Chapter 27 - Reliable Product Launches at Scale

Reliable Product Launches at Scale

Written by Rhandeev Singh and Sebastian Kirsch with Vivek Rau
Edited by Betsy Beyer

Internet companies like Google are able to launch new products and features in far more rapid iterations than traditional companies. Site Reliability's role in this process is to enable a rapid pace of change without compromising stability of the site. We created a dedicated team of "Launch Coordination Engineers" to consult with engineering teams on the technical aspects of a successful launch.

The team also curated a "launch checklist" of common questions to ask about a launch, and recipes to solve common issues. The checklist proved to be a useful tool for ensuring reproducibly reliable launches.

Consider an ordinary Google service—for example, Keyhole, which serves satellite imagery for Google Maps and Google Earth. On a normal day, Keyhole serves up to several thousand satellite images per second. But on Christmas Eve in 2011, it received 25 times its normal peak traffic—upward of one million requests per second. What caused this massive surge in traffic?

Santa was coming.

A few years ago, Google collaborated with NORAD (the North American Aerospace Defense Command) to host a Christmas-themed website that tracked Santa's progress around the world, allowing users to watch him deliver presents in real time. Part of the experience was a "virtual fly-over," which used satellite imagery to track Santa's progress over a simulated world.

While a project like NORAD Tracks Santa may seem whimsical, it had all the characteristics that define a difficult and risky launch: a hard deadline (Google couldn't ask Santa to come a week later if the site wasn't ready), a lot of publicity, an audience of millions, and a very steep traffic ramp-up (everybody was going to be watching the site on Christmas Eve). Never underestimate the power of millions of kids anxious for presents—this project had a very real possibility of bringing Google's servers to their knees.

Google's Site Reliability Engineering team worked hard to prepare our infrastructure for this launch, making sure that Santa could deliver all his presents on time under the watchful eyes of an expectant audience. The last thing we wanted was to make children cry because they couldn't watch Santa deliver presents. In fact, we dubbed the various kill switches built into the experience to protect our services "Make-children-cry switches." Anticipating the many different ways this launch could go wrong and coordinating between the different engineering groups involved in the launch fell to a special team within Site Reliability Engineering: the Launch Coordination Engineers (LCE).

Launching a new product or feature is the moment of truth for every company—the point at which months or years of effort are presented to the world. Traditional companies launch new products at a fairly low rate. The launch cycle at Internet companies is markedly different. Launches and rapid iterations are far easier because new features can be rolled out on the server side, rather than requiring software rollout on individual customer workstations.

Google defines a launch as any new code that introduces an externally visible change to an application. Depending on a launch's characteristics—the combination of attributes, the timing, the number of steps involved, and the complexity—the launch process can vary greatly. According to this definition, Google sometimes performs up to 70 launches per week.

This rapid rate of change provides both the rationale and the opportunity for creating a streamlined launch process. A company that only launches a product every three years doesn't need a detailed launch process. By the time a new launch occurs, most components of the previously developed launch process will be

outdated. Nor do traditional companies have the opportunity to design a detailed launch process, because they don't accumulate enough experience performing launches to generate a robust and mature process.

Launch Coordination Engineering

Good software engineers have a great deal of expertise in coding and design, and understand the technology of their own products very well. However, the same engineers may be unfamiliar with the challenges and pitfalls of launching a product to millions of users while simultaneously minimizing outages and maximizing performance.

Google approached the challenges inherent to launches by creating a dedicated consulting team within SRE tasked with the technical side of launching a new product or feature. Staffed by software engineers and systems engineers—some with experience in other SRE teams—this team specializes in guiding developers toward building reliable and fast products that meet Google's standards for robustness, scalability, and reliability. This consulting team, Launch Coordination Engineering (LCE), facilitates a smooth launch process in a few ways:

- Auditing products and services for compliance with Google's reliability standards and best practices, and providing specific actions to improve reliability
- Acting as a liaison between the multiple teams involved in a launch
- Driving the technical aspects of a launch by making sure that tasks maintain momentum
- Acting as gatekeepers and signing off on launches determined to be "safe"
- Educating developers on best practices and on how to integrate with Google's services, equipping them with internal documentation and training resources to speed up their learning

Members of the LCE team audit services at various times during the service lifecycle. Most audits are conducted before a new product or service launches. If a product development team performs a launch without SRE support, LCE provides the appropriate domain knowledge to ensure a smooth launch. But even products that already have strong SRE support often engage with the LCE team during critical launches. The challenges teams face when launching a new product are substantially different from the day-to-day operation of a reliable service (a task at which SRE teams already excel), and the LCE team can draw on the experience from hundreds of launches. The LCE team also facilitates service audits when new services first engage with SRE.

