Chapter 28 - Accelerating SREs to On-Call and Beyond

Accelerating SREs to On-Call and Beyond

How Can I Strap a Jetpack to My Newbies While Keeping Senior SREs Up to Speed?

Written by Andrew Widdowson
Edited by Shylaja Nukala

You've Hired Your Next SRE(s), Now What?

You've hired new employees into your organization, and they're starting as Site Reliability Engineers. Now you have to train them on the job. Investing up front in the education and technical orientation of new SREs will shape them into better engineers. Such training will accelerate them to a state of proficiency faster, while making their skill set more robust and balanced.

Successful SRE teams are built on trust—in order to maintain a service consistently and globally, you need to trust that your fellow on-callers know how your system works,[135] can diagnose atypical system behaviors, are comfortable with reaching out for help, and can react under pressure to save the day. It is essential, then, but not sufficient, to think of SRE education through the lens of, "What does a newbie need to learn to go on-call?" Given the requirements regarding trust, you also need to ask questions like:

- How can my existing on-callers assess the readiness of the newbie for on-call?
- How can we harness the enthusiasm and curiosity in our new hires to make sure that existing SREs benefit from it?
- What activities can I commit our team to that benefit everyone's education, but that everyone will like?

Students have a wide range of learning preferences. Recognizing that you will hire people who have a mix of these preferences, it would be shortsighted to only cater to one style at the expense of the others. Thus, there is no style of education that works best to train new SREs, and there is certainly no one magic formula that will work for all SRE teams. Table 28-1 lists recommended training practices (and their corresponding anti-patterns) that are well known to SRE at Google. These practices represent a wide range of options available for making your team well educated in SRE concepts, both now and on an ongoing basis.

Table 28-1. SRE education practices

Recommended patterns	Anti-patterns
Designing concrete, sequential learning experiences for students to follow	Deluging students with menial work (e.g., alert/ticket triage) to train them; "trial by fire"
Encouraging reverse engineering, statistical thinking, and working from fundamental principles	Training strictly through operator procedures, checklists, and playbooks
Celebrating the analysis of failure by suggesting postmortems for students to read	Treating outages as secrets to be buried in order to avoid blame
Creating contained but realistic breakages for students to fix using real monitoring and tooling	Having the first chance to fix something only occur after a student is already on-call
Role-playing theoretical disasters as a group, to	Creating experts on the team whose techniques

Recommended patterns	Anti-patterns
intermingle a team's problem-solving approaches	and knowledge are compartmentalized
Enabling students to shadow their on-call rotation early, comparing notes with the on-caller	Pushing students into being primary on-call before they achieve a holistic understanding of their service
Pairing students with expert SREs to revise targeted sections of the on-call training plan	Treating on-call training plans as static and untouchable except by subject matter experts
Carving out nontrivial project work for students to undertake, allowing them to gain partial ownership in the stack	Awarding all new project work to the most senior SREs, leaving junior SREs to pick up the scraps

The rest of this chapter presents major themes that we have found to be effective in accelerating SREs to on-call and beyond. These concepts can be visualized in a blueprint for bootstrapping SREs (Figure 28-1).

Figure 28-1. A blueprint for bootstrapping an SRE to on-call and beyond

This illustration captures best practices that SRE teams can pick from to help bootstrap new members, while keeping senior talent fresh. From the many tools here, you can pick and choose the activities that best suit your team.

The illustration has two axes:

- The x-axis represents the *spectrum between different types of work*, ranging from abstract to applied activities.
- The y-axis represents *time*. Read from the top down, new SREs have very little knowledge about the systems and services they'll be responsible for, so postmortems detailing how these systems have failed in the past are a good starting point. New SREs can also try to reverse engineer systems from fundamentals, since they're starting from zero. Once they understand more about their systems and have done some hands-on work, SREs are ready to shadow on-call and to start mending incomplete or out-of-date documentation.

Tips for interpreting this illustration:

- *Going on-call* is a milestone in a new SRE's career, after which point learning becomes a lot more nebulous, undefined, and self-directed—hence the dashed lines around activities that happen at or after the SRE goes on-call.
- The triangular shape of *project work & ownership* indicates that project work starts out small and builds over time, becoming more complex and likely continuing well after going on-call.
- Some of these activities and practices are very abstract/passive, and some are very applied/active. A few activities are mixes of both. It's good to have a variety of learning modalities to suit different learning styles.
- For maximum effect, training activities and practices should be appropriately paced: some are appropriate to undertake straightaway, some should happen right before an SRE officially goes on-call, and some should be continual and ongoing even by seasoned SREs. *Concrete learning experiences* should happen for the entire time leading up to the SRE going on-call.

