Software Engineering in SRE

Written by Dave Helstroom and Trisha Weir with Evan Leonard and Kurt Delimon
Edited by Kavita Guliani

Ask someone to name a Google software engineering effort and they'll likely list a consumer-facing product like Gmail or Maps; some might even mention underlying infrastructure such as Bigtable or Colossus. But in truth, there is a massive amount of behind-the-scenes software engineering that consumers never see. A number of those products are developed within SRE.

Google's production environment is—by some measures—one of the most complex machines humanity has ever built. SREs have firsthand experience with the intricacies of production, making them uniquely well suited to develop the appropriate tools to solve internal problems and use cases related to keeping production running. The majority of these tools are related to the overall directive of maintaining uptime and keeping latency low, but take many forms: examples include binary rollout mechanisms, monitoring, or a development environment built on dynamic server composition. Overall, these SRE-developed tools are full-fledged software engineering projects, distinct from one-off solutions and quick hacks, and the SREs who develop them have adopted a product-based mindset that takes both internal customers and a roadmap for future plans into account.

Why Is Software Engineering Within SRE Important?

In many ways, the vast scale of Google production has necessitated internal software development, because few third-party tools are designed at sufficient scale for Google's needs. The company's history of successful software projects has led us to appreciate the benefits of developing directly within SRE.

SREs are in a unique position to effectively develop internal software for a number of reasons:

- The breadth and depth of Google-specific production knowledge within the SRE organization allows its engineers to design and create software with the appropriate considerations for dimensions such as scalability, graceful degradation during failure, and the ability to easily interface with other infrastructure or tools.

- Because SREs are embedded in the subject matter, they easily understand the needs and requirements of the tool being developed.

- A direct relationship with the intended user—fellow SREs—results in frank and high-signal user feedback. Releasing a tool to an internal audience with high familiarity with the problem space means that a development team can launch and iterate more quickly. Internal users are typically more understanding when it comes to minimal UI and other alpha product issues.

From a purely pragmatic standpoint, Google clearly benefits from having engineers with SRE experience developing software. By deliberate design, the growth rate of SRE-supported services exceeds the growth rate of the SRE organization; one of SRE's guiding principles is that "team size should not scale directly with service growth." Achieving linear team growth in the face of exponential service growth requires perpetual automation work and efforts to streamline tools, processes, and other aspects of a service that introduce inefficiency into the day-to-day operation of production. Having the people with direct experience running production systems developing the tools that will ultimately contribute to uptime and latency goals makes a lot of sense.

On the flip side, individual SREs, as well as the broader SRE organization, also benefit from SRE-driven software development.

Fully fledged software development projects within SRE provide career development opportunities for SREs, as well as an outlet for engineers who don't want their coding skills to get rusty. Long-term project work provides much-needed balance to interrupts and on-call work, and can provide job satisfaction for engineers who want their careers to maintain a balance between software engineering and systems engineering.

Beyond the design of automation tools and other efforts to reduce the workload for engineers in SRE, software development projects can further benefit the SRE organization by attracting and helping to retain engineers with a broad variety of skills. The desirability of team diversity is doubly true for SRE, where a variety of backgrounds and problem-solving approaches can help prevent blind spots. To this end, Google always strives to staff its SRE teams with a mix of engineers with traditional software development experience and engineers with systems engineering experience.

Auxon Case Study: Project Background and Problem Space

This case study examines Auxon, a powerful tool developed within SRE to automate capacity planning for services running in Google production. To best understand how Auxon was conceived and the problems it addresses, we'll first examine the problem space associated with capacity planning, and the difficulties that traditional approaches to this task present for services at Google and across the industry as a whole. For more context on how Google uses the terms *service* and *cluster*, see The Production Environment at Google, from the Viewpoint of an SRE.

Traditional Capacity Planning

There are myriad tactics for capacity planning of compute resources (see [Hix15a]), but the majority of these approaches boil down to a *cycle* that can be approximated as follows:

1) Collect demand forecasts.

How many resources are needed? When and where are these resources needed?

- Uses the best data we have available today to plan into the future
- Typically covers anywhere from several quarters to years

2) Devise build and allocation plans.

Given this forecasted outlook, what's the best way to meet this demand with additional supply of resources? How much supply, and in what locations?

3) Review and sign off on plan.

Is the forecast reasonable? Does the plan line up with budgetary, product-level, and technical considerations?

4) Deploy and configure resources.

Once resources eventually arrive (potentially in phases over the course of some defined period of time), which services get to use the resources? How do I make typically lower-level resources (CPU, disk, etc.) useful for services?

