, as vulnerabilities are more commonplace and front-of-mind, we see security as a product differentiator.

Reliability (or more commonly, availability or *uptime*) tends to be mentioned mostly in the context of service-level agreements (SLAs) and similar agreements or expectation-setting fine print. However,

we feel its presence in customer satisfaction (CSAT) scores, third-party sites like Downdetector, and the overall trend of moving more of our lives and businesses onto the internet. During the COVID-19 pandemic, many software as a service (SaaS) products experienced huge uptakes in business and had to dramatically increase their reliability expectations.

Apart from availability, the commonly understood proxy for reliability, we can also consider features like durability, data residency, speed or performance under load, consistency, and quality of results as similar features of reliability that consumers and customers of internet services implicitly expect.

Once we understand that reliability is actually a highly desired feature of a product, we might even take the leap to state that it is the *most* desired feature of a product. Because if the product is not available, none of its features can be leveraged. If they cause frustration due to performance or quality, they won't be satisfying. If they cannot be used during peak, critical moments, they might not be worth having at all.

Google Search is famously "always up" to the point that its presence is felt as ubiquitous. The availability of Google Search is a key differentiating factor when comparing to its competitors, alongside speed, quality, ease of use, and user experience. This is not an accident, but a deliberate choice and investment made by Google for over a decade.

When to Focus on Reliability?

When startups consider their reliability investments, they might see it as premature. Especially when they consider the full measure taken by large organizations like Google. This is perfectly valid because a startup is meant to first build only a minimal viable product, not a durable, resilient service. However, as soon as you establish viability, integrate reliability into a product roadmap as soon as possible, alongside security and other "horizontal" efforts (internationalization, accessibility, etc).

Costly, custom reliability investments can be premature in these startups or early-stage businesses, but like security, many of the offerings in the reliability space are becoming more commoditized through open source software and third-party offered services and

tools. Leverage these early to avoid any late attempt at painful integrations into large, complex existing systems, or in response to reliability issues. Proactive consideration is key when it comes to reliability and preparedness. Also, it is worth noting that, although companies like Google build many reliability systems in-house, this is by no means the most cost-effective method. Leveraging external services and tools is a well-established best practice. Buying over building is recommended, especially in fields like security ("never roll your own cryptography") and reliability, due to the myriad edge cases and side effects possible. While the reliability vendor field is not as mature or large as security's is today, it is growing and can make a large difference to a growing company.

McKinsey's "The three horizons of growth"[1] model (see Figure 2-1) can be helpful in planning your investment. It describes three ways to think about the future of your company:

- Horizon 1 is the areas of effort that are already important today.
- Horizon 2 is your expected new areas of growth.
- Horizon 3 is long-term potential areas of growth, currently in a state of research and development.

By considering various levels of investment for each horizon, teams can break up the seemingly infinite amount of work in the reliability space.

By focusing initially on Horizon 1, we ensure reliability efforts that maintain continuous innovation in a company's current business model and short-term needs. This includes traditional Ops work, which can be automated through SRE practices. Service monitoring (service-level objectives [SLOs]), incident response, and continuous integration/continuous delivery (CI/CD) are all part of this work.

Horizon 2 considers extending the core business into new markets and targeting new customers. An extension of the existing reliability functions to support greater breadth of consumers and an expansion of infrastructure across the planet is likely in order. These present new reliability challenges, such as distributed teams (24-7

1 "Enduring Ideas: The Three Horizons of Growth." *McKinsey Quarterly*. December 1, 2009, 37. *https://www.mckinsey.com/business-functions/strategy-and-corporate-finance/our-insights/enduring-ideas-the-three-horizons-of-growth*

coverage), capacity planning for multiple customer bases, multiregion deployments, and less ability overall to fall back on things like maintenance windows, which are inherently reasonable only for a localized product, not a global one (it's never "the middle of the night" everywhere).

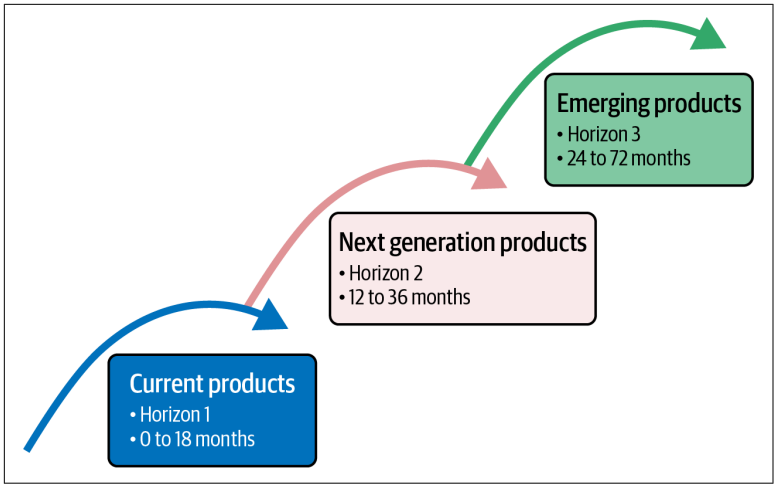


Figure 2-1. The three horizons of growth. Source: The Alchemy of Growth, by Mehrdad Baghai, Stephen Coley, and David White (Basic Books).

Finally, reliability efforts around Horizon 3 include ways the company might expand its business offerings. New capabilities and new business models should be achievable to respond to disruptive opportunities or to counter competitive threats. A business that is investing in Horizon 3 ensures that their platform and architecture do not tie them to a single business model, but allow systems of varying shapes to spawn and morph, all while retaining control and quality standards. A reliable but rigid system will not suffice here. Efforts like centralized approval boards and top-down architecture mandates stifle innovation that is required in Horizon 3.

Therefore, applying SRE to Horizon 1 can have immediate effects on your business. Placing SRE as a core foundation into Horizon 2 can ensure its potential success. However, Horizon 3 is not the best place to start SRE, as efforts there have not yet established viability.

Why Is SRE Happening Now?

Why wasn't SRE invented and popularized in the 1970s? Why not in 2010? The complexity of internet-based services has clearly risen recently, and most notable is the rise of cloud computing. We consider the cloud as the commercial successor to distributed systems, a deep academic field in computer science. Only in the past 10+ years has this branch of computer science become relevant to individual consumers (i.e., Google, Facebook, Apple) and businesses (Salesforce), or has its principles been used to deliver scalable internet systems effectively and broadly by service providers (Akamai, Stripe), not to mention cloud providers.

The introduction of "warehouse-scale computing" has brought a change in how businesses build, deliver, operate, and scale their services. Clearly, these models are changing how businesses treat costs, moving from a capital expenditure (CapEx) model of renting or building space and purchasing computer systems to an operational expenditure (OpEx) model of renting slices of compute services on demand. But beyond that are implications around systems design, architecture, and coping with changing failure modes.

Traditional infrastructure follows the model of a building or a pyramid: a large strong base, built from the bottom up. If the base fails, this spells disaster for everything above it. We call this a component-based reliability model, or a *union* model, in which *all* components in a system are expected to be available for it to work; this is visualized in Figure 2-2.

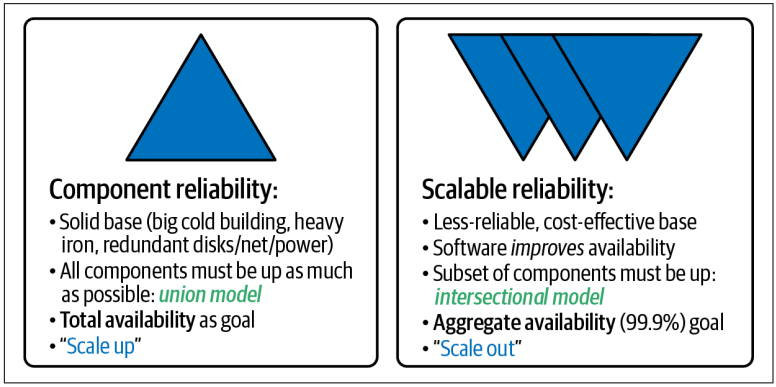


Figure 2-2. The pyramids of reliability.

The newer model used in cloud computing is that of a probabilistic-reliability or *intersectional* model, in which only a subset must be available for the system to work, due to architectural choices that *expect failure*.

While this concept is not totally novel or hard to understand, it's also not obvious to adopters of cloud services, especially when we present offerings like "lift and shift" that propose equity between the old and new models. While it's entirely possible to run the old model on a new platform, many considerations must be taken into account, often bewildering the unprepared. For example, a traditional IT department might take pride in the uptime of any given virtual machine (VM), while cloud VMs are expected to be more ephemeral: being created and destroyed intentionally, often quite quickly.

Enterprises see success stories in modern companies like Google, Facebook, and Apple, seeing two major benefits: (1) innovation at scale and (2) reliable systems at scale. Not only can these companies build new systems, but they can keep them available, fast, and correct. This combination is incredibly powerful to a business because it enables them to respond to market demands rapidly and deliver wide-reaching solutions to entire markets.