Initial Learning Experiences: The Case for Structure Over Chaos

As discussed elsewhere in this book, SRE teams undertake a natural mix of proactive[136] and reactive[137] work. It should be a strong goal of every SRE team to contain and reduce reactive work through ample proactivity, and the approach you take to onboarding your newbie(s) should be no exception. Consider the following all-too-common, but sadly suboptimal, onboarding process:

> John is the newest member of the FooServer SRE team. Senior SREs on this team are tasked with a lot of grunt work, such as responding to tickets, dealing with alerts, and performing tedious binary rollouts. On John's first day on the job, he is assigned all new incoming tickets. He is told that he can ask any member of the SRE team to help him obtain the background necessary to decipher a ticket. 'Sure, there will be a lot of upfront learning that you'll have to do,' says John's manager. 'But eventually you'll get much faster at these tickets. One day, it will just *click* and you'll know a lot about all of the tools we use, the procedures we follow, and the systems we maintain.' A senior team member comments, 'We're throwing you in the deep end of the pool here.'

This "trial by fire" method of orienting one's newbies is often born out of a team's current environment; ops-driven, reactive SRE teams "train" their newest members by making them…well, react! Over and over again. If you're lucky, the engineers who are already good at navigating ambiguity will crawl out of the hole you've put them in. But chances are, this strategy has alienated several capable engineers. While such an approach may eventually produce great operations employees, its results will fall short of the mark. The trial-by-fire approach also presumes that many or most aspects of a team can be taught strictly by doing, rather than by reasoning. If the set of work one encounters in a tickets queue will adequately provide training for said job, then this is not an SRE position.

SRE students will have questions like the following:

- What am I working on?
- How much progress have I made?
- When will these activities accumulate enough experience for me to go on-call?

Making the jump from a previous company or university, while changing job roles (from traditional software engineer or traditional systems administrator) to this nebulous *Site Reliability Engineer* role is often enough to knock students' confidence down several times. For more introspective personalities (especially regarding questions #2 and #3), the uncertainties incurred by nebulous or less-than-clear answers can lead to slower development or retention problems. Instead, consider the approaches outlined in the following sections. These suggestions are as concrete as any ticket or alert, but they are also sequential, and thus far more rewarding.

Learning Paths That Are Cumulative and Orderly

Put some amount of learning order into your system(s) so that your new SREs see a path before them. Any type of training is better than random tickets and interrupts, but do make a conscious effort to combine the right mix of theory and application: abstract concepts that will recur multiple times in a newbie's journey should be frontloaded in their education, while the student should also receive hands-on experience as soon as practically possible.

Learning about your stack(s) and subsystem(s) requires a starting point. Consider whether it makes more sense to group trainings together by similarity of purpose, or by normal-case order of execution. For example, if your team is responsible for a real-time, user-facing serving stack, consider a curriculum order like the following:

1) How a query enters the system

> Networking and datacenter fundamentals, frontend load balancing, proxies, etc.

2) Frontend serving

Application frontend(s), query logging, user experience SLO(s), etc.

3) Mid-tier services

Caches, backend load balancing

4) Infrastructure

Backends, infrastructure, and compute resources

5) Tying it all together

Debugging techniques, escalation procedures, and emergency scenarios

How you choose to present the learning opportunities (informal whiteboard chats, formal lectures, or hands-on discovery exercises) is up to you and the SREs helping you structure, design, and deliver training. The Google Search SRE team structures this learning through a document called the "on-call learning checklist." A simplified section of an on-call learning checklist might look like the following:

The Results Mixing Server ("Mixer")

Frontended by: Frontend server

Backends called: Results Retrieval Server,
Geolocation Server, Personalization Database

SRE experts: Sally W, Dave K, Jen P

Developer contacts: Jim T, *results-team@*

Know before moving on:

- Which clusters have Mixer deployed
- How to roll back a Mixer release
- Which backends of Mixer are considered "critical path" and why

Read and understand the following docs:

- Results Mixing Overview: "Query execution" section
- Results Mixing Overview: "Production" section
- Playbook: How to Roll Out a New Results Mixing Server
- A Performance Analysis of Mixer

Comprehension questions:

- Q: How does the release schedule change if a company holiday occurs on the normal release build day?
- Q: How can you fix a bad push of the geolocation dataset?

