Embedding an SRE to Recover from Operational Overload

Written by Randall Bosetti
Edited by Diane Bates

It's standard policy for Google's SRE teams to evenly split their time between projects and reactive ops work. In practice, this balance can be upset for months at a time by an increase in the daily ticket volume. A burdensome amount of ops work is especially dangerous because the SRE team might burn out or be unable to make progress on project work. When a team must allocate a disproportionate amount of time to resolving tickets at the cost of spending time improving the service, scalability and reliability suffer.

One way to relieve this burden is to temporarily transfer an SRE into the overloaded team. Once embedded in a team, the SRE focuses on improving the team's practices instead of simply helping the team empty the ticket queue. The SRE observes the team's daily routine and makes recommendations to improve their practices. This consultation gives the team a fresh perspective on its routines that team members can't provide for themselves.

When you are using this approach, it isn't necessary to transfer more than one engineer. Two SREs don't necessarily produce better results and may actually cause problems if the team reacts defensively.

If you are starting your first SRE team, the approach outlined in this chapter will help you to avoid turning into an operation team solely focused on a ticket rotation. If you decide to embed yourself or one of your reports in a team, take time to review SRE practices and

philosophy in Ben Treynor Sloss's introduction and the material on monitoring in Monitoring Distributed Systems.

The following sections provide guidance to the SRE who will be embedded on a team.

Phase 1: Learn the Service and Get Context

Your job while embedded with the team is to articulate why processes and habits contribute to, or detract from, the service's scalability. Remind the team that more tickets should not require more SREs: the goal of the SRE model is to only introduce more humans as more complexity is added to the system. Instead, try to draw attention to how healthy work habits reduce the time spent on tickets. Doing so is as important as pointing out missed opportunities for automation or simplification of the service.

Ops Mode Versus Nonlinear Scaling

The term *ops mode* refers to a certain method of keeping a service running. Various work items increase with the size of the service. For example, a service needs a way to increase the number of configured virtual machines (VMs) as it grows. A team in ops mode responds by having a greater number of administrators managing those VMs. SRE instead focuses on writing software or eliminating scalability concerns so that the number of people required to run a service doesn't increase as a function of load on the service.

Teams sliding into ops mode might be convinced that scale doesn't matter for them ("my service is tiny"). Shadow an on-call session to determine whether the assessment is true, because the element of scale affects your strategy.

If the primary service is important to the business but actually is tiny (entailing few resources or low complexity), put more focus on ways in which the team's current approach prevents them from improving the service's reliability. Remember that your job is to make the service work, not to shield the development team from alerts.

On the other hand, if the service is just getting started, focus on ways to prepare the team for explosive growth. A 100 request/second service can turn into a 10k request/second service in a year.

Identify the Largest Sources of Stress

SRE teams sometimes fall into ops mode because they focus on how to quickly address emergencies instead of how to reduce the number of emergencies. A default to ops mode usually happens in response to an overwhelming pressure, real *or imagined*. After you've learned enough about the service to ask hard questions about its design and deployment, spend some time prioritizing various service outages according to their impact on the team's stress levels. Keep in mind that, due to the team's perspective and history, some very small problems or outages may produce an inordinate amount of stress.

Identify Kindling

Once you identify a team's largest existing problems, move on to emergencies waiting to happen. Sometimes impending emergencies come in the form of a new subsystem that isn't designed to be self-managing. Other sources include:

Knowledge gaps

> In large teams, people can overspecialize without immediate consequence. When a person specializes, they run the risk of either not having the broad knowledge they need to perform on-call support or allowing team members to ignore the critical pieces of the system that they own.

Services developed by SRE that are quietly increasing in importance

> These services often don't get the same careful attention as new feature launches because they're smaller in scale and implicitly endorsed by at least one SRE.

Strong dependence on "the next big thing"

> People might ignore problems for months at a time because they believe the new solution that's on the horizon obviates temporary fixes.

Common alerts that aren't diagnosed by either the dev team or SREs

> Such alerts are frequently triaged as *transient*, but still distract your teammates from fixing real problems. Either investigate such alerts fully, or fix the alerting rules.