The Role of the Launch Coordination Engineer

Our Launch Coordination Engineering team is composed of Launch Coordination Engineers (LCEs), who are either hired directly into this role, or are SREs with hands-on experience running Google services. LCEs are held to the same technical requirements as any other SRE, and are also expected to have strong communication and leadership skills—an LCE brings disparate parties together to work toward a common goal, mediates occasional conflicts, and guides, coaches, and educates fellow engineers.

A team dedicated to coordinating launches offers the following advantages:

Breadth of experience

> As a true cross-product team, the members are active across almost all of Google's product areas. Extensive cross-product knowledge and relationships with many teams across the company make LCEs excellent vehicles for knowledge transfer.

Cross-functional perspective

> LCEs have a holistic view of the launch, which enables them to coordinate among disparate teams in SRE, development, and product management. This holistic approach is particularly important for

complicated launches that can span more than half a dozen teams in multiple time zones.

Objectivity

: As a nonpartisan advisor, an LCE plays a balancing and mediating role between stakeholders including SRE, product developers, product managers, and marketing.

Because Launch Coordination Engineer is an SRE role, LCEs are incentivized to prioritize reliability over other concerns. A company that does not share Google's reliability goals, but shares its rapid rate of change, may choose a different incentive structure.

Setting Up a Launch Process

Google has honed its launch process over a period of more than 10 years. Over time we have identified a number of criteria that characterize a good launch process:

Lightweight

: Easy on developers

Robust

: Catches obvious errors

Thorough

: Addresses important details consistently and reproducibly

Scalable

: Accommodates both a large number of simple launches and fewer complex launches

Adaptable

: Works well for common types of launches (for example, adding a new UI language to a product) and new types of launches (for example, the initial launch of the Chrome browser or Google Fiber)

As you can see, some of these requirements are in obvious conflict. For example, it's hard to design a process that is simultaneously lightweight and thorough. Balancing these criteria against each other requires continuous work. Google has successfully employed a few tactics to help us achieve these criteria:

Simplicity

: Get the basics right. Don't plan for every eventuality.

A high touch approach

: Experienced engineers customize the process to suit each launch.

Fast common paths

: Identify classes of launches that always follow a common pattern (such as launching a product in a new country), and provide a simplified launch process for this class.

Experience has demonstrated that engineers are likely to sidestep processes that they consider too burdensome or as adding insufficient value—especially when a team is already in crunch mode, and the

launch process is seen as just another item blocking their launch. For this reason, LCE must optimize the launch experience continuously to strike the right balance between cost and benefit.

The Launch Checklist

Checklists are used to reduce failure and ensure consistency and completeness across a variety of disciplines. Common examples include aviation preflight checklists and surgical checklists [Gaw09]. Similarly, LCE employs a launch checklist for launch qualification. The checklist (Launch Coordination Checklist) helps an LCE assess the launch and provides the launching team with action items and pointers to more information. Here are some examples of items a checklist might include:

- **Question**: Do you need a new domain name?
    - **Action item**: Coordinate with marketing on your desired domain name, and request registration of the domain. Here is a link to the marketing form.
- **Question**: Are you storing persistent data?
    - **Action item**: Make sure you implement backups. Here are instructions for implementing backups.
- **Question**: Could a user potentially abuse your service?
    - **Action item**: Implement rate limiting and quotas. Use the following shared service.

In practice, there is a near-infinite number of questions to ask about any system, and it is easy for the checklist to grow to an unmanageable size. Maintaining a manageable burden on developers requires careful curation of the checklist. In an effort to curb its growth, at one point, adding new questions to Google's launch checklist required approval from a vice president. LCE now uses the following guidelines:

- Every question's importance must be substantiated, ideally by a previous launch disaster.
- Every instruction must be concrete, practical, and reasonable for developers to accomplish.