It bears stressing that capacity planning is a neverending *cycle*: assumptions change, deployments slip, and budgets are cut, resulting in revision upon revision of The Plan. And each revision has trickle-down effects that must propagate throughout the plans of all subsequent quarters. For example, a shortfall this quarter must be made up in future quarters. Traditional capacity planning uses demand as a key driver, and manually shapes supply to fit demand in response to each change.

Brittle by nature

Traditional capacity planning produces a resource allocation plan that can be disrupted by any seemingly minor change. For example:

- A service undergoes a decrease in efficiency, and needs more resources than expected to serve the same demand.
- Customer adoption rates increase, resulting in an increase in projected demand.
- The delivery date for a new cluster of compute resources slips.
- A product decision about a performance goal changes the shape of the required service deployment (the service's footprint) and the amount of required resources.

Minor changes require cross-checking the entire allocation plan to make sure that the plan is still feasible; larger changes (such as delayed resource delivery or product strategy changes) potentially require re-creating the plan from scratch. A delivery slippage in a single cluster might impact the redundancy or latency requirements of multiple services: resource allocations in other clusters must be increased to make up for the slippage, and these and any other changes would have to propagate throughout the plan.

Also, consider that the capacity plan for any given quarter (or other time frame) is based on the expected outcome of the capacity plans of previous quarters, meaning that a change in any one quarter results in work to update subsequent quarters.

Laborious and imprecise

For many teams, the process of collecting the data necessary to generate demand forecasts is slow and error-prone. And when it is time to find capacity to meet this future demand, not all resources are equally suitable. For example, if latency requirements mean that a service must commit to serve user demand on the same continent as the user, obtaining additional resources in North America won't

alleviate a capacity shortfall in Asia. Every forecast has *constraints*, or parameters around how it can be fulfilled; constraints are fundamentally related to intent, which is discussed in the next section.

Mapping constrained resource requests into allocations of actual resources from the available capacity is equally slow: it's both complex and tedious to bin pack requests into limited space by hand, or to find solutions that fit a limited budget.

This process may already paint a grim picture, but to make matters worse, the tools it requires are typically unreliable or cumbersome. Spreadsheets suffer severely from scalability problems and have limited error-checking abilities. Data becomes stale, and tracking changes becomes difficult. Teams often are forced to make simplifying assumptions and reduce the complexity of their requirements, simply to render maintaining adequate capacity a tractable problem.

When service owners face the challenges of fitting a series of requests for capacity from various services into the resources available to them, in a manner that meets the various constraints a service may have, additional imprecision ensues. Bin packing is an NP-hard problem that is difficult for human beings to compute by hand. Furthermore, the capacity request from a service is generally an inflexible set of demand requirements: X cores in cluster Y. The reasons why X cores or Y cluster are needed, and any degrees of freedom around those parameters, are long lost by the time the request reaches a human trying to fit a list of demands into available supply.

The net result is a massive expenditure of human effort to come up with a bin packing that is approximate, at best. The process is brittle to change, and there are no known bounds on an optimal solution.

Our Solution: Intent-Based Capacity Planning

Specify the requirements, not the implementation.

At Google, many teams have moved to an approach we call *Intent-based Capacity Planning*. The basic premise of this approach is to programmatically encode the dependencies and parameters (*intent*) of a service's needs, and use that encoding to autogenerate an allocation plan that details which resources go to which service, in which cluster. If demand, supply, or service requirements change, we can simply autogenerate a new plan in response to the changed parameters, which is now the new best distribution of resources.

With a service's true requirements and flexibility captured, the capacity plan is now dramatically more nimble in the face of change, and we can reach an optimal solution that meets as many parameters as possible. With bin packing delegated to computers, human toil is drastically reduced, and service

owners can focus on high-order priorities like SLOs, production dependencies, and service infrastructure requirements, as opposed to low-level scrounging for resources.

As an added benefit, using computational optimization to map from intent to implementation achieves much greater precision, ultimately resulting in cost savings to the organization. Bin packing is still far from a solved problem, because certain types are still considered NP-hard; however, today's algorithms can solve to a known optimal solution.

Intent-Based Capacity Planning

Intent is the rationale for how a service owner wants to run their service. Moving from concrete resource demands to motivating reasons in order to arrive at the true capacity planning intent often requires several layers of abstraction. Consider the following chain of abstraction:

1) "I want 50 cores in clusters X, Y, and Z for service Foo."

> This is an explicit resource request. But…*why do we need this many resources specifically in these particular clusters?*

2) "I want a 50-core footprint in any 3 clusters in geographic region YYY for service Foo."

