Chapter 24. Continuous Delivery

Written by Radha Narayan, Bobbi Jones, Sheri Shipe, and David Owens

Edited by Lisa Carey

Given how quickly and unpredictably the technology landscape shifts, the competitive advantage for any product lies in its ability to quickly go to market. An organization's velocity is a critical factor in its ability to compete with other players, maintain product and service quality, or adapt to new regulation. This velocity is bottlenecked by the time to deployment. Deployment doesn't just happen once at initial launch. There is a saying among educators that no lesson plan survives its first contact with the student body. In much the same way, no software is perfect at first launch, and the only guarantee is that you'll have to update it. Quickly.

The long-term life cycle of a software product involves rapid exploration of new ideas, rapid responses to landscape shifts or user issues, and enabling developer velocity at scale. From Eric Raymond's *The Cathedral and the Bazaar* to Eric Reis' *The Lean Startup*, the key to any organization's long-term success has always been in its ability to get ideas executed and into users' hands as quickly as possible and reacting quickly to their feedback. Martin Fowler, in his book *Continuous Delivery* (aka CD), points out that "The biggest risk to any software effort is that you end up building something that isn't useful. The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is."

Work that stays in progress for a long time before delivering user value is high risk and high cost, and can even be a drain on morale. At Google, we strive to release early and often, or "launch and iterate," to enable teams to see the impact of their work quickly and to adapt faster to a shifting market. The value of code is not realized at the time of submission but when features are available to your users. Reducing the time between "code complete" and user feedback minimizes the cost of work that is in progress.

You get extraordinary outcomes by realizing that the launch never lands but that it begins a learning cycle where you then fix the next most important thing, measure how it went, fix the next thing, etc.—and it is never complete.

—David Weekly, Former Google product manager

At Google, the practices we describe in this book allow hundreds (or in some cases thousands) of engineers to quickly troubleshoot problems, to independently work on new features without worrying about the release, and to understand the effectiveness of new features through A/B experimentation. This chapter focuses on the key levers of rapid innovation, including managing risk, enabling developer velocity at scale, and understanding the cost and value trade-off of each feature you launch.

Idioms of Continuous Delivery at Google

A core tenet of Continuous Delivery (CD) as well as of Agile methodology is that over time, smaller batches of changes result in higher quality; in other words, *faster is safer*. This can seem deeply controversial to teams at first glance, especially if the prerequisites for setting up CD—for example, Continuous Integration (CI) and testing—are not yet in place. Because it might take a while for all teams to realize the ideal of CD, we focus on developing various aspects that deliver value independently en route to the end goal. Here are some of these:

Agility

Release frequently and in small batches

Automation

Reduce or remove repetitive overhead of frequent releases

Isolation

Strive for modular architecture to isolate changes and make troubleshooting easier

Reliability

Measure key health indicators like crashes or latency and keep improving them

Data-driven decision making

Use A/B testing on health metrics to ensure quality

Phased rollout

Roll out changes to a few users before shipping to everyone

At first, releasing new versions of software frequently might seem risky. As your userbase grows, you might fear the backlash from angry users if there are any bugs that you didn't catch in testing, and you might quite simply have too much new code in your product to test exhaustively. But this is precisely where CD can help. Ideally, there are so few changes between one release and the next that troubleshooting issues is trivial. In the limit, with CD, every change goes through the QA pipeline and is automatically deployed into production. This is often not a practical reality for many teams, and so there is often work of culture change toward CD as an intermediate step, during which teams can build their readiness to deploy at any time without actually doing so, building up their confidence to release more frequently in the future.

Velocity Is a Team Sport: How to Break Up a Deployment into Manageable Pieces

When a team is small, changes come into a codebase at a certain rate. We've seen an antipattern emerge as a team grows over time or splits into subteams: a subteam branches off its code to avoid stepping on anyone's feet, but then struggles, later, with integration and culprit finding. At Google, we prefer that teams continue to develop at head in the shared codebase and set up CI testing, automatic rollbacks, and culprit finding to identify issues quickly. This is discussed at length in Chapter 23.