The checklist needs continuous attention in order to remain relevant and up-to-date: recommendations change over time, internal systems are replaced by different systems, and areas of concern from previous launches become obsolete due to new policies and processes. LCEs curate the checklist continuously and make small updates when team members notice items that need to be modified. Once or twice a year a team member reviews the entire checklist to identify obsolete items, and then works with service owners and subject matter experts to modernize sections of the checklist.

Driving Convergence and Simplification

In a large organization, engineers may not be aware of available infrastructure for common tasks (such as rate limiting). Lacking proper guidance, they're likely to re-implement existing solutions. Converging on a set of common infrastructure libraries avoids this scenario, and provides obvious benefits to the company: it cuts down on duplicate effort, makes knowledge more easily transferable between services, and results in a higher level of engineering and service quality due to the concentrated attention given to infrastructure.

Almost all groups at Google participate in a common launch process, which makes the launch checklist a vehicle for driving convergence on common infrastructure. Rather than implementing a custom solution, LCE can recommend existing infrastructure as building blocks—infrastructure that is already hardened through years of experience and that can help mitigate capacity, performance, or scalability risks.

Examples include common infrastructure for rate limiting or user quotas, pushing new data to servers, or releasing new versions of a binary. This type of standardization helped to radically simplify the launch checklist: for example, long sections of the checklist dealing with requirements for rate limiting could be replaced with a single line that stated, "Implement rate limiting using system X."

Due to their breadth of experience across all of Google's products, LCEs are also in a unique position to identify opportunities for simplification. While working on a launch, they witness the stumbling blocks firsthand: which parts of a launch are causing the most struggle, which steps take a disproportionate amount of time, which problems get solved independently over and over again in similar ways, where common infrastructure is lacking, or where duplication exists in common infrastructure. LCEs have various ways to streamline the launch experience and act as advocates for the launching teams. For example, LCEs might work with the owners of a particularly arduous approval process to simplify their criteria and implement automatic approvals for common cases. LCEs can also escalate pain points to the owners of common infrastructure and create a dialogue with the customers. By leveraging experience gained over the course of multiple previous launches, LCEs can devote more attention to individual concerns and suggestions.

Launching the Unexpected

When a project enters into a new product space or vertical, an LCE may need to create an appropriate checklist from scratch. Doing so often involves synthesizing experience from relevant domain experts. When drafting a new checklist, it can be helpful to structure the checklist around broad themes such as reliability, failure modes, and processes.

For example, before launching Android, Google had rarely dealt with mass consumer devices with client-side logic that we didn't directly control. While we can more or less easily fix a bug in Gmail within hours or days by pushing new versions of JavaScript to browsers, such fixes aren't an option with mobile devices. Therefore, LCEs working on mobile launches engaged mobile domain experts to determine which sections of existing checklists did or did not apply, and where new checklist questions were needed. In such conversations, it's important to keep the *intent* of each question in mind in order to avoid mindlessly applying a concrete question or action item that's not relevant to the design of the unique product being launched. An LCE facing an unusual launch must return to abstract first principles of how to execute a safe launch, then respecialize to make the checklist concrete and useful to developers.

Developing a Launch Checklist

A checklist is instrumental to launching new services and products with reproducible reliability. Our launch checklist grew over time and was periodically curated by members of the Launch Coordination Engineering team. The details of a launch checklist will be different for every company, because the specifics must be tailored to a company's internal services and infrastructure. In the following sections, we extract a number of themes from Google's LCE checklists and provide examples of how such themes might be fleshed out.

Architecture and Dependencies

An architecture review allows you to determine if the service is using shared infrastructure correctly and identifies the owners of shared infrastructure as additional stakeholders in the launch. Google has a large number of internal services that are used as building blocks for new products. During later stages of capacity planning (see [Hix15a]), the list of dependencies identified in this section of the checklist can be used to make sure that every dependency is correctly provisioned.

Example checklist questions

- What is your request flow from user to frontend to backend?
- Are there different types of requests with different latency requirements?