Note that the preceding section does not directly encode procedures, diagnostic steps, or playbooks; instead, it's a relatively future-proof write-up focusing strictly on enumerating expert contacts, highlighting the most useful documentation resources, establishing basic knowledge you must gather and internalize, and asking probing questions that can only be answered once that basic knowledge has been absorbed. It also provides concrete outcomes, so that the student knows what kinds of knowledge and

skills they will have gained from completing this section of the learning checklist.

It's a good idea for all interested parties to get a sense of how much information the trainee is retaining. While this feedback mechanism perhaps doesn't need to be as formal as a quiz, it is a good practice to have complete bits of homework that pose questions about how your service(s) work. Satisfactory answers, checked by a student's mentor, are a sign that learning should continue to the next phase. Questions about the inner workings of your service might look similar to the following:

- Which backends of this server are considered "in the critical path," and why?
- What aspects of this server could be simplified or automated?
- Where do you think the first bottleneck is in this architecture? If that bottleneck were to be saturated, what steps could you take to alleviate it?

Depending on how the access permissions are configured for your service, you can also consider implementing a tiered access model. The first tier of access would allow your student read-only access to the inner workings of the components, and a later tier would permit them to mutate the production state. Completing sections of the on-call learning checklist satisfactorily would earn the student progressively deeper access to the system. The Search SRE team calls these attained levels "powerups"[138] on the route to on-call, as trainees are eventually added into the highest level of systems access.

Targeted Project Work, Not Menial Work

SREs are problem solvers, so give them a hearty problem to solve! When starting out, having even a minor sense of ownership in the team's service can do wonders for learning. In the reverse, such ownership can also make great inroads for trust building among senior colleagues, because they will approach their junior colleague to learn about the new component(s) or processes. Early opportunities for ownership are standard across Google in general: all engineers are given a starter project that's meant to provide a tour through the infrastructure sufficient to enable them to make a small but useful contribution early. Having the new SRE split time between learning *and* project work will also give them a sense of purpose and productivity, which would not happen if they spent time only on learning *or* project work. Several starter project patterns that seem to work well include:

- Making a trivial user-visible feature change in a serving stack, and subsequently shepherding the feature release all the way through to production. Understanding both the development toolchain and the binary release process encourages empathy for the developers.
- Adding monitoring to your service where there are currently blind spots. The newbie will have to reason with the monitoring logic, while reconciling their understanding of a system with how it actually (mis)behaves.
- Automating a pain point that isn't quite painful enough to have been automated already, providing the new SRE with an appreciation for the value SREs place on removing toil from our day-to-day operations.

Creating Stellar Reverse Engineers and Improvisational Thinkers

We can propose a set of guidelines for *how* to train new SREs, but *what* should we train them on? Training material will depend on the technologies being used on the job, but the more important question is: what kind of engineers are we trying to create? At the scale and complexity at which SREs operate, they cannot afford to merely be operations-focused, traditional system administrators. In addition to having a large-scale engineering mindset, SREs should exhibit the following characteristics:

- In the course of their jobs, they will come across systems they've never seen before, so they need to have *strong reverse engineering skills*.
- At scale, there will be anomalies that are hard to detect, so they'll need the ability to *think*

statistically, rather than procedurally, to uncloak problems.
- When standard operating procedures break down, they'll need to be able to *improvise fully*.

Let's examine these attributes further, so that we can understand how to equip our SREs for these skills and behaviors.

Reverse Engineers: Figuring Out How Things Work

Engineers are curious about how systems they've never seen before work—or, more likely, how the current versions of systems they used to know quite well work. By having a baseline understanding of how systems work at your company, along with a willingness to dig deep into the debugging tools, RPC boundaries, and logs of your binaries to unearth their flows, SREs will become more efficient at homing in on unexpected problems in unexpected system architectures. Teach your SREs about the diagnostic and debugging surfaces of your applications and have them practice drawing inferences from the information these surfaces reveal, so that such behavior becomes reflexive when dealing with future outages.