Any service that is both the subject of complaints from your clients and lacks a formal SLI/SLO/SLA

> See Service Level Objectives for a discussion of SLIs, SLOs, and SLAs.

Any service with a capacity plan that is effectively "Add more servers: our servers were running out of memory last night"

> Capacity plans should be sufficiently forward-looking. If your system model predicts that servers need 2 GB, a loadtest that passes in the short term (revealing 1.99 GB in its last run) doesn't necessarily mean that your system capacity is in adequate shape.

Postmortems that only have action items for rolling back the specific changes that caused an outage

> For example, "Change the streaming timeout back to 60 seconds," instead of "Figure out why it sometimes takes 60 seconds to fetch the first megabyte of our promo videos."

Any serving-critical component for which the existing SREs respond to questions by saying, "We don't know anything about that; the devs own it"

> To give acceptable on-call support for a component, you should at least know the consequences when it breaks and the urgency needed to fix problems.

Phase 2: Sharing Context

After scoping the dynamics and pain points of the team, lay the groundwork for improvement through best practices like postmortems and by identifying sources of toil and how to best address them.

Write a Good Postmortem for the Team

Postmortems offer much insight into a team's collective reasoning. Postmortems conducted by unhealthy teams are often ineffectual. Some team members might consider postmortems punitive, or even useless. While you might be tempted to review the postmortem archives and leave comments for improvement, doing so doesn't help the team. Instead, the exercise might put the team on the defensive.

Instead of trying to correct previous mistakes, take ownership of the next postmortem. There *will* be an outage while you're embedded. If you aren't the person on-call, team up with the on-call SRE to write a great, blameless postmortem. This document is an opportunity to demonstrate how a shift toward the SRE model benefits the team by making bug fixes more permanent. More permanent bug fixes reduce the impact of outages on team members' time.

As mentioned, you might encounter responses such as "Why me?" This response is especially likely when a team believes that the postmortem process is retaliatory. This attitude comes from subscribing to the Bad Apple Theory: the system is working fine, and if we get rid of all the bad apples and their mistakes, the system will continue to be fine. The Bad Apple Theory is demonstrably false, as shown by evidence [Dek14] from several disciplines, including airline safety. You should point out this falsity. The most effective phrasing for a postmortem is to say, "Mistakes are inevitable in any system with multiple subtle interactions. You were on-call, and I trust you to make the right decisions with the right information. I'd like you to write down what you were thinking at each point in time, so that we can find out where the system misled you, and where the cognitive demands were too high."

Sort Fires According to Type

There are two types of fires in this simplified-for-convenience model:

- Some fires shouldn't exist. They cause what is commonly called ops work or toil (see Eliminating Toil).

- Other fires that cause stress and/or furious typing are actually part of the job.

In either case, the team needs to build tools to control the burn.

Sort the team fires into toil and not-toil. When you're finished, present the list to the team and clearly explain why each fire is either work that should be automated or acceptable overhead for running the service.

Phase 3: Driving Change

Team health is a process. As such, it's not something that you can solve with heroic effort. To ensure that the team can self-regulate, you can help them build a good mental model for an ideal SRE engagement.

Note

Humans are pretty good at homeostasis, so *focus on creating (or restoring) the right initial conditions and teaching the small set of principles needed to make healthy choices*.

Start with the Basics

Teams struggling with the distinction between the SRE and traditional ops model are generally unable to articulate *why* certain aspects of the team's code, processes, or culture bother them. Rather than trying to address each of these issues point-by-point, work forward from the principles outlined in Chapters Introduction and Monitoring Distributed Systems.

Your first goal for the team should be writing a service level objective (SLO), if one doesn't already exist. The SLO is important because it provides a quantitative measure of the impact of outages, in addition to how important a process change could be. An SLO is probably the single most important lever for moving a team from reactive ops work to a healthy, long-term SRE focus. If this agreement is missing, no other advice in this chapter will be helpful. *If you find yourself on a team without SLOs, first read Service Level Objectives, then get the tech leads and management in a room and start arbitrating.*

Get Help Clearing Kindling

You may have a strong urge to simply fix the issues you identify. Please resist the urge to fix these issues yourself, because doing so bolsters the idea that "making changes is for other people." Instead, take the following steps:

1. Find useful work that can be accomplished by one team member.
2. Clearly explain how this work addresses an issue from the postmortem *in a permanent way*. Even otherwise healthy teams can produce shortsighted action items.
3. Serve as the reviewer for the code changes and document revisions.
4. Repeat for two or three issues.