> This request introduces more degrees of freedom and is potentially easier to fulfill, although it doesn't explain the reasoning behind its requirements. But…*why do we need this quantity of resources, and why 3 footprints?*

3) "I want to meet service Foo's demand in each geographic region, and have N + 2 redundancy."

> Suddenly greater flexibility is introduced and we can understand at a more "human" level what happens if service Foo does not receive these resources. But…*why do we need $N + 2$ for service Foo?*

4) "I want to run service Foo at 5 nines of reliability."

> This is a more abstract requirement, and the ramification if the requirement isn't met becomes clear: reliability will suffer. And we have even greater flexibility here: perhaps running at $N + 2$ is not actually sufficient or optimal for this service, and some other deployment plan would be more suitable.

So what level of intent should be used by intent-driven capacity planning? Ideally, all levels of intent should be supported together, with services benefiting the more they shift to specifying intent versus implementation. In Google's experience, services tend to achieve the best wins as they cross to step 3: good degrees of flexibility are available, and the ramifications of this request are in higher-level and understandable terms. Particularly sophisticated services may aim for step 4.

Precursors to Intent

What information do we need in order to capture a service's intent? Enter dependencies, performance metrics, and prioritization.

Dependencies

Services at Google depend on many other infrastructure and user-facing services, and these dependencies heavily influence where a service can be placed. For example, imagine user-facing service Foo, which depends upon Bar, an infrastructure storage service. Foo expresses a requirement that Bar must be located within 30 milliseconds of network latency of Foo. This requirement has important repercussions for where we place both Foo and Bar, and intent-driven capacity planning must take these constraints into account.

Furthermore, production dependencies are nested: to build upon the preceding example, imagine service Bar has its own dependencies on Baz, a lower-level distributed storage service, and Qux, an application management service. Therefore, where we can now place Foo depends on where we can place Bar, Baz, and Qux. A given set of production dependencies can be shared, possibly with different stipulations around intent.

Performance metrics

Demand for one service trickles down to result in demand for one or more other services. Understanding the chain of dependencies helps formulate the general scope of the bin packing problem, but we still need more information about expected resource usage. How many compute resources does service Foo need to serve N user queries? For every N queries of service Foo, how many Mbps of data do we expect for service Bar?

Performance metrics are the glue between dependencies. They convert from one or more higher-level resource type(s) to one or more lower-level resource type(s). Deriving appropriate performance metrics for a service can involve load testing and resource usage monitoring.

Prioritization

Inevitably, resource constraints result in trade-offs and hard decisions: of the many requirements that all services have, which requirements should be sacrificed in the face of insufficient capacity?

Perhaps N + 2 redundancy for service Foo is more important than N + 1 redundancy for service Bar. Or perhaps the feature launch of X is less important than N + 0 redundancy for service Baz.

Intent-driven planning forces these decisions to be made transparently, openly, and consistently. Resource constraints entail the same trade-offs, but all too often, the prioritization can be ad hoc and opaque to service owners. Intent-based planning allows prioritization to be as granular or coarse as needed.

Introduction to Auxon

Auxon is Google's implementation of an intent-based capacity planning and resource allocation solution, and a prime example of an SRE-designed and developed software engineering product: it was built by a small group of software engineers and a technical program manager within SRE over the course of two years. Auxon is a perfect case study to demonstrate how software development can be fostered within SRE.

Auxon is actively used to plan the use of many millions of dollars of machine resources at Google. It has become a critical component of capacity planning for several major divisions within Google.

As a product, Auxon provides the means to collect intent-based descriptions of a service's resource requirements and dependencies. These user intents are expressed as requirements for how the owner would like the service to be provisioned. Requirements might be specified as a request like, "My service must be N + 2 per continent" or "The frontend servers must be no more than 50 ms away from the backend servers." Auxon collects this information either via a user configuration language or via a programmatic API, thus translating human intent into machine-parseable constraints. Requirements can be prioritized, a feature that's useful if resources are insufficient to meet all requirements, and therefore trade-offs must be made. These requirements—the intent—are ultimately represented internally as a giant mixed-integer or linear program. Auxon solves the linear program, and uses the resultant bin packing solution to formulate an allocation plan for resources.

Figure 18-1 and the explanations that follow it outline Auxon's major components.