Example action items

- Isolate user-facing requests from non user–facing requests.
- Validate request volume assumptions. One page view can turn into many requests.

Integration

Many companies' services run in an internal ecosystem that entails guidelines on how to set up machines, configure new services, set up monitoring, integrate with load balancing, set up DNS addresses, and so forth. These internal ecosystems usually grow over time, and often have their own idiosyncrasies and pitfalls to navigate. Thus, this section of the checklist will vary widely from company to company.

Example action items

- Set up a new DNS name for your service.
- Set up load balancers to talk to your service.
- Set up monitoring for your new service.

Capacity Planning

New features may exhibit a temporary increase in usage at launch that subsides within days. The type of workload or traffic mix from a launch spike could be substantially different from steady state, throwing off load test results. Public interest is notoriously hard to predict, and some Google products had to accommodate launch spikes up to 15 times higher than initially estimated. Launching initially in one region or country at a time helps develop the confidence to handle larger launches.

Capacity interacts with redundancy and availability. For instance, if you need three replicated deployments to serve 100% of your traffic at peak, you need to maintain four or five deployments, one or two of which are redundant, in order to shield users from maintenance and unexpected malfunctions. Datacenter and network resources often have a long lead time and need to be requested far enough in advance for your company to obtain them.

Example checklist questions

- Is this launch tied to a press release, advertisement, blog post, or other form of promotion?
- How much traffic and rate of growth do you expect during and after the launch?
- Have you obtained all the compute resources needed to support your traffic?

Failure Modes

A systematic look at the possible failure modes of a new service ensures high reliability from the start. In this portion of the checklist, examine each component and dependency and identify the impact of its failure. Can the service deal with individual machine failures? Datacenter outages? Network failures? How do we deal with bad input data? Are we prepared for the possibility of a denial-of-service (DoS) attack? Can the service continue serving in degraded mode if one of its dependencies fails? How do we deal with unavailability of a dependency upon startup of the service? During runtime?

Example checklist questions

- Do you have any single points of failure in your design?
- How do you mitigate unavailability of your dependencies?

Example action items

- Implement request deadlines to avoid running out of resources for long-running requests.
- Implement load shedding to reject new requests early in overload situations.

Client Behavior

On a traditional website, there is rarely a need to take abusive behavior from legitimate users into account. When every request is triggered by a user action such as a click on a link, the request rates are limited by how quickly users can click. To double the load, the number of users would have to double.

This axiom no longer holds when we consider clients that initiate actions without user input—for example, a cell phone app that periodically syncs its data into the cloud, or a website that periodically refreshes. In either of these scenarios, abusive client behavior can very easily threaten the stability of a service. (There is also the topic of protecting a service from abusive traffic such as scrapers and denial-of-service attacks—which is different from designing safe behavior for first-party clients.)

Example checklist question

- Do you have auto-save/auto-complete/heartbeat functionality?

Example action items

- Make sure that your client backs off exponentially on failure.
- Make sure that you jitter automatic requests.

Processes and Automation

Google encourages engineers to use standard tools to automate common processes. However, automation is never perfect, and every service has processes that need to be executed by a human: creating a new release, moving the service to a different data center, restoring data from backups, and so on. For reliability reasons, we strive to minimize single points of failure, which include humans.

These remaining processes should be documented before launch to ensure that the information is translated from an engineer's mind onto paper while it is still fresh, and that it is available in an emergency. Processes should be documented in such a way that any team member can execute a given process in an emergency.

Example checklist question

- Are there any manual processes required to keep the service running?

Example action items

- Document all manual processes.
- Document the process for moving your service to a new datacenter.
- Automate the process for building and releasing a new version.

Development Process

Google is an extensive user of version control, and almost all development processes are deeply integrated with the version control system. Many of our best practices revolve around how to use the version control system effectively. For example, we perform most development on the mainline branch, but releases are built on separate branches per release. This setup makes it easy to fix bugs in a release without pulling in unrelated changes from the mainline.

Google also uses version control for other purposes, such as storing configuration files. Many of the advantages of version control—history tracking, attributing changes to individuals, and code reviews—apply to configuration files as well. In some cases, we also propagate changes from the version control system to the live servers automatically, so that an engineer only needs to submit a change to make it go live.