Beyond the Google Halo

Of course, Google was not always huge. In fact, in the early days, Google managed its fleet in a more traditional manner. What made Google different was an early shift from vertical scaling to horizontal scaling of its fleet. That is, moving from buying bigger and more powerful computers to a model of buying more, cheaper computers. This change was visible by walking the aisles of an early colocation facility. While many tenants during the first dot-com boom had slick-looking, expensive hardware in their cages, Google instead had a vast amount of commodity hardware instead, expecting single machines to fail at any time, while accounting for this failure in software.

An important factor in this transition was the deliberate choice not to scale the operational costs associated with this shift to horizontal. That is, when scaling horizontally, it didn't make financial sense to staff an operational team that scales linearly with the number of machines under their control.

This technological and financial choice drove an organizational one—arguably this is what begat SRE. Google simply made this choice before most other companies did because it was a very early web-scale company. Most companies at the time didn't have the need to consider the amount of processing or storage that Google needed to store, index, and serve search results for the entire internet.

We believe that many companies are now facing similar scaling challenges that Google faced back then. Except now these companies have the advantage of public cloud instead of having to fill their own datacenters with copious commodity hardware. We believe that the fact that Google was able to overcome this change in approach by evolving the SRE function should mean that SRE can help these companies overcome the same hurdles.

An important point we learned at Google about staffing levels in SRE, as compared to traditional IT Ops staffing, is that of sublinear scaling. By this, we mean that the size of a team operating a system should not grow at the same rate as the base unit of a system. If your system doubles in usage, you should not need twice as many operators. Google chose not to scale by number of machines, but by other higher-order measures: clusters, services, or platforms. By focusing on higher abstractions, teams can do more, with less overhead. These abstractions tend to be built and extended by those who operated systems modeled on previous abstractions.

Complexity can increase the need for SREs, but your staffing should grow more slowly than the onboarding of services, a concept referred to as sublinear scaling. This is actually directly related to the principle of reducing toil on a team. If a team finds itself doing repetitive tasks to keep systems healthy, they will inherently have to do more of these tasks as the systems grow and multiply. The SRE management must actively prevent and track this. This can be a slippery slope if teams allow too much toil to go unchecked, and a downward spiral results in unsustainable teams and increased outages.

Why Not More Traditional Ops?

By taking advantage of Google's experience shifting from vertical to horizontal scaling, and the associated changes that resulted through the development of SRE, your organization can reap the benefits of

scale sooner, while also saving money. Consider the alternative: to reduce the cost of scaling a team whose responsibilities are increasing with complexity, one might instead look for cheaper labor (e.g., "staff augmentation" or "right-shoring"). This is an all-too-common approach when dealing with scale and complexity. This often results in the unexpected result of stifling the growth of a system, causing outages, and actually *increasing cost* over time. These unexpected costs may come not only as outages or damage to one's brand but also other forms of revenue loss due to reduced speed to execute or scale, time-to-market for new products, and even eventually being overtaken by competitors. An organization needs to consider the whole picture before choosing how to optimize costs on operational and reliability investments.

Therefore, it's important to staff your SRE team appropriately. You don't need to hire all PhDs, but you don't want to skimp, either. Try not to focus just on the unit cost of operators but, rather, the *overall system cost*. To make a comparison, industrial food packaging uses million-dollar machines to pack peaches into cans. You might wonder, "Why not just hire unskilled workers? That would be much cheaper than the million-dollar machines." At first glance, this sounds cheaper. When you consider the *overall system cost* of hiring unskilled workers, however, it's actually more expensive. Therefore, the overall system of using the million-dollar machines ends up being better and cheaper than hiring unskilled workers. SREs and platform engineers will build your canning robots, if you let them. Don't force them to pack tins manually just because that's what you used to do.

Adopting a high-performance practice like SRE is much harder without high-performing staff. So are existing teams out of luck? Not at all. Evolving existing talent is entirely possible and highly suggested. Teams might be tempted to hire an outside expert or even an entire team of external folks with SRE experience, but this can be a mistake. Similarly, expecting to gain (*long-term*) SRE capabilities through staff augmentation or offshoring is not likely to result in the outcomes you expect. SREs tend to have a higher unit cost than traditional Ops teams, and attempting to undercut or find ways to gain high value for low cost when it comes to staffing SRE teams tends to fail. If reliability is valued within your organization, you should be able to rationalize this cost, and we'll explore ways of doing this in later sections.

Treating operations as a siloed cost center is a common mistake. You should consider the big picture of revenue and total cost of ownership, avoid local optimization of costs, and recognize that simply focusing on short-term cost-cutting can end up costing your company far more in the long run. For example, by considering failure scenarios and estimating their impact on revenue or brand, a team can position themselves as a form of long-term insurance, complete with a roadmap of mitigations and prevention strategies. Ideally, this team is more than just "insurance" but actually a driver of an improved (reliable!) ability to deliver innovation to customers, too. Consider your Horizon 2 goals and plan your platform for that. Don't just solve today's problems; plan for the future.

Consider the benefits of transforming your existing staff. Given the right *incentives*, *opportunities*, and sufficient *time*, an organization can transform its norms and gracefully adopt modernized roles and responsibilities for its people, while also minimizing unnecessary churn. Because, without a doubt, an organization's *most valuable asset* is always its *people*. A real understanding of your company's core business is not to be undervalued when assessing the skill set of a staff member.

CHAPTER 3
SRE Principles

Before we talk about specific practices, it's important to be clear on principles, similar to the legal terminology of adherence to the letter and spirit of the law. Practices themselves aren't enough; the spirit of SRE is in the principles. Practices also can't be exhaustive—they are proxies for principles and vary over time and from org to org.

Principles are the fundamental truths that form the foundation of your transformation and help guide decision making. There are often multiple ways to achieve business objectives, so encouraging people to live and breathe a core principle is better than setting exhaustive rules that can be followed in letter but not in spirit. Google's own principles are an example of this; while there are multiple internal policies regarding how we design and build new services, the core principle we always try to adhere to is, "Focus on the user and all else will follow."

Your focus should be on enabling people to demonstrate leadership at every level rather than being bound by a series of directives that disempower individuals. In particular, business functions and managers need to be convinced by the transformation narrative and must be willing to amend the detailed guidance within the context of their specialist areas. These influencers are your greatest assets once they are convinced, and your biggest hurdle if not.

Similar to principles, good policies focus on outcomes and not tasks; however, they are more prescriptive guidance. They are the vehicle to harness the bureaucracy of your business instead of fighting against it. Policies and policy frameworks should empower people to

operate safely within well-understood guardrails. They should also contain sensible defaults to nudge behavior in the right direction.

> *Antipattern: Having a big, up-front plan or design of how things are going to be implemented.*
>
> Inherently, you'll need to spend a lot of time learning, and we recommend building feedback loops (virtuous cycles) guided by coherent principles.

We're going to give a quick overview of each principle from the SRE book and how to translate that to your org. For more details, we recommend reading the referenced chapters in the Site Reliability Engineering book.

Embracing Risk (SRE Book Chapter 3)

This is one of the toughest first steps. We often phrase this as a trade-off between reliability and velocity; however, this isn't necessarily true. The most helpful way to frame reliability for enterprises is around exponential cost. Approximately each order of magnitude level of 9's (e.g., 99.9 to 99.99) results in an order of magnitude increase in costs, whether software or hardware or people. Considering whether this provides good return on investment helps adjust this to business requirements. Types of failures are also incredibly important. For example, services that operate 24-7 are much better suited to SRE (as opposed to internal company systems that run 8 hours, 5 days a week). Also, SRE will be much less useful for services that are not being actively maintained because, for these services, we apply less continuous improvement. This is especially true if you're intentionally not making many releases or writing new code.

> *Antipattern: 100% reliable services*
>
> 100% is the wrong target for pretty much everything.
>
> *Antipattern: Getting 99.999% in "normal" operations*
>
> Monthly metrics or maintenance windows can obscure the outsized impact of disasters.

Service-Level Objectives (SRE Book Chapter 4)

Start with service-level indicators (SLIs) before you worry about SLOs and SLAs, and use the data on your systems to craft accurate SLIs, which can then support your SLO/SLA negotiations. Do not let your existing business commitments drive your SLO/SLI *accuracy and relevance*—you have a choice of using metrics to drive changes or using changes to drive metrics. Don't sugar coat or cherry pick. Spend the time to understand what your customers want over using convenient data points that support your theory. In short, let evidence (SLI/SLO) drive your conclusions (SLAs). Try and focus SRE on the >99.9% services and let the <99.9% ones be maintained without SREs (until they need it). *We can't emphasize enough that if a service doesn't benefit from SLOs/SLIs, it probably won't benefit from SRE either.* Finally, if you can't make software or process changes in the event of an SLO violation, there won't be much benefit from SRE either.