One of our codebases, YouTube, is a large, monolithic Python application. The release process is laborious, with Build Cops, release managers, and other volunteers. Almost every release has multiple cherry-picked changes and respins. There is also a 50-hour manual regression testing cycle run by a remote QA team on every release. When the operational cost of a release is this high, a cycle begins to develop in which you wait to push out your release until you're able to test it a bit more. Meanwhile, someone wants to add just one more feature that's almost ready, and pretty soon you have yourself a release process that's laborious, error prone, and slow. Worst of all, the experts who did the release last time are burned out and have left the team, and now nobody even knows how to troubleshoot those strange crashes that happen when you try to release an update, leaving you panicky at the very thought of pushing that button.

If your releases are costly and sometimes risky, the *instinct* is to slow down your release cadence and increase your stability period. However, this only provides short-term stability gains, and over time it slows velocity and frustrates teams and users. The *answer* is to reduce cost, increase discipline, and make the risks more incremental, but it is critical to resist the obvious operational fixes and invest in long-term architectural changes. The obvious operational fixes to this problem lead to a few traditional approaches: reverting to a traditional planning model that leaves little room for learning or iteration, adding more governance and oversight to the development process, and implementing risk reviews or rewarding low-risk (and often low-value) features.

The investment with the best return, though, is migrating to a microservice architecture, which can empower a large product team with the ability to remain scrappy and innovative while simultaneously reducing risk. In some cases, at Google, the answer has been to rewrite an application from scratch rather than simply migrating it, establishing the desired modularity into the new architecture. Although either of these options can take months and is likely painful in the short term, the value gained in terms of operational cost and cognitive simplicity will pay off over an application's lifespan of years.

Evaluating Changes in Isolation: Flag-Guarding Features

A key to reliable continuous releases is to make sure engineers "flag guard" *all changes*. As a product grows, there will be multiple features under various stages of development coexisting in a binary. Flag guarding can be used to control the inclusion or expression of feature code in the product on a feature-by-feature basis and can be expressed differently for release and development builds. A feature flag disabled for a build should allow build tools to strip the feature from the build if the language permits it. For instance, a stable feature that has already shipped to customers might be enabled for both development and release builds. A feature under development might be enabled only for development, protecting users from an unfinished feature. New feature code lives in the binary alongside the old codepath—both can run, but the new code is guarded by a flag. If the new code works, you can remove the old codepath and launch the feature fully in a subsequent release. If there's a problem, the flag value can be updated independently from the binary release via a dynamic config update.

In the old world of binary releases, we had to time press releases closely with our binary rollouts. We had to have a successful rollout before a press release about new functionality or a new feature could be issued. This meant that the feature would be out in the wild before it was announced, and the risk of it being discovered ahead of time was very real.

This is where the beauty of the flag guard comes to play. If the new code has a flag, the flag can be updated to turn your feature on immediately before the press release, thus minimizing the risk of leaking a feature. Note that flag-guarded code is not a *perfect* safety net for truly sensitive features. Code can still be scraped and analyzed if it's not well obfuscated, and not all features can be hidden behind flags without adding a lot of complexity. Moreover, even flag configuration changes must be rolled out with care. Turning on a flag for 100% of your users all at once is not a great idea, so a configuration service that manages safe configuration rollouts is a good investment. Nevertheless, the level of control and the ability to decouple the destiny of a particular feature from the overall product release are powerful levers for long-term sustainability of the application.

Striving for Agility: Setting Up a Release Train

Google's Search binary is its first and oldest. Large and complicated, its codebase can be tied back to Google's origin—a search through our codebase can still find code written at least as far back as 2003, often earlier. When smartphones began to take off, feature after mobile feature was shoehorned into a hairball of code written primarily for server deployment. Even though the Search experience was becoming more vibrant and interactive, deploying a viable build became more and more difficult. At one point, we were releasing the Search binary into production only once per week, and even hitting that target was rare and often based on luck.

When one of our contributing authors, Sheri Shipe, took on the project of increasing our release velocity in Search, each release cycle was taking a group of engineers days to complete. They built the binary, integrated data, and then began testing. Each bug had to be manually triaged to make sure it wouldn't impact Search quality, the user experience (UX), and/or revenue. This process was grueling and time consuming and did not scale to the volume or rate of change. As a result, a developer could never know when their feature was going to be released into production. This made timing press releases and public launches challenging.