Example action items

- Check all code and configuration files into the version control system.
- Cut each release on a new release branch.

External Dependencies

Sometimes a launch depends on factors beyond company control. Identifying these factors allows you to mitigate the unpredictability they entail. For instance, the dependency may be a code library maintained by third parties, or a service or data provided by another company. When a vendor outage, bug, systematic error, security issue, or unexpected scalability limit actually occurs, prior planning will enable you to avert or mitigate damage to your users. In Google's history of launches, we've used filtering and/or rewriting proxies, data transcoding pipelines, and caches to mitigate some of these risks.

Example checklist questions

- What third-party code, data, services, or events does the service or the launch depend upon?
- Do any partners depend on your service? If so, do they need to be notified of your launch?
- What happens if you or the vendor can't meet a hard launch deadline?

Rollout Planning

In large distributed systems, few events happen instantaneously. For reasons of reliability, such immediacy isn't usually ideal anyway. A complicated launch might require enabling individual features on a number of different subsystems, and each of those configuration changes might take hours to complete. Having a working configuration in a test instance doesn't guarantee that the same configuration can be rolled out to the live instance. Sometimes a complicated dance or special functionality is required to make all components launch cleanly and in the correct order.

External requirements from teams like marketing and PR might add further complications. For example, a team might need a feature to be available in time for the keynote at a conference, but need to keep the feature invisible before the keynote.

Contingency measures are another part of rollout planning. What if you don't manage to enable the feature in time for the keynote? Sometimes these contingency measures are as simple as preparing a backup slide deck that says, "We will be launching this feature over the next days" rather than "We have launched this feature."

Example action items

- Set up a launch plan that identifies actions to take to launch the service. Identify who is responsible

for each item.
- Identify risk in the individual launch steps and implement contingency measures.

Selected Techniques for Reliable Launches

As described in other parts of this book, Google has developed a number of techniques for running reliable systems over the years. Some of these techniques are particularly well suited to launching products safely. They also provide advantages during regular operation of the service, but it's particularly important to get them right during the launch phase.

Gradual and Staged Rollouts

One adage of system administration is "never change a running system." Any change represents risk, and risk should be minimized in order to assure reliability of a system. What's true for any small system is doubly true for highly replicated, globally distributed systems like those run by Google.

Very few launches at Google are of the "push-button" variety, in which we launch a new product at a specific time for the entire world to use. Over time, Google has developed a number of patterns that allow us to launch products and features gradually and thereby minimize risk; see A Collection of Best Practices for Production Services.

Almost all updates to Google's services proceed gradually, according to a defined process, with appropriate verification steps interspersed. A new server might be installed on a few machines in one datacenter and observed for a defined period of time. If all looks well, the server is installed on all machines in one datacenter, observed again, and then installed on all machines globally. The first stages of a rollout are usually called "canaries"—an allusion to canaries carried by miners into a coal mine to detect dangerous gases. Our canary servers detect dangerous effects from the behavior of the new software under real user traffic.

Canary testing is a concept embedded into many of Google's internal tools used to make automated changes, as well as for systems that change configuration files. Tools that manage the installation of new software typically observe the newly started server for a while, making sure that the server doesn't crash or otherwise misbehave. If the change doesn't pass the validation period, it's automatically rolled back.

The concept of gradual rollouts even applies to software that does not run on Google's servers. New versions of an Android app can be rolled out in a gradual manner, in which the updated version is offered to a subset of the installs for upgrade. The percentage of upgraded instances gradually increases over time until it reaches 100%. This type of rollout is particularly helpful if the new version results in additional traffic to the backend servers in Google's datacenters. This way, we can observe the effect on our servers as we gradually roll out the new version and detect problems early.

The invite system is another type of gradual rollout. Frequently, rather than allowing free signups to a new service, only a limited number of users are allowed to sign up per day. Rate-limited signups are often coupled with an invite system, in which a user can send a limited number of invites to friends.

Feature Flag Frameworks

Google often augments prelaunch testing with strategies that mitigate the risk of an outage. A mechanism to roll out changes slowly, allowing for observation of total system behavior under real workloads, can pay for its engineering investment in reliability, engineering velocity, and time to market. These mechanisms have proven particularly useful in cases where realistic test environments are impractical, or for particularly complex launches for which the effects can be hard to predict.