Statistical and Comparative Thinkers: Stewards of the Scientific Method Under Pressure

You can think of an SRE's approach to incident response for large-scale systems as navigating through a massive decision tree unfolding in front of them. In the limited time window afforded by the demands of incident response, the SRE can take a few actions out of hundreds with the goal of mitigating the outage, either in the short term or the long term. Because time is often of the utmost importance, the SRE has to effectively and efficiently prune this decision tree. The ability to do so is partially gained through experience, which only comes with time and exposure to a breadth of production systems. This experience must be paired with careful construction of hypotheses that, when proven or disproven, even further narrow down that decision space. Put another way, tracking down system breakages is often akin to playing a game of "which of these things is not like the other?" where "things" might entail kernel version, CPU architecture, binary version(s) in your stack, regional traffic mix, or a hundred other factors. Architecturally, it's the team's responsibility to ensure all of these factors can be controlled for and individually analyzed and compared. However, we should also train our newest SREs to become good analysts and comparators from their earliest moments on the job.

Improv Artists: When the Unexpected Happens

You try out a fix for the breakage, but it doesn't work. The developer(s) behind the failing system are nowhere to be found. What do you do now? You improvise! Learning multiple tools that can solve parts of your problem allows you to practice defense in depth in your own problem-solving behaviors. Being too procedural in the face of an outage, thus forgetting your analytical skills, can be the difference between getting stuck and finding the root cause. A case of bogged-down troubleshooting can be further compounded when an SRE brings too many untested assumptions about the cause of an outage into their decision making. Demonstrating that there are many analytical traps that SREs can fall into, which require "zooming out" and taking a different approach to resolution, is a valuable lesson for SREs to learn early on.

Given these three aspirational attributes of high-performing SREs, what courses and experiences can we provide new SREs in order to send them along a path in the right direction? You need to come up with your own course content that embodies these attributes, in addition to the other attributes specific to your SRE culture. Let's consider one class that we believe hits all of the aforementioned points.

Tying This Together: Reverse Engineering a Production Service

> When it came time to learn [part of the Google Maps stack], [a new SRE] asked if, rather than passively having someone explain the service, she could do this herself—learning everything via reverse engineering class techniques, and having the rest of us correct her/fill in the blanks for whatever she missed or got wrong. The result? Well, it was probably more correct and useful than it would have been if *I'd* given the talk, and I've been on-call for this for over 5 years!
>
> Paul Cowan, *Google Site Reliability Engineer*

One popular class we offer at Google is called "Reverse Engineering a Production Service (without help from its owners)." The problem scenario presented appears simple at first. The entire Google News Team —SRE, Software Engineers, Product Management, and so forth—has gone on a company trip: a cruise of the Bermuda Triangle. We haven't heard from the team for 30 days, so our students are the newly appointed Google News SRE Team. They need to figure out how the serving stack works from end-to-end in order to commandeer it and keep it running.

After being given this scenario, the students are led through interactive, purpose-driven exercises in which they trace the inbound path of their web browser's query through Google's infrastructure. At each stage in the process, we emphasize that it is important to learn multiple ways to discover the connectivity between production servers, so that connections are not missed. In the middle of the class, we challenge the students to find another endpoint for the incoming traffic, demonstrating that our initial assumption was too narrowly scoped. We then challenge our students to find other ways into the stack. We exploit the highly instrumented nature of our production binaries, which self-report their RPC connectivity, as well as our available white-box and black-box monitoring, to determine which path(s) users' queries take.[139] Along the way, we build a system diagram and also discuss components that are shared infrastructure that our students are likely to see again in the future.

At the end of the class, the students are charged with a task. Each student returns to their home team and asks a senior SRE to help them select a stack or slice of a stack for which they'll be on-call. Using the skills learned in classes, the student then diagrams that stack on their own and presents their findings to the senior SRE. Undoubtedly the student will miss a few subtle details, which will make for a good discussion. It's also likely that the senior SRE will learn something from the exercise as well, exposing drifts in their prior understanding of the ever-changing system. Because of the rapid change of production systems, it is important that your team welcome any chance to refamiliarize themselves with a system, including by learning from the newest, rather than oldest, members of the team.

Five Practices for Aspiring On-Callers

Being on-call is not the single most important purpose of any SRE, but production engineering responsibilities usually do involve some kind of urgent notification coverage. Someone who is capable of responsibly taking on-call is someone who understands the system that they work on to a reasonable depth and breadth. So we'll use "able to take on-call" as a useful proxy for "knows enough and can figure out the rest."