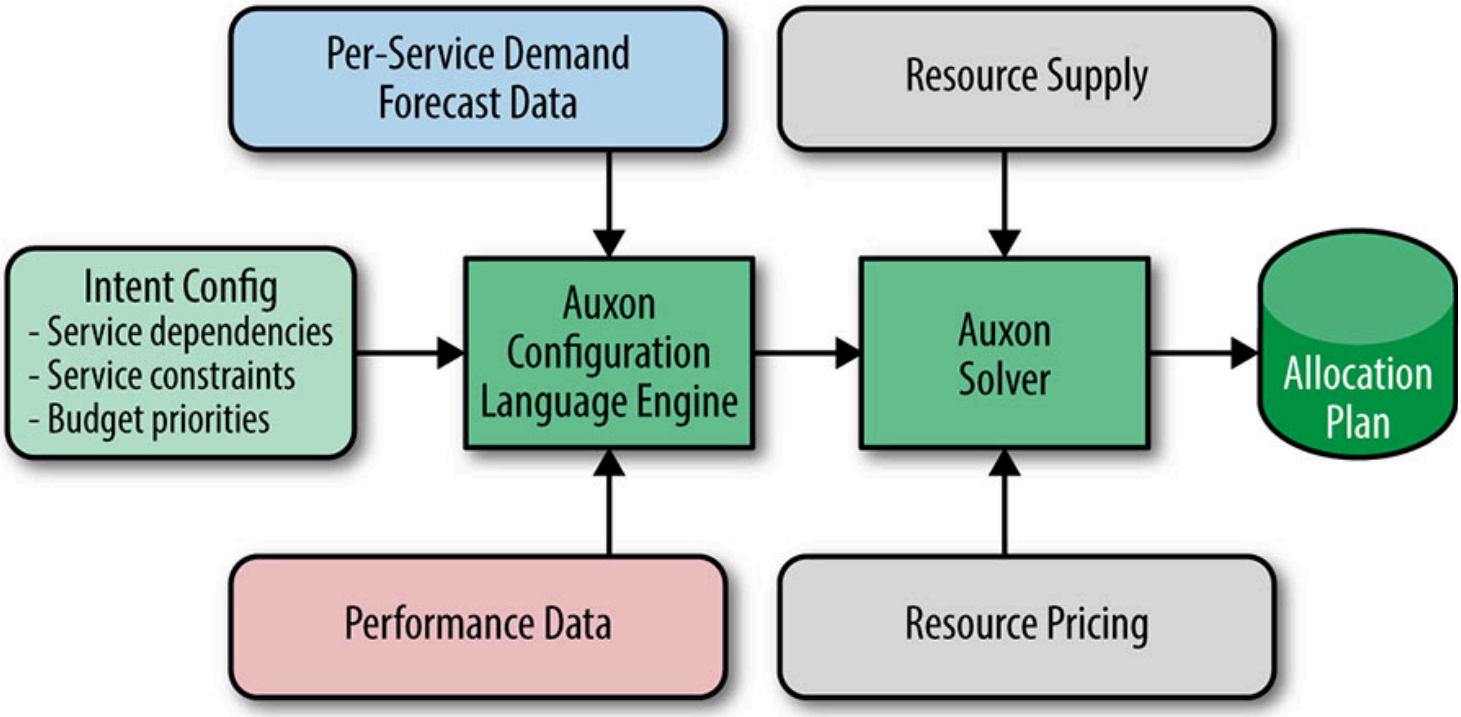

Figure 18-1. The major components of Auxon

Performance Data describes how a service scales: for every unit of demand X in cluster Y, how many units of dependency Z are used? This scaling data may be derived in a number of ways depending on the maturity of the service in question. Some services are load tested, while others infer their scaling based upon past performance.

Per-Service Demand Forecast Data describes the usage trend for forecasted demand signals. Some services derive their future usage from demand forecasts—a forecast of queries per second broken down by continent. Not all services have a demand forecast: some services (e.g., a storage service like Colossus) derive their demand purely from services that depend upon them.

Resource Supply provides data about the availability of base-level, fundamental resources: for example, the number of machines expected to be available for use at a particular point in the future. In linear program terminology, the resource supply acts as an *upper bound* that limits how services can grow and where services can be placed. Ultimately, we want to make the best use of this resource supply as the intent-based description of the combined group of services allows.

Resource Pricing provides data about how much base-level, fundamental resources cost. For instance, the cost of machines may vary globally based upon the space/power charges of a given facility. In

linear program terminology, the prices inform the overall calculated costs, which act as the *objective* that we want to minimize.

Intent Config is the key to how intent-based information is fed to Auxon. It defines what constitutes a service, and how services relate to one another. The config ultimately acts as a configuration layer that allows all the other components to be wired together. It's designed to be human-readable and configurable.