Furthermore, not all changes are equal. Sometimes you simply want to check whether a small tweak to the user interface improves the experience of your users. Such small changes shouldn't involve thousands of lines of code or a heavyweight launch process. You may want to test hundreds of such changes in parallel.

Finally, sometimes you want to find out whether a small sample of users like using an early prototype of a new, hard-to-implement feature. You don't want to spend months of engineering effort to harden a new feature to serve millions of users, only to find that the feature is a flop.

To accommodate the preceding scenarios, several Google products devised feature flag frameworks. Some of those frameworks were designed to roll out new features gradually from 0% to 100% of users. Whenever a product introduced any such framework, the framework itself was hardened as much as possible so that most of its applications would not need any LCE involvement. Such frameworks usually meet the following requirements:

- Roll out many changes in parallel, each to a few servers, users, entities, or datacenters
- Gradually increase to a larger but limited group of users, usually between 1 and 10 percent
- Direct traffic through different servers depending on users, sessions, objects, and/or locations
- Automatically handle failure of the new code paths by design, without affecting users
- Independently revert each such change immediately in the event of serious bugs or side effects
- Measure the extent to which each change improves the user experience

Google's feature flag frameworks fall into two general classes:

- Those that primarily facilitate user interface improvements
- Those that support arbitrary server-side and business logic changes

The simplest feature flag framework for user interface changes in a stateless service is an HTTP payload rewriter at frontend application servers, limited to a subset of cookies or another similar HTTP request/response attribute. A configuration mechanism may specify an identifier associated with the new code paths and the scope of the change (e.g., cookie hash mod range), whitelists, and blacklists.

Stateful services tend to limit feature flags to a subset of unique logged-in user identifiers or to the actual product entities accessed, such as the ID of documents, spreadsheets, or storage objects. Rather than rewrite HTTP payloads, stateful services are more likely to proxy or reroute requests to different servers depending on the change, conferring the ability to test improved business logic and more complex new features.

Dealing with Abusive Client Behavior

The simplest example of abusive client behavior is a misjudgment of update rates. A new client that syncs every 60 seconds, as opposed to every 600 seconds, causes 10 times the load on the service. Retry behavior has a number of pitfalls that affect user-initiated requests, as well as client-initiated requests. Take the example of a service that is overloaded and is therefore failing some requests: if the clients retry the failed requests, they add load to an already overloaded service, resulting in more retries and even more requests. Instead, clients need to reduce the frequency of retries, usually by adding exponentially increasing delay between retries, in addition to carefully considering the types of errors that warrant a retry. For example, a network error usually warrants a retry, but a 4xx HTTP error (which indicates an error on the client's side) usually does not.

Intentional or inadvertent synchronization of automated requests in a thundering herd (much like those described in Chapters Distributed Periodic Scheduling with Cron and Data Processing Pipelines) is another common example of abusive client behavior. A phone app developer might decide that 2 a.m. is a good time to download updates, because the user is most likely asleep and won't be inconvenienced by the download. However, such a design results in a barrage of requests to the download server at 2 a.m. every

night, and almost no requests at any other time. Instead, every client should choose the time for this type of request randomly.

Randomness also needs to be injected into other periodic processes. To return to the previously mentioned retries: let's take the example of a client that sends a request, and when it encounters a failure, retries after 1 second, then 2 seconds, then 4 seconds, and so on. Without randomness, a brief request spike that leads to an increased error rate could repeat itself due to retries after 1 second, then 2 seconds, then 4 seconds. In order to even out these synchronized events, each delay needs to be jittered (that is, adjusted by a random amount).

The ability to control the behavior of a client from the server side has proven an important tool in the past. For an app on a device, such control might mean instructing the client to check in periodically with the server and download a configuration file. The file might enable or disable certain features or set parameters, such as how often the client syncs or how often it retries.