A Hunger for Failure: Reading and Sharing Postmortems

> Those who cannot remember the past are condemned to repeat it.
>
> George Santayana, *philosopher and essayist*

Postmortems (see Postmortem Culture: Learning from Failure) are an important part of continuous improvement. They are a blame-free way of getting at the many root causes of a significant or visible outage. When writing a postmortem, keep in mind that its most appreciative audience might be an engineer who hasn't yet been hired. Without radical editing, subtle changes can be made to our best postmortems to

make them "teachable" postmortems.

Even the best postmortems aren't helpful if they languish in the bottom of a virtual filing cabinet. It then follows that your team should collect and curate valuable postmortems to serve as educational resources for future newbies. Some postmortems are rote, but "teachable postmortems" that provide insights into structural or novel failures of large-scale systems are as good as gold for new students.

Ownership of postmortems isn't limited just to authorship. It's a point of pride for many teams to have survived and documented their largest outages. Collect your best postmortems and make them prominently available for your newbies—in addition to interested parties from related and/or integrating teams—to read. Ask related teams to publish their best postmortems where you can access them.

Some SRE teams at Google run "postmortem reading clubs" where fascinating and insightful postmortems are circulated, pre-read, and then discussed. The original author(s) of the postmortem can be the guest(s) of honor at the meeting. Other teams organize "tales of fail" gatherings where the postmortem author(s) semiformally present, recounting the outage and effectively driving the discussion themselves.

Regular readings or presentations on outages, including trigger conditions and mitigation steps, do wonders for building a new SRE's mental map and understanding of production and on-call response. Postmortems are also excellent fuel for future abstract disaster scenarios.

Disaster Role Playing

> Once a week we have a meeting where a victim is chosen to be on the spot in front of the group, and a scenario—often a real one taken from the annals of Google history—is thrown at him or her. The victim, whom I think of as a game show contestant, tells the game show host what s/he would do or query to understand or solve the problem, and the host tells the victim what happens with each action or observation. It's like *SRE Zork*. You are in a maze of twisty monitoring consoles, all alike. You must save innocent users from slipping into the Chasm of Excessive Query Latency, save datacenters from Near-Certain Meltdown, and spare us all the embarrassment of Erroneous Google Doodle Display.
>
> Robert Kennedy, former Site Reliability Engineer for Google Search and healthcare.gov[140]

When you have a group of SREs of wildly different experience levels, what can you do to bring them all together, and enable them to learn from each other? How do you impress the SRE culture and problem-solving nature of your team upon a newbie, while also keeping grizzled veterans apprised of new changes and features in your stack? Google SRE teams address these challenges through a time-honored tradition of regular disaster role playing. Among other names, this exercise is commonly referred to as "Wheel of Misfortune" or "Walk the Plank." The sense of humorous danger such titles lend the exercise makes it less intimidating to freshly hired SREs.

At its best, these exercises become a weekly ritual in which every member of the group learns something. The formula is straightforward and bears some resemblance to a tabletop RPG (Role Playing Game): the "game master" (GM) picks two team members to be primary and secondary on-call; these two SREs join the GM at the front of the room. An incoming page is announced, and the on-call team responds with what they would do to mitigate and investigate the outage.

The GM has carefully prepared a scenario that is about to unfold. This scenario might be based upon a previous outage for which the newer team members weren't around or that older team members have forgotten. Or perhaps the scenario is a foray into a hypothetical breakage of a new or soon-to-be-launched feature in the stack, rendering all members of the room equally unprepared to grapple with the situation. Better still, a coworker might find a new and novel breakage in production, and today's scenario expands on this new threat.

Over the next 30–60 minutes, the primary and secondary on-callers attempt to root-cause the issue. The GM happily provides additional context as the problem unfolds, perhaps informing the on-callers (and their audience) of what the graphs on their monitoring dashboard might look like during the outage. If the incident requires escalation outside of the home team, the GM pretends to be a member of that other team for the purposes of the scenario. No virtual scenario will be perfect, so at times the GM may have to steer participants back on track by redirecting the on-callers away from red herrings, introducing urgency and clarity by adding other stimuli,[141] or asking urgent and pointed questions.[142]

When your disaster RPG is successful, everyone will have learned something: perhaps a new tool or trick, a different perspective on how to solve a problem, or (especially gratifying to new team members) a validation that you could have solved this week's problem if you had been picked. With some luck, this exercise will inspire teammates to eagerly look forward to next week's adventure or to ask to become the game master for an upcoming week.