Auxon Configuration Language Engine acts based upon the information it receives from the Intent Config. This component formulates a machine-readable request: a protocol buffer that can be understood by the Auxon Solver. It applies light sanity checking to the configuration, and is designed to act as the gateway between the human-configurable intent definition and the machine-parseable optimization request.

Auxon Solver is the brain of the tool. It formulates the giant mixed-integer or linear program based upon the optimization request received from the Configuration Language Engine. It is designed to be very scalable, which allows the solver to run in parallel upon hundreds or even thousands of machines running within Google's clusters. In addition to mixed-integer linear programming toolkits, there are also components within the Auxon Solver that handle tasks such as scheduling, managing a pool of workers, and descending decision trees.

Allocation Plan is the output of the Auxon Solver. It prescribes which resources should be allocated to which services in what locations. It is the computed implementation details of the intent-based definition of the capacity planning problem's requirements. The Allocation Plan also includes information about any requirements that could not be satisfied—for example, if a requirement couldn't be met due to a lack of resources, or competing requirements that were otherwise too strict.

Requirements and Implementation: Successes and Lessons Learned

Auxon was first imagined by an SRE and a technical program manager who had separately been tasked by their respective teams with capacity planning large portions of Google's infrastructure. Having performed manual capacity planning in spreadsheets, they were well positioned to understand the inefficiencies and opportunities for improvement through automation, and the features such a tool might require.

Throughout Auxon's development, the SRE team behind the product continued to be deeply involved in the production world. The team maintained a role in on-call rotations for several of Google's services, and participated in design discussions and technical leadership of these services. Through these

ongoing interactions, the team was able to stay grounded in the production world: they acted as both the consumer and developer of their own product. When the product failed, the team was directly impacted. Feature requests were informed through the team's own firsthand experiences. Not only did firsthand experience of the problem space buy a huge sense of ownership in the product's success, but it also helped give the product credibility and legitimacy within SRE.

Approximation

Don't focus on perfection and purity of solution, especially if the bounds of the problem aren't well known. Launch and iterate.

Any sufficiently complex software engineering effort is bound to encounter uncertainty as to how a component should be designed or how a problem should be tackled. Auxon met with such uncertainty early in its development because the linear programming world was uncharted territory for the team members. The limitations of linear programming, which seemed to be a central part of how the product would likely function, were not well understood. To address the team's consternation over this insufficiently understood dependency, we opted to initially build a simplified solver engine (the so-called "Stupid Solver") that applied some simple heuristics as to how services should be arranged based upon the user's specified requirements. While the Stupid Solver would never yield a truly optimal solution, it gave the team a sense that our vision for Auxon was achievable even if we didn't build something perfect from day one.

When deploying approximation to help speed development, it's important to undertake the work in a way that allows the team to make future enhancements and revisit approximation. In the case of the Stupid Solver, the entire solver interface was abstracted away within Auxon such that the solver internals could be swapped out at a later date. Eventually, as we built confidence in a unified linear programming model, it was a simple operation to switch out the Stupid Solver for something, well, smarter.

Auxon's product requirements also had some unknowns. Building software with fuzzy requirements can be a frustrating challenge, but some degree of uncertainty need not be a showstopper. Use this fuzziness as an incentive to ensure that the software is designed to be both general and modular. For instance, one of the aims of the Auxon project was to integrate with automation systems within Google to allow an Allocation Plan to be directly enacted on production (assigning resources and turning up/turning down/resizing services as appropriate). However, at the time, the world of automation systems was in a great deal of flux, as a huge variety of approaches were in use. Rather than try to design unique solutions to allow Auxon to work with each individual tool, we instead shaped the Allocation Plan to be universally useful such that these automation systems could work on their own integration points. This "agnostic" approach became key to Auxon's process for

onboarding new customers, because it allowed customers to begin using Auxon without switching to a particular turnup automation tool, forecasting tool, or performance data tool.

We also leveraged modular designs to deal with fuzzy requirements when building a model of machine performance within Auxon. Data on future machine platform performance (e.g., CPU) was scarce, but our users wanted a way to model various scenarios of machine power. We abstracted away the machine data behind a single interface, allowing the user to swap in different models of future machine performance. We later extended this modularity further, based on increasingly defined requirements, to provide a simple machine performance modeling library that worked within this interface.