The client configuration might even enable completely new user-facing functionality. By hosting the code that supports new functionality in the client application before we activate that feature, we greatly reduce the risk associated with a launch. Releasing a new version becomes much easier if we don't need to maintain parallel release tracks for a version with the new functionality versus without the functionality. This holds particularly true if we're not dealing with a single piece of new functionality, but a set of independent features that might be released on different schedules, which would necessitate maintaining a combinatorial explosion of different versions.

Having this sort of dormant functionality also makes aborting launches easier when adverse effects are discovered during a rollout. In such cases, we can simply switch the feature off, iterate, and release an updated version of the app. Without this type of client configuration, we would have to provide a new version of the app without the feature, and update the app on all users' phones.

Overload Behavior and Load Tests

Overload situations are a particularly complex failure mode, and therefore deserve additional attention. Runaway success is usually the most welcome cause of overload when a new service launches, but there are myriad other causes, including load balancing failures, machine outages, synchronized client behavior, and external attacks.

A naive model assumes that CPU usage on a machine providing a particular service scales linearly with the load (for example, number of requests or amount of data processed), and once available CPU is exhausted, processing simply becomes slower. Unfortunately, services rarely behave in this ideal fashion in the real world. Many services are much slower when they are not loaded, usually due to the effect of various kinds of caches such as CPU caches, JIT caches, and service-specific data caches. As load increases, there is usually a window in which CPU usage and load on the service correspond linearly, and response times stay mostly constant.

At some point, many services reach a point of nonlinearity as they approach overload. In the most benign cases, response times simply begin to increase, resulting in a degraded user experience but not necessarily causing an outage (although a slow dependency might cause user-visible errors up the stack, due to exceeded RPC deadlines). In the most drastic cases, a service locks up completely in response to overload.

To cite a specific example of overload behavior: a service logged debugging information in response to backend errors. It turned out that logging debugging information was more expensive than handling the backend response in a normal case. Therefore, as the service became overloaded and timed out backend responses inside its own RPC stack, the service spent even more CPU time logging these responses, timing out more requests in the meantime until the service ground to a complete halt. In services running on the Java Virtual Machine (JVM), a similar effect of grinding to a halt is sometimes called "GC (garbage

collection) thrashing." In this scenario, the virtual machine's internal memory management runs in increasingly closer cycles, trying to free up memory until most of the CPU time is consumed by memory management.

Unfortunately, it is very hard to predict from first principles how a service will react to overload. Therefore, load tests are an invaluable tool, both for reliability reasons and capacity planning, and load testing is required for most launches.

Development of LCE

In Google's formative years, the size of the engineering team doubled every year for several years in a row, fragmenting the engineering department into many small teams working on many experimental new products and features. In such a climate, novice engineers run the risk of repeating the mistakes of their predecessors, especially when it comes to launching new features and products successfully.

To mitigate the repetition of such mistakes by capturing the lessons learned from past launches, a small band of experienced engineers, called the "Launch Engineers," volunteered to act as a consulting team. The Launch Engineers developed checklists for new product launches, covering topics such as:

- When to consult with the legal department
- How to select domain names
- How to register new domains without misconfiguring DNS
- Common engineering design and production deployment pitfalls

"Launch Reviews," as the Launch Engineers' consulting sessions came to be called, became a common practice days to weeks before the launch of many new products.

Within two years, the product deployment requirements in the launch checklist grew long and complex. Combined with the increasing complexity of Google's deployment environment, it became more and more challenging for product engineers to stay up-to-date on how to make changes safely. At the same time, the SRE organization was growing quickly, and inexperienced SREs were sometimes overly cautious and averse to change. Google ran a risk that the resulting negotiations between these two parties would reduce the velocity of product/feature launches.

To mitigate this scenario from the engineering perspective, SRE staffed a small, full-time team of LCEs in 2004. They were responsible for accelerating the launches of new products and features, while at the same time applying SRE expertise to ensure that Google shipped reliable products with high availability and low latency.

LCEs were responsible for making sure launches were executing quickly without the services falling over, and that if a launch did fail, it didn't take down other products. LCEs were also responsible for keeping stakeholders informed of the nature and likelihood of such failures whenever corners were cut in order to accelerate time to market. Their consulting sessions were formalized as Production Reviews.