Antipattern: SLO = SLA
: Always set your SLOs tighter than your SLAs (e.g., SLO: 99.95%, SLA: 99.9%)

Antipattern: SLI = OKR (objectives and key results)/KPI (key performance indicator)
: Goodhart's law applies here: when a measure becomes a target, it ceases to be a good measure.

Eliminating Toil (SRE Book Chapter 5)

This is probably one of the most important principles since it's closely tied to the generative culture required for SRE to succeed. Most of the time, enterprise leadership wants to speed things up and does this by making sure all resources are 100% busy. If you genuinely want to make sure your teams are doing the right thing instead of doing the wrong thing fast, then you'll aim for less than 50% busywork (or as we call it, *toil*). This is the secret to reliability (and speed) at scale. Do not equate this with technical debt, i.e., something you can store up and pay back later, or tackle this once a quarter with "toil week." Once toil overwhelms your team, all the other SRE activities will grind to a halt. You must decide what toil is

for your org, and this must be decided by the SRE practitioners, not decided from above. The definition of toil also changes over time (again set by practitioners).

Antipattern: Toil as an optional principle
: Ignoring toil reduction will have an outsized impact on your SRE adoption. If you don't have time to reduce toil, then you don't have time to implement SRE.

Antipattern: Toil is one person's/one team's job instead of everyone's
: The people closest to the work need to be the ones fixing it. If you try to offload this work, it will drive the wrong behaviors.

Antipattern: Toil fixing week
: Having a "toil fix week" once a quarter is a common temptation, but this won't work. Your approach to eliminating toil needs to be more systematic and constant.

Monitoring Distributed Systems (SRE Book Chapter 6)

Observability is a specialized discipline in its own right, and it needs as much care and thought as the rest of your development practices. Realistically, most enterprises should expect to invest in a variety of systems that will help your teams do their jobs better. A "single pane of glass" isn't going to work well; neither will having hundreds of overlapping tools. Try to find a balance that works for you by understanding your unique SRE user journeys and how they will need to use multiple tools to diagnose and resolve logical connections between systems. Treat the observability systems as an internal product deserving of investment and thoughtfulness, emphasizing usable tools over "perfect" dashboards because systems are always changing. Keep in mind that over-alerting is as bad as under-alerting: *alerts should not go to a human unless there is an expectation of an action.* Building this alert learning cycle is a common way to accelerate learning; getting it wrong will rapidly burn out SREs.

Antipattern: Alerts to nowhere
: An email inbox full of ignored alerts means that no one will respond to high-severity alerts because of too much noise.

Antipattern: "NoOps" tooling that replaces SREs
: Tooling augments SREs but doesn't replace them. Trying to completely eliminate operations as a discipline isn't possible and will quickly alienate your SRE teams.

Antipattern: Alerts are causes
: You can log many things but always alert on symptoms, not causes.

The Evolution of Automation at Google (SRE Book Chapter 7)

Automation is most important when it comes to extremely high reliability levels (99.99% or over), since at this point, you'll almost always experience SLO breaches if a human has to intervene. The balance of intervention as error budgets shrink is increasingly toward proactive maintenance, with approaches such as graceful failure, retries, etc. Automation for its own sake is also a common problem, and taking time to fix bad processes is incredibly important but very difficult to embed in team culture. Automation always needs to be as easy to maintain as other parts of the system.

Antipattern: Defaulting to automation regardless of process quality or fit
: The best code is the code not written! Playbooks are a good intermediate solution for infrequent processes.

Antipattern: Unnecessary human intervention for the "really important" parts
: Only involve a human if you genuinely need them to make a decision and they are empowered to do so.

Release Engineering (SRE Book Chapter 8)

Release engineering overlaps extensively with the continuous integration/continuous delivery (CI/CD) practices your DevOps teams are probably already working on. Leverage that work rather than trying to impose top-down practices. Emphasize outcomes and flow metrics to align teams, and make sure you have sufficient investment in a platform team (or teams depending on your scale). Shift left on release aspects as much as possible—i.e., involve testing teams earlier in the process and think about testing at all stages. Don't overload your developers but ensure each part of the release cycle is treated as valuable and aligned with the others. The release pipeline is the cause (and therefore the fix) of most SRE problems. The on-call/maintenance staff need to be tightly aligned.

Antipattern: The DevOps/SRE team does releases for all things
: That's just operations with different job titles.

Antipattern: Release engineering has to introduce CI/CD
: Continuous delivery is a discipline in its own right, and your platform and development teams need to build that foundation (SRE can help).

Simplicity (SRE Book Chapter 9)

Cognitive load on teams is important, and will change over time as the remit of teams expands or shrinks. Make sure you allow for teams to be merged or split to match cognitive load. Essential complexity means that many things will be hard to understand, so try to incentivize the reduction in accidental complexity and split essentially complex things into smaller, easier-to-manage chunks, e.g., domain-driven design (DDD). Another important concept to reuse from DevOps is low context versus high context, and SRE concepts such as playbooks, documentation, Disaster Recovery Testing (DiRT) exercises, and so on are all important parts of making things low context. Less code and fewer product features likely flies in the face of most of your product incentives, so make sure you keep this in check when you consider its reliability impact.

Antipattern: Simple means I can understand it
: An executive dashboard can't meaningfully display everything. Don't try to force it.

Antipattern: Static teams based on annual reviews
: Dynamic team formation is needed more than once a year.

How Do You Map These Principles to Your Existing Organization?

It's highly unlikely that these principles are already fully aligned to your organization, and that's okay! Your version of SRE doesn't need to exactly match Google's—only the principles do. However, make sure you're intentionally choosing what to pursue, check for mismatches between existing principles, and use this time to double check for vanity metrics (see "success theater" as explained in *The Lean Startup* by Eric Ries [Crown Business]). Basing change on shaky foundations can be hard, so if you aren't confident, then assume you need to check and change things. Try not to hedge on the principles—if you don't think something can be done, then delaying its implementation is better than pretending it's working.

Preventing Org-Destroying Mistakes

Changes can have very different potential impacts. It's inevitable that some changes you make when adopting new principles won't always work. The impact of the change is usually less important than the ability to roll it back, meaning that the changes that are hardest to reverse can often cause the biggest pain. Focus on the changes that are easier to reverse, so even if these are mistakes they will still give rise to learning. Consider as an example that you can always have another "reorg" if the first one doesn't work, but you can't unfire people.

Antipattern: Fire all the Ops people who can't code
: Aside from the obvious ethical or legal implications, you simply can't reverse this decision.

Antipattern: Give root/production access to all developers
: Good security and operations practices such as least privilege apply to automation more than ever.

Antipattern: Picking the most critical system in the business to start with
: You wouldn't start a marathon training program with a 26-mile run on day one.

Create a Safe-to-Fail Environment for Your Adoption Journey

Expect failure, but make sure you're learning and improving from it. When doing *complicated* things, make sure you have subject matter experts involved, but when doing *complex* things, make sure you either reward failure or have failure budgets. It's hard to genuinely reward failure in most orgs, so a failure budget is sometimes more appropriate. This means that you're measured on the top n% of things that go right rather than the mean/median. It's crucial that you role-model these behaviors in your leadership team or they will not be able to embed these across the organization.

Antipattern: We'll support any risks you take as long as they are successful
: Real risk budgets mean accepting some failures.

Beware Diverging Priorities

It's possible you have absolute buy-in from your entire leadership team. What's more likely, however, is that people will want reliability but have valid concerns over change and cost. We recommend acknowledging the J curve of change, as shown in Figure 3-1—this means that after an initial set of somewhat easy wins, the curve to making impactful changes becomes difficult. For example, adapting your own new automation can feel like a step backward before significant gains are realized. Make sure you're setting up for success by making roofshots versus moonshots. You can still aim for dramatic improvements, but be conservative at first.

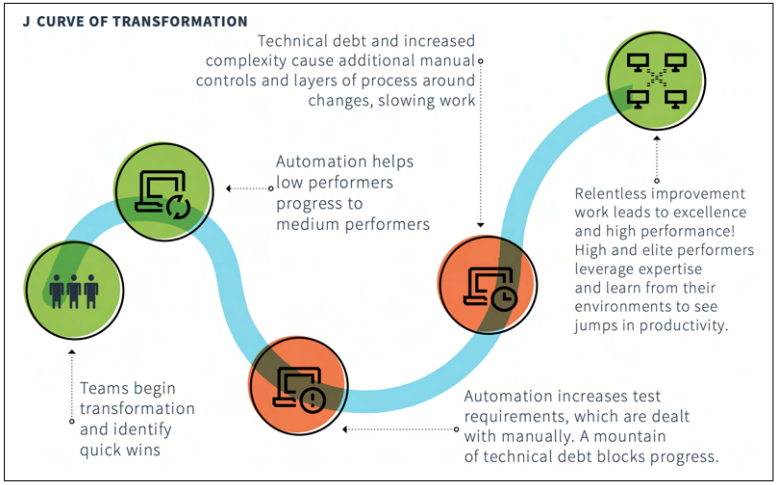


Figure 3-1. The J curve of transformation from the DORA 2018 State of DevOps Report.