If there's one theme to draw from our Auxon case study, it's that the old motto of "launch and iterate" is particularly relevant in SRE software development projects. Don't wait for the perfect design; rather, keep the overall vision in mind while moving ahead with design and development. When you encounter areas of uncertainty, design the software to be flexible enough so that if process or strategy changes at a higher level, you don't incur a huge rework cost. But at the same time, stay grounded by making sure that general solutions have a real-world–specific implementation that demonstrates the utility of the design.

Raising Awareness and Driving Adoption

As with any product, SRE-developed software must be designed with knowledge of its users and requirements. It needs to drive adoption through utility, performance, and demonstrated ability to both benefit Google's production reliability goals and to better the lives of SREs. The process of socializing a product and achieving buy-in across an organization is key to the project's success.

Don't underestimate the effort required to raise awareness and interest in your software product—a single presentation or email announcement isn't enough. Socializing internal software tools to a large audience demands all of the following:

- A consistent and coherent approach
- User advocacy
- The sponsorship of senior engineers and management, to whom you will have to demonstrate the utility of your product

It's important to consider the perspective of the customer in making your product usable. An engineer might not have the time or the inclination to dig into the source code to figure out how to use a tool. Although internal customers are generally more tolerant of rough edges and early alphas than external

customers, it's still necessary to provide documentation. SREs are busy, and if your solution is too difficult or confusing, they will write their own solution.

Set expectations

When an engineer with years of familiarity in a problem space begins designing a product, it's easy to imagine a utopian end-state for the work. However, it's important to differentiate aspirational goals of the product from minimum success criteria (or Minimum Viable Product). Projects can lose credibility and fail by promising too much, too soon; at the same time, if a product doesn't promise a sufficiently rewarding outcome, it can be difficult to overcome the necessary activation energy to convince internal teams to try something new. Demonstrating steady, incremental progress via small releases raises user confidence in your team's ability to deliver useful software.

In the case of Auxon, we struck a balance by planning a long-term roadmap alongside short-term fixes. Teams were promised that:

- Any onboarding and configuration efforts would provide the immediate benefit of alleviating the pain of manually bin packing short-term resource requests.
- As additional features were developed for Auxon, the same configuration files would carry over and provide new, and much broader, long-term cost savings and other benefits. The project road map enabled services to quickly determine if their use cases or required features weren't implemented in the early versions. Meanwhile, Auxon's iterative development approach fed into development priorities and new milestones for the road map.

Identify appropriate customers

The team developing Auxon realized that a one-size solution might not fit all; many larger teams already had home-grown solutions for capacity planning that worked passably well. While their custom tools weren't perfect, these teams didn't experience sufficient pain in the capacity planning process to try a new tool, especially an alpha release with rough edges.

The initial versions of Auxon intentionally targeted teams that had no existing capacity planning processes in place. Because these teams would have to invest configuration effort whether they adopted an existing tool or our new approach, they were interested in adopting the newest tool. The early successes Auxon achieved with these teams demonstrated the utility of the project, and turned the customers themselves into advocates for the tool. Quantifying the usefulness of the product

proved further beneficial; when we onboarded one of Google's Business Areas, the team authored a case study detailing the process and comparing the before and after results. The time savings and reduction of human toil alone presented a huge incentive for other teams to give Auxon a try.

Customer service

Even though software developed within SRE targets an audience of TPMs and engineers with high technical proficiency, any sufficiently innovative software still presents a learning curve to new users. Don't be afraid to provide white glove customer support for early adopters to help them through the onboarding process. Sometimes automation also entails a host of emotional concerns, such as fear that someone's job will be replaced by a shell script. By working one-on-one with early users, you can address those fears personally, and demonstrate that rather than owning the toil of performing a tedious task manually, the team instead owns the configurations, processes, and ultimate results of their technical work. Later adopters are convinced by the happy examples of early adopters.

Furthermore, because Google's SRE teams are distributed across the globe, early-adopter advocates for a project are particularly beneficial, because they can serve as local experts for other teams interested in trying out the project.

Designing at the right level

An idea that we've termed *agnosticism*—writing the software to be generalized to allow myriad data sources as input—was a key principle of Auxon's design. Agnosticism meant that customers weren't required to commit to any one tool in order to use the Auxon framework. This approach allowed Auxon to remain of sufficient general utility even as teams with divergent use cases began to use it. We approached potential users with the message, "come as you are; we'll work with what you've got." By avoiding over-customizing for one or two big users, we achieved broader adoption across the organization and lowered the barrier to entry for new services.

We've also consciously endeavored to avoid the pitfall of defining success as 100% adoption across the organization. In many cases, there are diminishing returns on closing the last mile to enable a feature set that is sufficient for every service in the long tail at Google.