Break Real Things, Fix Real Things

A newbie can learn much about SRE by reading documentation, postmortems, and taking trainings. Disaster role playing can help get a newbie's mind into the game. However, the experience derived from hands-on experience breaking and/or fixing *real* production systems is even better. There will be plenty of time for hands-on experience once a newbie has gone on-call, but such learning should happen *before* a new SRE reaches that point. Therefore, provide for such hands-on experiences much earlier in order to develop the student's reflexive responses for using your company's tooling and monitoring to approach a developing outage.

Realism is paramount in these interactions. Ideally, your team has a stack that is multihomed and provisioned in such a way that you have at least one instance you can divert from live traffic and temporarily loan to a learning exercise. Alternatively, you might have a smaller, but still fully featured, staging or QA instance of your stack that can be borrowed for a short time. If possible, subject the stack to synthetic load that approximates real user/client traffic, in addition to resource consumption, if possible.

The opportunities for learning from a real production system under synthetic load are abundant. Senior SREs will have experienced all sorts of troubles: misconfigurations, memory leaks, performance regressions, crashing queries, storage bottlenecks, and so forth. In this realistic but relatively risk-free environment, proctors can manipulate the job set in ways that alter the behavior of the stack, forcing new SREs to find differences, determine contributing factors, and ultimately repair systems to restore appropriate behavior.

As an alternative to the overhead of asking a senior SRE to carefully plan a specific type of breakage that the new SRE(s) must repair, you can also work in the opposite direction with an exercise that may also increase participation from the entire team: work from a known good configuration and slowly impair the stack at selected bottlenecks, observing upstream and downstream efforts through your monitoring. This exercise is valued by the Google Search SRE team, whose version of this exercise is called "Let's burn a search cluster to the ground!" The exercise proceeds as follows:

1. As a group, we discuss what observable performance characteristics might change as we cripple the stack.
2. Before inflicting the planned damage, we poll the participants for their guesses and reasoning about their predictions about how the system will react.
3. We validate assumptions and justify the reasoning behind the behaviors we see.

This exercise, which we perform on a quarterly basis, shakes out new bugs that we eagerly fix, because our systems do not always degrade as gracefully as we would expect.

Documentation as Apprenticeship

Many SRE teams maintain an "on-call learning checklist," which is an organized reading and comprehension list of the technologies and concepts relevant to the system(s) they maintain. This list must be internalized by a student before they're eligible to serve as a shadow on-caller. Take a moment to revisit the example on-call learning checklist in Table 28-2. The learning checklist serves different purposes for different people:

- **To the student**:
    - This doc helps establish the boundaries of the system their team supports.
    - By studying this list, the student gains a sense of what systems are most important and why. When they understand the information therein, they can move on to other topics they need to learn, rather than dwelling on learning esoteric details that can be learned over time.
- **To mentors and managers**: Student progress through the learning checklist can be observed. The checklist answers questions such as:
    - What sections are you working on today?
    - What sections are the most confusing?
- **To all team members**: The doc becomes a social contract by which (upon mastery) the student joins the ranks of on-call. The learning checklist sets the standard that all team members should aspire to and uphold.

In a rapidly changing environment, documentation can fall out of date quickly. Outdated documentation is less of a problem for senior SREs who are already up to speed, because they keep state on the world and its changes in their own heads. Newbie SREs are much more in need of up-to-date documentation, but may not feel empowered or knowledgeable enough to make changes. When designed with just the right amount of structure, on-call documentation can become an adaptable body of work that harnesses newbie enthusiasm and senior knowledge to keep everyone fresh.

In Search SRE, we anticipate the arrival of new team member(s) by reviewing our on-call learning checklist, and sorting its sections by how up-to-date they are. As the new team member arrives, we point them to the overall learning checklist, but also task them with overhauling one or two of the most outdated sections. As you can see in Table 28-2, we label the senior SRE and developer contacts for each technology. We encourage the student to make an early connection with those subject matter experts, so that they might learn the inner workings of the selected technology directly. Later, as they become more familiar with the scope and tone of the learning checklist, they are expected to contribute a revised learning checklist section, which must be peer-reviewed by one or more senior SREs that are listed as experts.