> *Antipattern: Giving up too soon. For example, trying SRE for six months, then stopping after no immediate wins*
>
> This doesn't mean that you need to achieve everything up front, but there has to be a clear narrative around moving in the right direction after a couple of quarters.

How Do You Get Buy-In to These Principles, with the Critical Sign-Off and Backing You Need?

Make sure you are setting SRE up for success by considering general enterprise change principles such as the ones mentioned by John Kotter or BJ Fogg. It's okay if your leadership team doesn't fully believe in what you're attempting, but you need to make sure, at minimum, that there is enough urgency to make the change and the motivation to enact it.

In technology, we often reward fixing problems instead of preventing them from happening, and the adoption of SRE principles and practices is liable to fall victim to this mode of operation. Ensure that your SRE efforts aren't turned down after several months due

to "insufficient impact" by making the ongoing value of your SRE adoption visible. A proven way to do so is to find the right metrics for *your organization* to demonstrate this return on investment. For example, in retail, you might focus on maximizing sales during Black Friday, while in healthcare you might focus on continuous compliance and availability, and in finance it might be about throughput of a trading system or the speed to complete an analysis pipeline.

Antipattern: If you build SRE, they will come
: Practices can't exist in a vacuum. You have to start doing the work to be able to make genuine improvements

Antipattern: Steady gradual upward progress
: Real-world change has ups and downs. If you're not failing, then you're not learning.

CHAPTER 4
SRE Practices

Once you've established an SRE team and have a grasp on the principles, it's time to develop a set of practices. A team's practices are made out of what its members can do, what they know, what tools they have, and what they're comfortable doing with all these.

What teams do is initially based on their charter and their environment. Often, it defaults to "everything the dev team is *not* doing," which can be dangerous. By focusing a team on a subset of operational duties, they can produce a flywheel of capabilities that build on each other, over time. If they're just thrown in the deep end with an undefined scope, toil and frustration will certainly result. Another common antipattern is to add SRE work onto an already overburdened team.

What the team knows can be expanded via education, either self-imposed or centrally organized. Teams should be encouraged to hold regular peer education sessions—for example, a weekly hour where any question about production is welcome, from either new or veteran team members. If a question is answerable by someone, a teaching session can result. If nobody knows the answer, it can turn into a collaborative investigation. In our experience, these sessions are highly valuable for everyone on the team. Junior team members learn new things, seniors get a chance to spread their knowledge, and often something new is discovered that *nobody* knew about. Similarly, Wheel of Misfortunes, or tabletop exercises, in which team members meet in an informal setting to discuss their roles during an emergency and their responses to a particular emergency

situation, are extremely helpful for getting people more comfortable with touching production in a stress-free environment. Reliving a recent outage is an easy way to start. If one team member can play the part of the Dungeon Master and present the evidence as it played out in real life, other team members can talk through what they would have done and/or directly use tooling to investigate the system as it was during the event.

Teams should also be encouraged to gain more knowledge about the systems they are operating from development teams. This is not only a good exercise to better understand the existing system but also an opportunity to directly introduce new instrumentation, discuss and plan changes to the system such as performance improvements, or address scalability or consistency concerns. These conversations tend to be highly valuable in developing trust between teams.

A team's capabilities can also be expanded through the introduction of new, third-party tooling, through open source software tools, or by the teams writing their own tools.

Where to Start?

When adding capabilities to a team, where do you begin? The problem space of reliability and SRE is vast, and not all capabilities are appropriate at the same time. We suggest starting with a set of practices that allow a team to learn what to work on next. Abstractly, we refer to a model called plan–do–check–act (PDCA). By basing your next step on how the system currently is working, your next step will always be relevant. We explain later in this chapter how to build a platform of these capabilities and where to start. This set of early capabilities will form a flywheel, so your teams won't have to *guess* at what to build or adopt next—it will develop naturally from their observations of the system.

Where Are You Going?

It's important to set your goals appropriately. Not all systems need to be "five 9s" and super reliable. We recommend classifying your services and apps based on their reliability needs and set levels of investment accordingly. As we mentioned previously, remember that each nine costs 10 times as much as the previous nine, which

is to say that 99.99% costs 10 times more than 99.9%. While this statement is difficult to prove exactly, the principle is true. Therefore, setting targets blindly or too broadly without consideration can be expensive and run efforts aground. Forcing excessive reliability targets onto systems that don't need them is also a good way to cause teams to lose top talent. Don't aim for the moon if you just need to get to low earth orbit.

Make sure your path to success is one of "roofshots," making incremental progress toward your goals. Don't expect to achieve it in one large project or revolution. Incremental improvement is the name of the game here.

As you spin out new practices within your team, make sure you record the benefits you're gaining. These gains should be promoted within the team and to stakeholders or other peer teams. Peer recognition is very important and includes praising members in a team standup, putting people on stage to retell how they avoided a catastrophe, publishing near-misses in a newsletter, and drawing out to the larger organization what might have happened if it weren't for preventative measures. It's important to celebrate this type of work, especially in an environment that hasn't done so in the past. Verbal and written praise can also be coupled with monetary bonuses or gifts. Even a small gift can go a long way.

How to Get There

Don't try to have a long-term (e.g., three year) detailed plan. Instead, focus on knowing the direction of travel. Know your north star, but generate your next steps as you accomplish your last ones. Once you've established your direction of travel, you don't need to "blow up" your existing teams and processes that don't align with the new model. Instead, try to "steer the ship" in the right direction.

We think of this as the Fog of War approach, wherein you know your destination but are ready for any hiccups along the way. Short-term planning and agility are essential here, especially early on, when quick wins and immediately demonstrable impact can have major positive effects on a fledgling program and the morale of the team. Give yourself achievable goals that solve today's problems, while starting to build generic, reusable capabilities that multiple teams can use. By building out a *platform* that delivers these

capabilities, you can scale the impact of your investment. We expand on this concept of platform and capabilities later in this chapter.

Not every product development team within an organization is equal in terms of their needs and their current capabilities. As you introduce SRE to an enterprise, you should strive to be flexible in your engagement models. By meeting product teams where they are, you can solve today's problems while also introducing org-wide norms and best practices. As an SRE team gets off the ground, they can feel over-subscribed if many teams are looking for their help. By developing a clear engagement "menu," you can avoid one-off engagements or other unsustainable models. There are several types of engagement models: embedded, consulting, infrastructure, etc. These are described well in a blog post by Google's Customer Reliability Engineering (CRE) team, as well as the model described in Chapter 32 of the SRE book.

For SRE adoption, reporting structure is important to clarify early on. We recommend an independent organization, with SRE leaders having a "seat at the table" with the executive team. By separating the SRE leadership structure from the product development one, it's easier for SRE teams to maintain focus on the core goals of reliability, without direct pressure from teams that are more motivated by velocity and feature delivery. However, take care when doing this not to build an isolated "Ops" silo because it's critical that SREs work closely with other parts of the enterprise. Development teams should invest in these common SRE teams in such a way that the value derived from this team is greater than building out the SRE function from within their own ranks.

What Makes SRE Possible?

What makes SRE possible? Is it just a series of practices like SLOs and postmortems? Not exactly. Those are actually *products of the culture* that made SRE work to begin with. Therefore, a successful adoption of SRE should not just mimic the practices, but must also adopt a compatible culture to achieve success.

This culture is rooted in the *trust and safety of the team* itself. The team must feel psychologically safe when they're put in the high-pressure position of control over major systems. They must be able to say "no" to their peers and leaders without fear of retribution. They must feel their time is valued, their opinions heard, and their

contributions recognized. Most of all, SREs should not be made to feel "other" or "less" than their counterparts in a development organization. This is a common pitfall, based on the historically rejected models of Dev versus Ops.

A well-known example of this is *blameless postmortems*. By writing down "what went wrong," a team is able to collaboratively determine contributing factors that result in outages, which might be either technical or procedural. Often, when mistakes are made by humans, it can be tempting to cite "human error," but this has been shown to be somewhat meaningless and not an effective way to improve a system. Instead, SRE promotes *blamelessness*. An easy way to think of this is that the system should make it difficult for a human to make a mistake. Automation and checks should be in place to validate operator input, and peer reviews should be encouraged to promote agreement and collaboration. You know you have blameless postmortems when people freely include their names in reports for situations in which they made mistakes—when they know there will be no shaming, no demotion, and no negative performance reviews due to simple mistakes that could happen to anyone. If you see postmortems referring to "the engineer" or "Person 1," you may consider this is a good blameless practice, but this could actually be due to underlying cultural problems that must be addressed directly. If names are redacted and replaced with "the engineer" or "Person 1" on paper, but blame is still cast on the engineer outside the context of the postmortem, the culture of blame has not been addressed. You should definitely not automate the process of explicitly redacting names from logs or documents—this does not solve the cultural problem, and it just makes documents harder to read and understand. Rather than superficially redacting names, address the culture underneath to move toward blamelessness.