Team Dynamics

In selecting engineers to work on an SRE software development product, we've found great benefit from creating a seed team that combines generalists who are able to get up to speed quickly on a new topic with engineers possessing a breadth of knowledge and experience. A diversity of

experiences covers blind spots as well as the pitfalls of assuming that every team's use case is the same as yours.

It's essential for your team to establish a working relationship with necessary specialists, and for your engineers to be comfortable working in a new problem space. For SRE teams at most companies, venturing into this new problem space requires outsourcing tasks or working with consultants, but SRE teams at larger organizations may be able to partner with in-house experts. During the initial phases of conceptualizing and designing Auxon, we presented our design document to Google's in-house teams that specialize in Operations Research and Quantitative Analysis in order to draw upon their expertise in the field and to bootstrap the Auxon team's knowledge about capacity planning.

As project development continued and Auxon's feature set grew more broad and complex, the team acquired members with backgrounds in statistics and mathematical optimization, which at a smaller company might be akin to bringing an outside consultant in-house. These new team members were able to identify areas for improvement when the project's basic functionality was complete and adding finesse had become our top priority.

The right time to engage specialists will, of course, vary from project to project. As a rough guideline, the project should be successfully off the ground and demonstrably successful, such that the skills of the current team would be significantly bolstered by the additional expertise.

Fostering Software Engineering in SRE

What makes a project a good candidate to take the leap from one-off tool to fully fledged software engineering effort? Strong positive signals include engineers with firsthand experience in the relative domain who are interested in working on the project, and a target user base that is highly technical (and therefore able to provide high-signal bug reports during the early phases of development). The project should provide noticeable benefits, such as reducing toil for SREs, improving an existing piece of infrastructure, or streamlining a complex process.

It's important for the project to fit into the overall set of objectives for the organization, so that engineering leaders can weigh its potential impact and subsequently advocate for your project, both with their reporting teams and with other teams that might interface with their teams. Cross-organizational socialization and review help prevent disjoint or overlapping efforts, and a product that can easily be established as furthering a department-wide objective is easier to staff and support.

What makes a poor candidate project? Many of the same red flags you might instinctively identify in any software project, such as software that touches many moving parts at once, or software design that requires an all-or-nothing approach that prevents iterative development. Because Google SRE

teams are currently organized around the services they run, SRE-developed projects are particularly at risk of being overly specific work that only benefits a small percentage of the organization. Because team incentives are aligned primarily to provide a great experience for the users of one particular service, projects often fail to generalize to a broader use case as standardization across SRE teams comes in second place. At the opposite end of the spectrum, overly generic frameworks can be equally problematic; if a tool strives to be too flexible and too universal, it runs the risk of not quite fitting any use case, and therefore having insufficient value in and of itself. Projects with grand scope and abstract goals often require significant development effort, but lack the concrete use cases required to deliver end-user benefit on a reasonable time frame.

As an example of a broad use case: a layer-3 load balancer developed by Google SREs proved so successful over the years that it was repurposed as a customer-facing product offering via Google Cloud Load Balancer [Eis16].

Successfully Building a Software Engineering Culture in SRE: Staffing and Development Time

SREs are often generalists, as the desire to learn breadth-first instead of depth-first lends itself well to understanding the bigger picture (and there are few pictures bigger than the intricate inner workings of modern technical infrastructure). These engineers often have strong coding and software development skills, but may not have the traditional SWE experience of being part of a product team or having to think about customer feature requests. A quote from an engineer on an early SRE software development project sums up the conventional SRE approach to software: "I have a design doc; why do we need requirements?" Partnering with engineers, TPMs, or PMs who are familiar with user-facing software development can help build a team software development culture that brings together the best of both software product development and hands-on production experience.

Dedicated, noninterrupted, project work time is essential to any software development effort. Dedicated project time is necessary to enable progress on a project, because it's nearly impossible to write code—much less to concentrate on larger, more impactful projects—when you're thrashing between several tasks in the course of an hour. Therefore, the ability to work on a software project without interrupts is often an attractive reason for engineers to begin working on a development project. Such time must be aggressively defended.

The majority of software products developed within SRE begin as side projects whose utility leads them to grow and become formalized. At this point, a product may branch off into one of several possible directions:

- Remain a grassroots effort developed in engineers' spare time
- Become established as a formal project through structured processes (see Getting There)
- Gain executive sponsorship from within SRE leadership to expand into a fully staffed software development effort

However, in any of these scenarios—and this is a point worth stressing—it's essential that the SREs involved in any development effort continue working as SREs instead of becoming full-time developers embedded in the SRE organization. Immersion in the world of production gives SREs performing development work an invaluable perspective, as they are both the creator and the customer for any product.