Releases don't happen in a vacuum, and having reliable releases makes the dependent factors easier to synchronize. Over the course of several years, a dedicated group of engineers implemented a continuous release process, which streamlined everything about sending a Search binary into the world. We automated what we could, set deadlines for submitting features, and simplified the integration of plug-ins and data into the binary. We could now consistently release a new Search binary into production every other day.

What were the trade-offs we made to get predictability in our release cycle? They narrow down to two main ideas we baked into the system.

No Binary Is Perfect

The first is that *no binary is perfect*, especially for builds that are incorporating the work of tens or hundreds of developers independently developing dozens of major features. Even though it's impossible to fix every bug, we constantly need to weigh questions such as: If a line has been moved two pixels to the left, will it affect an ad display and potential revenue? What if the shade of a box has been altered slightly? Will it make it difficult for visually impaired users to read the text? The rest of this book is arguably about minimizing the set of unintended outcomes for a release, but in the end we must admit that software is fundamentally complex. There is no perfect binary—decisions and trade-offs have to be made every time a new change is released into production. Key performance indicator metrics with clear thresholds allow features to launch even if they aren't perfect[1] and can also create clarity in otherwise contentious launch decisions.

One bug involved a rare dialect spoken on only one island in the Philippines. If a user asked a search question in this dialect, instead of an answer to their question, they would get a blank web page. We had to determine whether the cost of fixing this bug was worth delaying the release of a major new feature.

We ran from office to office trying to determine how many people actually spoke this language, if it happened every time a user searched in this language, and whether these folks even used Google on a regular basis. Every quality engineer we spoke with deferred us to a more senior person. Finally, data in hand, we put the question to Search's senior vice president. Should we delay a critical release to fix a bug that affected only a very small Philippine island? It turns out that no matter how small your island, you should get reliable and accurate search results: we delayed the release and fixed the bug.

Meet Your Release Deadline

The second idea is that *if you're late for the release train, it will leave without you*. There's something to be said for the adage, "deadlines are certain, life is not." At some point in the release timeline, you must put a stake in the ground and turn away developers and their new features. Generally speaking, no amount of pleading or begging will get a feature into today's release after the deadline has passed.

There is the *rare* exception. The situation usually goes like this. It's late Friday evening and six software engineers come storming into the release manager's cube in a panic. They have a contract with the NBA and finished the feature moments ago. But it must go live before the big game tomorrow. The release must stop and we must cherry-pick the feature into the binary or we'll be in breach of contract! A bleary-eyed release engineer shakes their head and says it will take four hours to cut and test a new binary. It's their kid's birthday and they still need to pick up the balloons.

A world of regular releases means that if a developer misses the release train, they'll be able to catch the next train in a matter of hours rather than days. This limits developer panic and greatly improves work–life balance for release engineers.

Quality and User-Focus: Ship Only What

Gets Used

Bloat is an unfortunate side effect of most software development life cycles, and the more successful a product becomes, the more bloated its code base typically becomes. One downside of a speedy, efficient release train is that this bloat is often magnified and can manifest in challenges to the product team and even to the users. Especially if the software is delivered to the client, as in the case of mobile apps, this can mean the user's device pays the cost in terms of space, download, and data costs, even for features they never use, whereas developers pay the cost of slower builds, complex deployments, and rare bugs. In this section, we'll talk about how dynamic deployments allow you to ship only what is used, forcing necessary trade-offs between user value and feature cost. At Google, this often means staffing dedicated teams to improve the efficiency of the product on an ongoing basis.

Whereas some products are web-based and run on the cloud, many are client applications that use shared resources on a user's device—a phone or tablet. This choice in itself showcases a trade-off between native apps that can be more performant and resilient to spotty connectivity, but also more difficult to update and more susceptible to platform-level issues. A common argument against frequent, continuous deployment for native apps is that users dislike frequent updates and must pay for the data cost and the disruption. There might be other limiting factors such as access to a network or a limit to the reboots required to percolate an update.

Even though there is a trade-off to be made in terms of how frequently to update a product, the goal is to *have these choices be intentional*. With a smooth, well-running CD process, how often a viable release is *created* can be separated from how often a user *receives* it. You might achieve the goal of being able to deploy weekly, daily, or hourly, without actually doing so, and you should intentionally choose release processes in the context of your users' specific needs and the larger organizational goals, and determine the staffing and tooling model that will best support the long-term sustainability of your product.