One sign of bad culture is *watermelon metrics*: green on the outside, but red inside. These are metrics reflecting the efforts of a team that are contrived to *look good* but in reality hide real flaws. These are similar to *Goodhart's Law*, which tells us that any measurement that becomes a target ceases to be a good measure. For example, focusing on the number of support tickets or overall mean time to resolve (MTTR) can often be abused either intentionally or by those with good intent who don't realize their mistake. By measuring the activity of a team, we make that activity the goal, not the customer outcomes. Instead, a team should be able to define their own success

metrics that are directly representative of things such as customer happiness, system stability, and development velocity.

SRE should not just be a "20% time" role but, rather, a dedicated title and position within your organization. There should be a job ladder with published transfer requirements and promotion expectations. Leveling and pay should be equitable between teams. A transfer should not feel any significant effects either way.

A good way to know if an established SRE team is succeeding is by looking at transfers into and out of SRE. By ensuring that transfers are routine and free of any sort of bureaucracy or limitations, you'll quickly learn if people feel "stuck" in SRE or if it is a desirable role. By observing the rate of volunteer transfers into SRE from Development, you find out if it's working or not.

SREs must know that their time is valued, especially when their job demands exceed "normal hours." An example of this at Google is that of *time-in-lieu*: when an SRE must be available outside of normal hours (aka "on-call"), they should be compensated. Some teams at Google allow on-call engineers to choose between monetary compensation or time off, at some percentage of on-call hours, often with an agreed-to cap. There should not be more demands on a team than what can be delivered by that team, so it's important to ensure the on-call pool is of sufficient size. A common mistake is to make the on-call pool consist of only SREs. This is an artificial limitation. On-call pools should be done on an opt-in basis, as well. As soon as a team feels their time is being abused, it's a swift downward spiral.

Another cultural touchpoint is that of *planning and goal setting*. Because SREs are closest to the problems of production, they tend to have a good sense of what's most important, what's burning, what's causing the most pain. By allowing an SRE team to set their own priorities and roadmap, you empower that team, and they will be much more effective and happier. Management should follow the practice of developing an agreed-upon, shared understanding of expected outcomes. Does the business need to move faster? Do users need their results faster? A common antipattern of this is Taylorism: the model of leaders independently setting and prioritizing detailed plans and tasks, then assigning them to the workers.

Building a Platform of Capabilities

An SRE team can build a platform to deliver capabilities to their partner teams, ideally scaling their contribution to the entire organization over time. By introducing resilience mechanisms into shared services, practices, norms, and code, these teams can develop a shared *platform* made of automation, code, shared libraries, pipelines, procedures, norms, documentation, playbooks, and, yes, even that special undocumented knowledge that lives only in people's heads. Instead of each team attempting to create their own best practices, these can be baked into the platform. Products can be built from scratch on the platform (so called "digital natives") or can be ported onto the platform. As the platform's capabilities increase over time, and the team becomes more confident and comfortable with its operational characteristics, increasingly critical workloads can be ported over. By adopting this model of encoding capabilities into a platform, the SRE team can scale their impact by applying capabilities to many services together. The platform is an internal product and should be governed like one, treating service teams as customers, taking feature requests, and tracking defects (see Figure 4-1).

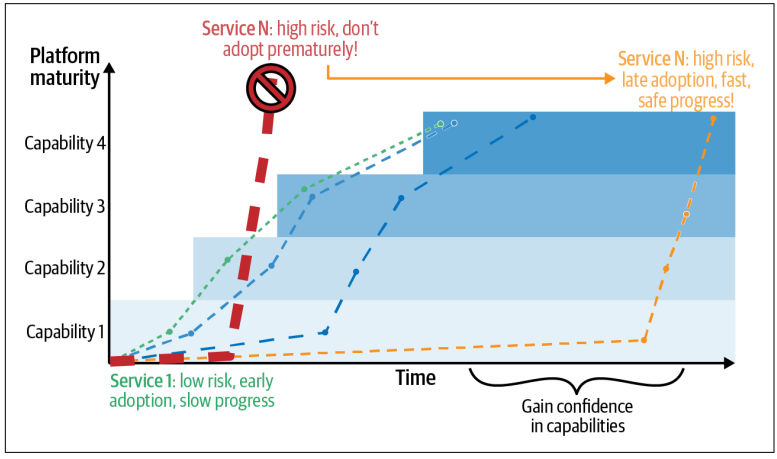


Figure 4-1. A platform of capabilities.

As a team builds a platform, the question arises, "What to build first?" By adopting low-risk services first, you can minimize that list to be an MVP, or minimum viable product. Over time, you'll add more capabilities. But which ones are next? There are two sources:

your developers and your environment. That is, build what they ask for, e.g., "We need a message bus!" and build what you know they'll need, e.g., "There has to be a scalable service discovery system or else this will never work."

For the environmental capabilities, these often come down to:

- DevOps improvements such as enhancing the software development life cycle (SDLC) and getting more code out, faster and safer
- Reliability engineering improvements: minimizing risk from the errors that do creep in

For reliability engineering improvements, we recommend developing "the virtuous cycle" within your teams. If you're not sure what to improve, you can learn by looking at your outages and doing the following:

- Institute SLOs.
- Formalize incident response.
- Practice blameless postmortems and reviews.
- Use risk modeling to perform prioritization.
- Burn through your reliability backlog based on error budget or other risk tolerance methods.

Let this cycle be your flywheel to spin out new capabilities. For example, if you have an outage in which a deployment introduced a bug that crashed every server in the fleet, you'll want to develop a way to reduce that risk, possibly through something like blast radius reduction, using canary releases, experiment frameworks, or other forms of progressive rollouts. Similarly, if you find that a memory leak is introduced, you might add a new form of load test to the predeployment pipeline. Each of these is a capability that is added to your platform, which can provide benefit and protection for each service running on the platform. One-off fixes become rare as generic mitigation strategies show their value.

Leadership

Of course, to build such a platform, you need to devote engineering hours, which might otherwise be used to develop features. This is where influence is needed, all the way up the chain. When development talent is used for both features and stability, a trade-off must be made. It's important to make sure that the *people making this trade-off* have the big picture in mind and have the appropriate incentives in place. We are increasingly seeing the role of *Chief Reliability Officer*, someone senior within your organization who has a *seat at the table* for strategic reliability decisions (this might be a familiar concept for fans of the book *A Seat at the Table* by Mark Schwartz [IT Revolution Press]). While this is a common *job role* for successful SRE adoption, it's not a common *job title*, and it is frequently an additional hat that an existing executive is wearing.

Knowing If It Is Working

A well-run organization that understands and values reliability will exhibit a few observable traits. First is *the ability to slow or halt feature delivery in the face of reliability concerns*. When velocity and shipping is the only goal, reliability and other nonfunctional demands will always suffer. Do reliability efforts always get deprioritized by features? Are projects proposed but never finished due to "not enough time"? An important caveat here is that this should not be seen as slowing down the code delivery pipeline—you should keep your foot on the gas.

Another indicator of success is when individual heroism is no longer being praised, but instead is actively discouraged. When the success of a system is propped on the shoulders of a small set of people, teams create an unsustainable culture of heroism that is bound to collapse. Heroes will be incentivized to keep sole control of their knowledge and unmotivated to systematically prevent the need to use that knowledge. This is similar to the character *Brent* in *The Phoenix Project* by Gene Kim, Kevin Behr, and George Spafford (IT Revolution Press). Not only is it inefficient to have a Brent, it can also be downright dangerous. A team has to *actively discourage individual heroism while maintaining the team's responsibility* because heroism can feel like a rational approach in short-term thinking.

Another sign of a well-functioning team is that reliability efforts are funded *before* outages, as a part of proactive planning. In poorly

behaved teams, we see that an investment in reliability is used to treat an outage or series of outages. Although this might be a necessary increase, the investment needs to be maintained over time, not just treated as a one-off and abandoned or clawed-back "once things get better."

To illustrate this further, consider a simplification of your organization's approach to reliability as two modes: "peacetime" and "wartime." Respectively, "things are fine" or "everybody knows it's all about to fall apart." By considering these two modes distinctly, you're able to make a choice about investment. During wartime, you spend more time and money on hidden features of your platform, infrastructure, process, and training. During peacetime, you don't abandon that work, but you certainly invest far less.

However, who decides when a company is in wartime? How is that decision made? How is it communicated throughout the company in ways that don't cause panic or attrition? One method is to use *priority codes*, such as: Code Yellow or Code Red. These are two organizational practices that aid teams in prioritization of work. *Code Yellow* implies a technical problem that will become a business emergency within a quarter. *Code Red* implies the same within days, or it is used for an already-present threat. These codes should have well-defined criteria that must be understood and agreed to by all your leadership team. Their declaration must be approved by leadership for the intended effect to take place. The outcome of such codes should result in changing of team priorities, potentially the cessation of existing work (as in the case of a Code Red), the approval of large expenditures, and the ability to pull other teams in to help directly. Priority codes are expensive operations for an enterprise, so you should make sure there are explicit outcomes. These should be defined from the outset as exit criteria and clearly articulated upon completion. Without this, teams will experience signal fatigue and no longer respond appropriately.