When you identify an additional issue, put it in a bug report or a doc for the team to consult. Doing so serves the dual purposes of distributing information and encouraging team members to write docs (which are often the first victim of deadline pressure). Always explain your reasoning, and emphasize that good documentation ensures that the team doesn't repeat old mistakes in a slightly new context.

Explain Your Reasoning

As the team recovers its momentum and grasps the basics of your suggested changes, move on to tackle the quotidian decisions that originally led to ops overload. Prepare for this undertaking to be challenged. If you're lucky, the challenge will be along the lines of, "Explain why. Right now. In the middle of the weekly production meeting."

If you're unlucky, no one demands an explanation. Sidestep this problem entirely by simply explaining all of your decisions, whether or not someone requests an explanation. Refer to the basics that underscore your suggestions. Doing so helps build the team's mental model. *After you leave, the team should be able to predict what your comment on a design or changelist would be.* If you don't explain your reasoning, or do so poorly, there is a risk that the team will simply emulate that lackadaisical behavior, so be explicit.

Examples of a thorough explanation of your decision:

- "I'm not pushing back on the latest release because the tests are bad. I'm

pushing back because the error budget we set for releases is exhausted."

- "Releases need to be rollback-safe because our SLO is tight. Meeting that SLO requires that the mean time to recovery is small, so in-depth diagnosis before a rollback is not realistic."

Examples of an insufficient explanation of your decision:

- "I don't think having every server generate its routing config is safe, because we can't see it."

This decision is probably correct, but the reasoning is poor (or poorly explained). The team can't read your mind, so they very likely might emulate the observed poor reasoning. Instead, try "[…] isn't safe because a bug in that code can cause a correlated failure across the service and the additional code is a source of bugs that might slow down a rollback."

- "The automation should give up if it encounters a conflicting deployment."

Like the previous example, this explanation is probably correct, but insufficient. Instead, try "[…] because we're making the simplifying assumption that all changes pass through the automation, and something has clearly violated that rule. If this happens often, we should identify and remove sources of unorganized change."

Ask Leading Questions

Leading questions are not loaded questions. When talking with the SRE team, try to ask questions in a way that encourages people to think about the basic principles. It's particularly valuable for *you* to model this behavior because, by definition, a team in ops mode rejects this sort of reasoning from its own constituents. Once you've spent some time explaining your reasoning for various policy questions, this practice reinforces the team's understanding of SRE philosophy.

Examples of leading questions:

- "I see that the TaskFailures alert fires frequently, but the on-call engineers usually don't do anything to respond to the alert. How does this impact the SLO?"

- "This turnup procedure looks pretty complicated. Do you know why there are so many config files to update when creating a new instance of the service?"

Counterexamples of leading questions:

- "What's up with all of these old, stalled releases?"
- "Why does the Frobnitzer do so many things?"

Conclusion

Following the tenets outlined in this chapter provides an SRE team with the following:

- A technical, possibly quantitative, perspective on why they should change.
- A strong example of what change looks like.
- A logical explanation for much of the "folk wisdom" used by SRE.
- The core principles needed to address novel situations in a scalable manner.

Your final task is to write an after-action report. This report should reiterate your perspective, examples, and explanation. It should also provide some action items for the team to ensure they exercise what you've taught them. You can organize the report as a postvitam,[145] explaining the critical decisions at each step that led to success.

The bulk of the engagement is now complete. Once your embedded assignment concludes, you should remain available for design and code reviews. Keep an eye on the team for the next few months to confirm that they're slowly improving their capacity planning, emergency response, and rollout processes.

[145] In contrast to a postmortem.

PREVIOUS
Chapter 29 - Dealing with Interrupts

NEXT
Chapter 31 - Communication and Collaboration in SRE