Getting There

If you like the idea of organized software development in SRE, you're probably wondering how to introduce a software development model to an SRE organization focused on production support.

First, recognize that this goal is as much an organizational change as it is a technical challenge. SREs are used to working closely with their teammates, quickly analyzing and reacting to problems. Therefore, you're working against the natural instinct of an SRE to quickly write some code to meet their immediate needs. If your SRE team is small, this approach may not be problematic. However, as your organization grows, this ad hoc approach won't scale, instead resulting in largely functional, yet narrow or single-purpose, software solutions that can't be shared, which inevitably lead to duplicated efforts and wasted time.

Next, think about what you want to achieve by developing software in SRE. Do you just want to foster better software development practices within your team, or are you interested in software development that produces results that can be used across teams, possibly as a standard for the organization? In larger established organizations, the latter change will take time, possibly spanning multiple years. Such a change needs to be tackled on multiple fronts, but has a higher payback. The following are some guidelines from Google's experience:

Create and communicate a clear message

> It's important to define and communicate your strategy, plans, and—most importantly—the benefits SRE gains from this effort. SREs are a skeptical lot (in fact, skepticism is a trait for which we specifically hire); an SRE's initial response to such an effort will likely be, "that sounds like too much overhead" or "it will never work." Start by making a compelling case of how this strategy will help SRE; for example:

- Consistent and supported software solutions speed ramp-up for new SREs.
- Reducing the number of ways to perform the same task allows the entire department to benefit from the skills any single team has developed, thus making knowledge and effort portable across teams.

When SREs start to ask questions about *how* your strategy will work, rather than *if* the strategy should be pursued, you know you've passed the first hurdle.

Evaluate your organization's capabilities

SREs have many skills, but it's relatively common for an SRE to lack experience as part of a team that built and shipped a product to a set of users. In order to develop useful software, you're effectively creating a product team. That team includes required roles and skills that your SRE organization may not have formerly demanded. Will someone play the role of product manager, acting as the customer advocate? Does your tech lead or project manager have the skills and/or experience to run an agile development process?

Begin filling these gaps by taking advantage of the skills already present in your company. Ask your product development team to help you establish agile practices via training or coaching. Solicit consulting time from a product manager to help you define product requirements and prioritize feature work. Given a large enough software-development opportunity, there may be a case to hire dedicated people for these roles. Making the case to hire for these roles is easier once you have some positive experiment results.

Launch and iterate

As you initiate an SRE software development program, your efforts will be followed by many watchful eyes. It's important to establish credibility by delivering some product of value in a reasonable amount of time. Your first round of products should aim for relatively straightforward and achievable targets—ones without controversy or existing solutions. We also found success in pairing this approach with a six-month rhythm of product update releases that provided additional useful features. This release cycle allowed teams to focus on identifying the right set of features to build, and then building those features while simultaneously learning how to be a productive software development team. After the initial launch, some Google teams moved to a push-on-green model for even faster delivery and feedback.

Don't lower your standards

As you start to develop software, you may be tempted to cut corners. Resist this urge by holding yourself to the same standards to which your product development teams are held. For example:

- Ask yourself: if this product were created by a separate dev team, would you onboard the product?

- If your solution enjoys broad adoption, it may become critical to SREs in order to successfully perform their jobs. Therefore, reliability is of utmost importance. Do you have proper code review practices in place? Do you have end-to-end or integration testing? Have another SRE team review the product for production readiness as they would if onboarding any other service.

It takes a long time to build credibility for your software development efforts, but only a short time to lose credibility due to a misstep.

Conclusions

Software engineering projects within Google SRE have flourished as the organization has grown, and in many cases the lessons learned from and successful execution of earlier software development projects have paved the way for subsequent endeavors. The unique hands-on production experience that SREs bring to developing tools can lead to innovative approaches to age-old problems, as seen with the development of Auxon to address the complex problem of capacity planning. SRE-driven software projects are also noticeably beneficial to the company in developing a sustainable model for supporting services at scale. Because SREs often develop software to streamline inefficient processes or automate common tasks, these projects mean that the SRE team doesn't have to scale linearly with the size of the services they support. Ultimately, the benefits of having SREs devoting some of their time to software development are reaped by the company, the SRE organization, and the SREs themselves.

PREVIOUS
Chapter 17 - Testing for Reliability

NEXT
Chapter 19 - Load Balancing at the Frontend