Choosing to Invest in Reliability

What other less-dramatic changes might be under the purview of such a reliability leader? These would be *policy* and *spending*. Setting organization-wide policy tends to be inconsistent at best when driven from the bottom-up. It's far more effective to have a leadership role in place to vet, dedupe, approve, and disseminate these policies as they're proposed by teams. Similarly, spending

company funds on staffing, hardware, software, travel, and services is often done in a hierarchical manner.

One has to consider the value of reliability within the organization before building out a structure as mentioned earlier. For this to make sense, the organization must consider reliability not as a cost center but as an *investment* and even as a *product differentiator*. The case to be made is that reliability is the hidden, most important, product feature. A product that is not available for use, too slow, or riddled with errors is far less beneficial to customers, regardless of its feature set. Setting this direction must be done at the executive level to set a consistent tone, especially if this is a new orientation.

One simple argument for this is that reliability can be a proxy for concepts that are better understood, like code quality. If a system introduces user-visible failures, the application of reliability practices such as gradual change can make the system appear to have fewer errors to your end customers, before directly addressing the code quality issues. For example, by rolling out a broken change to 1% of customers, 99% of those customers don't experience the problem. This makes the system appear 100 times better than it actually is and reduces support costs and reputational damage.

Making Decisions

By setting up reliability as an investment into a stronger product, you're able to make longer-term plans that have far greater impact. Traditional models treat IT as a cost center, focusing entirely on reducing that cost over time. At the end of the day, it doesn't matter that the service is cheap if it's not up. You can still apply cost reduction, but you should consider it *after* you've achieved reliability goals. If you're finding that the cost of maintaining your stated reliability goals is too high, you can explicitly redefine those goals—i.e., "drop a 9"—and evaluate the trade-offs that result.

To achieve all these goals, you'll likely need to persuade some governing board, group of decision makers, or executives. You'll need their buy-in to staff and maintain a team over time, provision resources, and train and further develop team members. This should be seen as a long-term investment and explicitly funded accordingly, not as a hidden line-item in some other budget.

Antipattern: Ignoring Ulysses

When it comes to reliability, a common antipattern is to let outages or other "bad news" affect your planning cycle, even when they're expected. Often, it's tempting for leadership to feel the need to "do something" in the face of bad news, and "sticking to the plan" often doesn't feel impactful. However, given a plan that expects outages to happen, unless a significant change in the understanding of a system occurs, "sticking to the plan" is exactly the right thing to do. The term *Ulysses pact* can be a useful illustration here. This is where a leader (Odysseus) tells his team (his sailors) to stick to the plan (sail past the Sirens as he is tied to the mast). When his team sticks to the plan (despite his thrashing and begging to stop), he congratulates them. They didn't get tempted by short-term thinking. Their plan considered the long-term impact, and they had time to make a clear plan before the chaos started.

By allowing a team to make in-the-moment decisions, you're often choosing to ignore a good plan and make emotional or ego-driven choices instead. A classic example of this is a leader "walking into an outage" and taking over command without having full context, and despite a capable team already in control. This is often the outcome of a company's culture. A culture of HiPPOs (decisions based on the Highest Paid Person's Opinion) can have a drastically bad effect on incident management and reliability in general. Instead, listen to Odysseus, stick to the plan, and don't abandon ship. This applies not only to incident response, but also things like error budget exhaustion or tracking SLOs in the face of "really bad" incidents. If your plan is to halt a feature release in the face of error budget exhaustion, but you make an exception for "this important feature" every time, your leadership will be severely undercut. An effective practice to improve this is the introduction of "silver bullets" in which a leader is granted three silver bullets to be used sparingly as an override to the expected plan. By introducing this artificial scarcity, leaders are required to make explicit trade-offs. Similarly, if a single bad event wipes out an SLO, don't ignore it. Gather the team to analyze how this changes your collective understanding of the system. Was this type of failure never considered before? Was the response to an outage insufficient?

Antipattern: Both at Once

Another antipattern is that of trying to mix old and new models without modification. This pulls teams in strange directions and should be avoided. For example, in the case of ITIL problem management, a central team is often expected to drive down causes and resolution times for all problems through a problem manager. In comparison, SRE expects embedded engineers to drive their own problem resolution through postmortems and reviews. While the outcomes are still aligned (fewer, shorter outages), the methods and personas differ greatly. By trying to do both at the same time, you end up with confusion, and the intended outcomes from both approaches conflict with each other and suffer.

We call these bad mixtures of SRE and non-SRE principles "toxic combinations," similar to the medical term referring to bad mixtures of medicines. Each on its own can be beneficial, but the combination of the two together causes an unintended bad consequence. Often, we find good intent behind using both, often due to trying to keep existing staff involved, or in an attempt for continuity in reporting. However, the appeal of this is far outweighed by the worsened outcomes: longer outages, more toil, and reduced reliability.

Staffing and Retention

Staffing and role definition can also present antipatterns. When building out an SRE team, it can be tempting to hire an SRE from the outside to impose order on the existing team. This can actually result in wasted effort, often with the hired SRE failing to understand the nuances of the team or technology in place already and falling back to applying previously used methods, without knowing if they're in fact reasonable in the new job.

We suggest growing existing teams into SRE teams instead. Simply renaming them isn't effective, but providing a structured learning path and an environment to grow and thrive can certainly work. There are cases where the transition might fail, of course. If individuals are not set up to succeed and instead are expected to immediately turn into a senior SRE just by "reading the book," they can become frustrated and look for employment elsewhere. Similarly, some engineers don't see the reason for change, aren't incentivized,

or otherwise are highly resistant to adopting a new role. By providing paid education, time and room to learn, and the context to help your team understand why the change is needed, you can successfully transition a team into the SRE role. This takes time, effort, and patience. In cases where it doesn't stick, it's important to conduct an exit interview, specifically to address the transition and what did or didn't work for an individual. You may uncover flaws in your plans or discover that it isn't being executed in the way you intended. Finally, as you ask teams to do more complex work that has higher impact, note that this is, literally, higher-value work and the team should be compensated for it. That is, as your team starts acting like SREs, you should pay them like SREs, or else they'll move to somewhere that does. If you pay teams to learn high-value skills and they leave to use those skills elsewhere, you have only yourself to blame.

Upskilling

When growing and transitioning existing staff into SREs, it is critical to build an upskilling plan. This includes both the what and the how—that is, what skills are needed in the role and how you'll go about enabling staff to acquire those skills. Tools like skills gap analyses and surveys are intrinsically useful here to check assumptions about the foundational skills that are required for the job. These skills, often not talked about specifically in SRE literature, are nevertheless essential to allow SREs to scale their contributions organization-wide. For example, it is not unheard of for traditional operations teams to be unfamiliar with software engineering fundamentals such as version control, unit testing, and software design patterns. Ensuring that these baselines are a part of your upskilling plan and that they are tailored to each learner profile is crucial, not just to establish a critical mass of skill on the team but to provide a smooth on-ramp for individuals into the new expectations of their role (and thus help reduce team churn).

CHAPTER 5

Actively Nurturing Success

Once you've decided that SRE is worth pursuing for your organization and resolved to invest in it, it's important to ensure that your investment is a successful one. It's always hard to introduce change into a system, but it's even harder to make that change stick. Here are some tips on how to keep SRE working in your organization.

Think Big, Act Small

The statement, "If you can't measure it, you can't manage it," is frequently associated with Edwards Deming. The full quote, however, is "It is wrong to suppose that if you can't measure it, you can't manage it—a costly myth." SRE, at its core, is a metrics-driven methodology. No amount of SLOs or SLIs, however, will help you understand whether your SRE adoption is both working and aligned with your enterprise strategy. You'll have to find this out through continuous experimentation and learning.

In previous chapters we've asked you to "think big," but when it comes to nurturing success, you should "act small." Any kind of large-scale change is achieved iteratively and incrementally, and SRE isn't immune from these challenges. There's also an obvious caveat to this—if you have too short of a timeline, you won't be able to make meaningful change, so be prepared to find the balance.

Google internally uses shared objectives and key results (OKRs) to align teams and set goals when it's not always clear how they'll be achieved. Your organization might have its own processes to do this, but they must be extended to include explicit iterations and periodic reviews of SRE team metrics (toil, alerting, Software Engineering impact, capacity plans, etc.). The nonlinear nature of adoption means your progress will always include setbacks, so this should also be treated as a normal part of the process.

Culture Eats Strategy for Breakfast

One of the assumptions Google makes, which is a key unwritten part of the SRE story, is the underlying generative Google culture. Google also shared the research we conducted to describe these attributes. As it turns out, *who* is on a team matters less than *how* the team members interact, structure their work, and view their contributions.