Evolution of the LCE Checklist

As Google's environment grew more complex, so did both the Launch Coordination Engineering checklist (see Launch Coordination Checklist) and the volume of launches. In 3.5 years, one LCE ran 350 launches through the LCE Checklist. As the team averaged five engineers during this time period, this translates into a Google launch throughput of over 1,500 launches in 3.5 years!

While each question on the LCE Checklist is simple, much complexity is built in to what prompted the question and the implications of its answer. In order to fully understand this degree of complexity, a new LCE hire requires about six months of training.

As the volume of launches grew, keeping pace with the annual doubling of Google's engineering team, LCEs sought ways to streamline their reviews. LCEs identified categories of low-risk launches that were highly unlikely to face or cause mishaps. For example, a feature launch involving no new server executables and a traffic increase under 10% would be deemed low risk. Such launches were faced with an almost trivial checklist, while higher-risk launches underwent the full gamut of checks and balances. By 2008, 30% of reviews were considered low-risk.

Simultaneously, Google's environment was scaling up, removing constraints on many launches. For instance, the acquisition of YouTube forced Google to build out its network and utilize bandwidth more efficiently. This meant that many smaller products would "fit within the cracks," avoiding complex network capacity planning and provisioning processes, thus accelerating their launches. Google also began building very large datacenters capable of hosting several dependent services under one roof. This development simplified the launch of new products that needed large amounts of capacity at multiple preexisting services upon which they depended.

Problems LCE Didn't Solve

Although LCEs tried to keep the bureaucracy of reviews to a minimum, such efforts were insufficient. By 2009, the difficulties of launching a small new service at Google had become a legend. Services that grew to a larger scale faced their own set of problems that Launch Coordination could not solve.

Scalability changes

When products are successful far beyond any early estimates, and their usage increases by more than two orders of magnitude, keeping pace with their load necessitates many design changes. Such scalability changes, combined with ongoing feature additions, often make the product more complex, fragile, and difficult to operate. At some point, the original product architecture becomes unmanageable and the product needs to be completely rearchitected. Rearchitecting the product and then migrating all users from the old to the new architecture requires a large investment of time and resources from developers and SREs alike, slowing down the rate of new feature development during that period.

Growing operational load

When running a service after it launches, operational load, the amount of manual and repetitive engineering needed to keep a system functioning, tends to grow over time unless efforts are made to control such load. Noisiness of automated notifications, complexity of deployment procedures, and the overhead of manual maintenance work tend to increase over time and consume increasing amounts of the service owner's bandwidth, leaving the team less time for feature development. SRE has an internally advertised goal of keeping operational work below a maximum of 50%; see Eliminating Toil. Staying below this maximum requires constant tracking of sources of operational work, as well as directed effort to remove these sources.

Infrastructure churn

If the underlying infrastructure (such as systems for cluster management, storage, monitoring, load balancing, and data transfer) is changing due to active development by infrastructure teams, the owners of services running on the infrastructure must invest large amounts of work to simply keep up with the infrastructure changes. As infrastructure features upon which services rely are deprecated and replaced by new features, service owners must continually modify their configurations and rebuild their executables, consequently "running fast just to stay in the same place." The solution to this scenario is to enact some type of churn reduction policy that prohibits infrastructure engineers from releasing backward-incompatible features until they also automate the migration of their clients to the new feature. Creating automated migration tools to accompany new features minimizes the work imposed on service owners to

keep up with infrastructure churn.

Solving these problems requires company-wide efforts that are far beyond the scope of LCE: a combination of better platform APIs and frameworks (see The Evolving SRE Engagement Model), continuous build and test automation, and better standardization and automation across Google's production services.

Conclusion

Companies undergoing rapid growth with a high rate of change to products and services may benefit from the equivalent of a Launch Coordination Engineering role. Such a team is especially valuable if a company plans to double its product developers every one or two years, if it must scale its services to hundreds of millions of users, and if reliability despite a high rate of change is important to its users.

The LCE team was Google's solution to the problem of achieving safety without impeding change. This chapter introduced some of the experiences accumulated by our unique LCE role over a 10-year period under exactly such circumstances. We hope that our approach will help inspire others facing similar challenges in their respective organizations.

previous

Chapter 26 - Data Integrity: What You Read Is What You Wrote

next

Part IV - Management