Shadow On-Call Early and Often

Ultimately, no amount of hypothetical disaster exercises or other training mechanisms will fully prepare an SRE for going on-call. At the end of the day, tackling real outages will always be more beneficial from a learning standpoint than engaging with hypotheticals. Yet it's unfair to make newbies wait until their first real page to have a chance to learn and retain knowledge.

After the student has made their way through all system fundamentals (by completing, for example, an on-call learning checklist), consider configuring your alerting system to copy incoming pages to your newbie, at first only during business hours. Rely on their curiosity to lead the way. These "shadow" on-call shifts are a great way for a mentor to gain visibility into a student's progress, and for a student to gain visibility into the responsibilities of being on-call. By arranging for the newbie to shadow multiple members of their

team, the team will become increasingly comfortable with the thought of this person entering the on-call rotation. Instilling confidence in this manner is an effective method of building trust, allowing more senior members to detach when they aren't on-call, thus helping to avoid team burnout.

When a page comes in, the new SRE is not the appointed on-caller, a condition which removes any time pressure for the student. They now have a front-row seat to the outage while it unfolds, rather than after the issue is resolved. It may be that the student and the primary on-caller share a terminal session, or sit near each other to readily compare notes. At a time of mutual convenience after the outage is complete, the on-caller can review the reasoning and processes followed for the student's benefit. This exercise will increase the shadow on-caller's retention of what actually occurred.

Tip

Should an outage occur for which writing a postmortem is beneficial, the on-caller should include the newbie as a coauthor. *Do not dump the writeup solely on the student, because it could be mislearned that postmortems are somehow grunt work to be passed off on those most junior. It would be a mistake to create such an impression.*

Some teams will also include a final step: having the experienced on-caller "reverse shadow" the student. The newbie will become primary on-call and own all incoming escalations, but the experienced on-caller will lurk in the shadows, independently diagnosing the situation without modifying any state. The experienced SRE will be available to provide active support, help, validation, and hints as necessary.

On-Call and Beyond: Rites of Passage, and Practicing Continuing Education

As comprehension increases, the student will reach a point in their career at which they are capable of reasoning through most of the stack comfortably, and can improvise their way through the rest. At this point, they should go on-call for their service. Some teams create a final exam of sorts that tests their students one last time before bestowing them with on-call powers and responsibilities. Other new SREs will submit their completion of the on-call learning checklist as evidence that they are ready. Regardless of how you gate this milestone, going on-call is a rite of passage and it should be celebrated as a team.

Does learning stop when a student joins the ranks of on-call? Of course not! To remain vigilant as SREs, your team will always need to be active and aware of changes to come. While your attention is elsewhere, portions of your stack may be rearchitected and extended, leaving your team's operational knowledge as historic at best.

Set up a regular learning series for your whole team, where overviews of new and upcoming changes to your stack are given as presentations by the SREs who are shepherding the changes, who can co-present with developers as needed. If you can, record the presentations so that you can build a training library for future students.

With some practice, you'll gain much timely involvement from both SREs within your team and developers who work closely with your team, all while keeping everyone's minds fresh about the future. There are other venues for educational engagement, too: consider having SREs give talks to your developer counterparts. The better your development peers understand your work and the challenges your team faces, the easier it will be to reach fully informed decisions on later projects.

Closing Thoughts

An upfront investment in SRE training is absolutely worthwhile, both for the students eager to grasp their production environment and for the teams grateful to welcome students into the ranks of on-call. Through the use of applicable practices outlined in this chapter, you will create well-rounded SREs faster, while sharpening team skills in perpetuity. How you apply these practices is up to you, but the charge is clear: as

SRE, you have to scale your humans faster than you scale your machines. Good luck to you and your teams in creating a culture of learning and teaching!

[135]And doesn't work!

[136]Examples of proactive SRE work include software automation, design consulting, and launch coordination.

[137]Examples of reactive SRE work include debugging, troubleshooting, and handling on-call escalations.

[138]A nod to video games of yesteryear.

[139]This "follow the RPC" approach also works well for batch/pipeline systems; start with the operation that kicks off the system. For batch systems, this operation could be data arriving that needs to be processed, a transaction that needs to be validated, or many other events.

[140]See "Life in the Trenches of healthcare.gov".

[141]For example: "You're getting paged by another team that brings you more information. Here's what they say…"

[142]For example: "We're losing money quickly! How could you stop the bleeding in the short term?"

previous

Part IV - Management

next

Chapter 29 - Dealing with Interrupts