We learned that there are five key dynamics that set successful teams apart from other teams at Google:

Psychological safety
: Can we take risks on this team without feeling insecure or embarrassed?

Dependability
: Can we count on each other to do high-quality work on time?

Structure and clarity
: Are goals, roles, and execution plans on our team clear?

Meaning of work
: Are we working on something that is personally important for each of us?

Impact of work
: Do we fundamentally believe that the work we're doing matters?

Many of the typical concerns we see with SRE adoption are things such as cost implications, specialized industry concerns, technical debt, etc. However, the best thing about this discovery is that, like all good things, these five dynamics are essentially free! Regardless of your industry or situation, consider targeting these items as a priority. The high-performing teams at Google relied on these

cultural norms to make SRE succeed, effectively making SRE an emergent behavior from this cultural base.

Avoiding Culture Won't Help; Neither Will Waiting for It

It's usually frustrating to hear us talk about culture being critical to successful SRE adoption, with the implication that you should wait for your culture to get to a certain point before you can adopt SRE. To paraphrase a popular proverb, the best time to start changing your culture might have been 20 years ago, but the second best time is now. There are also substantial repercussions aside from reliability concerns to not making your culture responsive to reliability feedback.

What Does Nurturing SRE Mean?

To nurture and grow SRE, there are some key activities to consider.

1. Sublinear Scaling

We've already mentioned this earlier, but it's important to clarify that this isn't about "doing more with less" but, rather, using automation and continuous improvement culture to change the way we approach reliability problems. SRE is explicitly designed not to be scalable through headcount, so resist the temptation to add more people to existing steps in your software assembly line, and use SREs to automate or eliminate those steps instead.

2. Building and Retaining Sustainable, Happy Teams

While the tech industry has moved in the direction of project to product approaches, it's still very common to see individuals treated as fungible resources moved between activities on a whim. This directly conflicts with our cultural advice. Don't expect to do this and succeed with SRE.

3. Acknowledging That Sre Is Not Static—It's Inherently a Dynamic Role, and Grows over Time

Part of the evolutionary process of reducing toil and implementing automation means that SRE will evolve within your organization.

You can still budget and plan for this, but aim for outcomes rather than specific tasks and fixed team sizes. This will feel strange at first because it conflicts with a lot of top-down planning activities. When SREs dynamically re-team, however, it's usually a sign that you are succeeding.

4. Assessing Your Reliability Mindset Level and Target Within Your Organization

It takes a longer time than you'd expect to attain a high level of SRE adoption. Inside Google, we think getting to a strategic level of reliability for a product can be a 3- to 5-year journey. Given the constant effort to maintain this level, it's also common to revert to old habits. Therefore, take the time and energy to assess and adjust to this new mindset on a continual basis.

SRE Care and Feeding

Once SRE has been started, you'll want to care for and feed your fledgling organization. As your SRE practice develops, you'll need to consider the following.

Growing a Foothold Team into a Larger Org

Don't start with the biggest problem you have or the giant core monolith of your enterprise that everyone is afraid to touch. You'll need some level of quick wins to get started and build your team, principles, and practices in a supportive environment. Conversely, don't start with a toy service. SRE is only valuable where there is an important reliability need. Once you have a foothold, you need to continuously learn to expand safely. It might seem attractive to take on a large number of less important services, but resist this temptation. SRE value is in the high reliability services. Other services should follow the "you build it, you run it" model.

SRE org structure: Separate SRE Org Versus Embedded Teams

Google has always had a dedicated SRE org since its inception, and we think there are considerable benefits to doing so—such as reliability culture, release prioritization, hiring, and so on. Often, we are asked to compare this with the DevOps approach of "breaking down

silos." It's critical to understand that the separate SRE management chain is never supposed to be a silo. SREs have a multitude of ways of working with development teams, from embedded individuals to light-touch consultancy. Having said this, you might still have success deploying SRE without a dedicated organization structure, but be prepared to need extensive senior leadership support.

Promotion, Training, and Compensation

SREs are developers and should expect compensation and incentives at least equal to the other developers in your organization. Promotion rates are also a great indicator to see if there is parity with other teams. You should regularly compare both pay and promotion rates to close any gaps. Guard against any assumption that this pay scale allows you to mistreat SREs (e.g., with longer hours). Also note that SRE expectations will be higher about being able to do meaningful and impactful work.

Going on-call is a scary and draining activity that needs careful preparation and training for. It's also critical to compensate teams for on-call in a meaningful way. If you have restrictions on direct compensation, then get creative with indirect methods (e.g., time in lieu).

Communication and Community Building

SRE enablement covers a wide variety of activities such as formal training courses, tech talks, reading groups, etc. Much of this will be indirect work, done by giving time and resources to experiment (e.g., 20% work). Autonomy and empowerment are key to building communities, and this needs to be done through active (as opposed to passive) leadership. This means setting out a clear leadership vision or north star, and visibly role modeling empowerment within the organization. It's easy to underestimate the amount of communication involved in any kind of transformation, and SREs are also exceptionally good at detecting inauthentic messaging.

Gauging When Your SRE Adoption Is Effective

It's common to acquire a large number of SRE artifacts in your adoption journey such as SLOs, SLIs, error budgets, dashboards, etc. These are all proxy metrics for your organization, but they won't always give you a holistic picture of how reliability is changing.

For this, you might need to consider some more unconventional perspectives. If things are genuinely going well, the virtuous cycle will start to feel a lot calmer over time. Instead of just firefighting from incident to incident, there will be a sense of proactive fire prevention. This can be unsettling, especially if your organization has been used to demonstrating its value through busyness. Resist the temptation at this point in making tactical optimizations to regain that busy feeling. Your SREs will naturally start improving SLOs and error budgets as they experience failures and build their capabilities.

Steering the Ship

Getting to a more proactive approach frees you up to spend more time on vision. You'll start to get a better idea of exactly what reliability levels different services in your organization actually need. Decide where to optimize by using this data and setting new expected outcomes. Perhaps some of your internal systems were marked as business-critical, but your SREs now know they only need a 99.9% SLO. Other systems might now need greater reliability levels, and a surefire way to know if you are successful is when you start to see interest from those other teams in getting the benefits of SRE.

CHAPTER 6
Not Just Google

To round out the perspectives presented in this report, we spoke with three SRE leaders in different industries who have all adopted SRE in various forms over the past several years. Each has a unique story about how that adoption worked and what they might have done differently, in addition to insights into what makes SRE work in their industry or organization.

Healthcare // Joseph

Joseph Bironas has been leading SRE adoption in several healthcare organizations since his time as a Google SRE leader. As such, he was able to provide an industry-wide view of how implementing SRE in this space differs from other tech and startup cultures. Due to the nature of its life-critical workflows, reliability is often top-of-mind. However, the healthcare industry faces specific challenges that span organizational models, culture, budgeting, and regulatory requirements.

After working with a company that focused on very tight margins in areas like medical device manufacturing, as well as FDA-regulated fields, Joseph observed that reliability is understood as a requirement, but that the cost–benefit ratio of SRE is far from well-understood in the industry. As a result, SRE and infrastructure teams can find themselves as "catch-all engineering" being pulled into an IT cost center, with their scope increased dramatically.

What's wrong with an SRE team being managed under an IT cost center, you might ask? When enterprises are used to managing through broad IT frameworks like ITIL, it's hard to make value judgments about SRE, which is a mere subset of ITIL—which also handles things like hardware procurement that SRE has no opinion on. More to the point, a CIO who manages all of corporate and production IT is not in the best position to make judgments on software systems reliability. Instead, rolling up to a software-focused leader—e.g., an SVP of Engineering or perhaps a CTO—makes more sense.

Organizations in this field often face the steep curve of hoping to adopt SRE when they haven't yet adopted DevOps practices. For example, they were releasing software once a month, with very little CI/CD automation due to the necessary complexity around regulations and organization-wide compliance controls. Many healthcare organizations simply don't *want* to deploy quickly: for some customers, deploying too fast implies inadequate testing or insufficient safety.

The willingness to implement changes—such as a meaningful pivot to SRE—varies widely across the industry, perhaps due to differing leadership priorities and styles. Joseph described one scenario in which a team was able to send designers out into the field to gather requirements, build new workflows, and revolutionize care through a better product. In another scenario, a different team was only incentivized to be better than the incumbents, which didn't require the same level of investment. In a third scenario, a team was plagued by inertia, waiting for a top-down mandate before making any change or investment. In Joseph's experience, more progressive leadership tends to be more sensitive to customer demands for reliability.

In contrast to startup culture, change is very slow for some of these teams. One particular team questioned if they could accomplish "anything" (e.g., adopting SRE) in 18 months—an eternity for a startup. When considering significant changes in organizations with this pace, you have to have models to help understand planned returns on investment. Knowing about the J curve (see roofshots versus moonshots) is important here to avoid abandoning an effort in its trough, before the real return. Joseph recommended checking in quarterly with teams to keep a steady cadence on progress. He recommends starting the transition to SRE with incident response

and building a cycle of continuous learning with incident reviews (say, for six months) before focusing on SLOs. To make a "real investment" that can't just fail silently, you might seek executive sponsorship, implement top-level OKRs, or focus on whatever makes an effort "real" in your organization. It's also critical to not just learn from this cycle, but to put what you learn into action.

Another common mistake in the healthcare industry is to overlook "the software side of SRE," when a team is used to focusing on traditional Ops work: "It's difficult to imagine that you can make XX% of your IT spend just go away via software." This core value of SRE is often a foreign concept to leaders, and might even be deliberately resisted or even undermined by some entrenched sysadmins and operators. Ignoring this aspect can make SRE seem highly ineffective. Software engineering is also difficult and expensive. Even if you're buying commercial SRE-adjacent tooling (which is imperfect despite the huge number of contributors over many years), you can't escape the integration work, which is largely a software engineering effort.

Budgeting for reliability can also be problematic. Joseph points out that "the industry doesn't have ads-revenue curves [that Google had when SRE was built]." This impacts their ability to specialize and invest like Google did and leads them to depend more on commercial solutions. Business budgeting and planning are often still in waterfall mode, which can be a challenge for SRE work—the time required to explore, understand, and design new solutions isn't well suited to waterfall-style work.

In terms of takeaways, he's seen that it can apply across all industries. Joseph shared a story illustrating how sometimes even an imperfect effort can be a valuable starting point. In the case of one company he worked with, leadership wanted a dramatically simple version of error budgets. Instead of choosing SLOs that were appropriate for their critical user journey (CUJ), they declared a single SLO (99.95% available) for everything. This target was simple to understand, but it crippled the concept of SLOs for the whole engineering team. Stateful and stateless applications, batch and real-time, all had the same SLO, which ultimately wasn't useful and undermined confidence in the technique. This also resulted in error budgets that didn't make any sense, so those were similarly crippled, as was any process that attempted to use those error budgets.

At the end of the day, though, there was value in the fact that people started measuring things they were not measuring before. SLOs gave the teams a way to ask each other a question they were not asking before. This all goes to show that it's important to talk to each other and help each other make good decisions based on the data in front of us.

Retail // Kip and Randy

Commodore "Kip" Primous and Randall Lee from The Home Depot (THD) provided insight about how a large retailer adopted SRE, their successes, and some of their current challenges. THD was an early big retail customer with Google Cloud Platform (GCP), and part of their cloud adoption was to adopt SRE, following the principles detailed in the recently published SRE book. Six years later, what they expected to build and what exists now are entirely different.

Kip started out as the Reliability Engineering (RE) manager with the "dot-com" business unit, working on "the Browse Stack" for THD's ecommerce website. Randy started at THD before Kip and shared a joint goal for SRE: to improve resilience through a better platform. They initially considered building their own cloud and datacenters, but then they evaluated the various cloud service providers and landed on GCP. As they moved to the cloud, the only way to succeed was simultaneously changing how they worked, via something like SRE or "DevOps 2.0."

Originally, THD's goal was to move away from their huge monolithic commerce service. Project Aurora was funded and championed by their VP, who sought to achieve economies of scale, reduce the size of the operational team, and change the team from hundreds of contractors to significantly fewer full-time associates. There was also a general intent to improve reliability and reduce the reliance on other teams (that might not be well-aligned) within the organization. The dot-com team hoped to operate at "internet speed."

Alignment was important. Before moving to Cloud, every deployment was "like launching the space shuttle: lots of coordination based on years of hard work." The team felt that the current model of DevOps had run its course within THD/dot-com. By introducing RE, the team was able to follow a new pattern of distinct work on

a new platform and with a new remit and felt empowered to work through anything in the stack that related to reliability. They hired a lot of cloud native engineers and automated as much as they could. They were able to skip past the boundaries that were limiting the existing DevOps teams.

From 2015 to 2017, SRE was able to move swiftly and independently because they were working on new cloud infrastructure, with different and modern tools and hardware. Then, in 2018, the enterprise teams caught up—SRE was no longer the only team working on GCP. Much to everyone's relief, the two sides were able to converge as the centralized enterprise team updated their traditional models—for example, acknowledging that they shouldn't keep track of patches on individual machines in the new world of ephemeral VMs. By bringing the teams together through a series of constructive conversations, The dot-com RE team was able to work with the newly formed centralized enterprise team and was able to help establish more enterprise-friendly processes and better adherence to company security guidelines. In addition, they were able to offload much of the broader GCP platform management (billing, permissions, quotas, etc.) from the RE team to the enterprise team.

As THD underwent their SRE journey, Kip and Randy observed several patterns and lessons that will likely translate to other industries. The process of getting other teams to adopt SRE concepts took years and happened in cycles: there would be a push to improve compliance automation, followed by cost improvements, then access control, then cyber security. Each interaction required a lot of discussion and education. During quiet periods of few outages or little downtime, a sense of urgency might instead come from outside events. The Equifax outage or the Akamai or Facebook DNS problems might cause everyone to scramble and kick off another cycle of reliability improvement.

Executive sponsorship was critical for the success of SRE adoption within THD. After the initial success of the dot-com migration to the cloud by using the SRE model, the SRE role became synonymous with high performance within the company. Many other teams wanted to replicate the model, and some were forced to implement SRE regardless of defined SLO requirements. However, not all were as fortunate to start green-field and cloud native like the dot-com team did. This led teams to struggle to recognize the value the RE team brought to the table, and they sometimes mistook

expectations of roles and responsibilities. This kind of ambiguity can cause problems when a team interacts with an "unofficial" RE in one part of the organization who might not work at the same level or use the same principles as the original team—for example, someone who "just pushes buttons" with no real plan to automate toil. Such an experience leaves a bad taste in the mouth of the team, who then aren't interested in working with other RE teams in the future.

Kip also warns that without a new SRE-inspired effort every few years, reliability standards devolve. REs broke down walls, but those walls are being re-erected. Teams think, "Reliability isn't my problem, it's the REs' problem!", which is the wrong message to send. Randy adds that a well-functioning RE team can backslide without constant reinforcement and education of RE practices and principles, in addition to well-defined roles and responsibilities.

Currently, THD is in the mode of "doubling down" on RE, which can actually be an antipattern if changes aren't held to SRE principles. SRE isn't a panacea that you can apply to every problem, but it's hard for a team to see success with SRE and not want to then apply SRE everywhere. Recently, Kip was told to run RE in distribution centers and on vendor-supported physical hardware, which isn't a natural fit for SRE. While there are always opportunities to improve the reliability of these systems, it's more challenging to apply many of the RE practices in non-cloud-native environments. Perhaps a better path forward for some areas of the business is not SRE, but practices like value stream mapping or Lean. To avoid these types of dynamics, it might make more sense to apply SRE as a "pull" model, as opposed to a "push" model: don't force SRE on teams, just offer it as a service and let them come to you.

Kip and Randy's biggest piece of advice is to focus on executive education and to recognize the value of a champion. If you don't have top-down support, it's difficult to fund any meaningful change. Trying to obtain funding through product dev teams results in a dynamic where those teams "never want to pay the tax." Whenever they do pay the tax, they only want SREs to work directly on their products and toward their product goals.

At THD, there was originally a senior leader who championed the creation and growth of the RE team for dot-com and beyond. THD is now in a strange position of having many RE teams working on

various projects, with varying degrees of the ability to apply SRE principles. Randy and Kip expect that having a more senior player would improve things at THD. A VP of Reliability who owned all the RE roles would provide economies of scale. Without a central RE organization, the SRE role can morph to the point where SREs from different orgs are doing completely different things and following different standards and principles.

Conclusion

We hope this provides some insight into how your enterprise might adopt SRE, and where the challenges may lie. We think your chances of success are higher if you clearly define your SRE principles, map those to practices and capabilities, and prioritize growth and nurturing of those within your team. We also showed some examples of teams that have gone through the process of spinning up an SRE practice within an enterprise, and the specific challenges they faced and overcame.

We think this report will help your adoption of SRE and lead to a more reliable technology experience for everyone. And we hope that through this adoption operations teams can become more sustainable, services can be more scalable, and development velocity can increase.

"May the queries flow and the pager be silent."

About the Authors

James Brookbank is a cloud solutions architect at Google. Solution architects help make cloud easier for Google's customers by solving complex technical problems and providing expert architectural guidance. Before joining Google, James worked at a number of large enterprises with a focus on IT infrastructure and financial services.

Steve McGhee is a reliability advocate, helping teams understand how best to build and operate world-class, reliable services. Before that, he spent more than 10 years as an SRE within Google, learning how to scale global systems in Search, YouTube, Android, and Cloud. He managed multiple engineering teams in California, Japan, and the UK. Steve also spent some time with a California-based enterprise to help them transition onto the Cloud.