Earlier in the chapter, we talked about keeping your code modular. This allows for dynamic, configurable deployments that allow better utilization of constrained resources, such as the space on a user's device. In the absence of this practice, every user must receive code they will never use to support translations they don't need or architectures that were meant for other kinds of devices. Dynamic deployments allow apps to maintain small sizes while only shipping code to a device that brings its users value, and A/B experiments allow for intentional trade-offs between a feature's cost and its value to users and your business.

There is an upfront cost to setting up these processes, and identifying and removing frictions that keep the frequency of releases lower than is desirable is a painstaking process. But the long-term wins in terms of risk management, developer velocity, and enabling rapid innovation are so high that these initial costs become worthwhile.

Shifting Left: Making Data-Driven Decisions Earlier

If you're building for all users, you might have clients on smart screens, speakers, or Android and iOS phones and tablets, and your software may be flexible enough to allow users to customize their experience. Even if you're building for only Android devices, the sheer diversity of the more than two billion Android devices can make the prospect of qualifying a release overwhelming. And with the pace of innovation, by the time someone reads this chapter, whole new categories of devices might have bloomed.

One of our release managers shared a piece of wisdom that turned the situation around when he said that the diversity of our client market was not a *problem*, but a *fact*. After we accepted that, we could switch our release qualification model in the following ways:

- If *comprehensive* testing is practically infeasible, aim for *representative* testing instead.
- Staged rollouts to slowly increasing percentages of the userbase allow for fast fixes.
- Automated A/B releases allow for statistically significant results proving a release's quality, without tired humans needing to look at dashboards and make decisions.

When it comes to developing for Android clients, Google apps use specialized testing tracks and staged rollouts to an increasing percentage of user traffic, carefully monitoring for issues in these channels. Because the Play Store offers unlimited testing tracks, we can also set up a QA team in each country in which we plan to launch, allowing for a global overnight turnaround in testing key features.

One issue we noticed when doing deployments to Android was that we could expect a statistically significant change in user metrics *simply from pushing an update*. This meant that even if we made no changes to our product, pushing an update could affect device and user behavior in ways that were difficult to predict. As a result, although canarying the update to a small percentage of user traffic could give us good information about crashes or stability problems, it told us very little about whether the newer version of our app was in fact better than the older one.

Dan Siroker and Pete Koomen have already discussed the value of A/B testing[2] your features, but at Google, some of our larger apps also A/B test their *deployments*. This means sending out two versions of the product: one that is the desired update, with the baseline being a placebo (your old version just gets shipped again). As the two versions roll out simultaneously to a large enough base of similar users, you can compare one release against the other to see whether the latest version of your software is in fact an improvement over the previous one. With a large enough userbase, you should be able to get statistically significant results within days, or even hours. An automated metrics pipeline can enable the fastest possible release by pushing forward a release to more traffic as soon as there is enough data to know that the guardrail metrics will not be affected.

Obviously, this method does not apply to every app and can be a lot of overhead when you don't have a large enough userbase. In these cases, the recommended best practice is to aim for change-neutral releases. All new features are flag guarded so that the only change being tested during a rollout is the stability of the deployment itself.

Changing Team Culture: Building Discipline into Deployment

Although "Always Be Deploying" helps address several issues affecting developer velocity, there are also certain practices that address issues of scale.

The initial team launching a product can be fewer than 10 people, each taking turns at deployment and production-monitoring responsibilities. Over time, your team might grow to hundreds of people, with subteams responsible for specific features. As this happens and the organization scales up, the number of changes in each deployment and the amount of risk in each release attempt is increasing superlinearly. Each release contains months of sweat and tears. Making the release successful becomes a high-touch and labor-intensive effort. Developers can often be caught trying to decide which is worse: abandoning a release that contains a quarter's worth of new features and bug fixes, or pushing out a release without confidence in its quality.

At scale, increased complexity usually manifests as increased release latency. Even if you release every day, a release can take a week or longer to fully roll out safely, leaving you a week behind when trying to debug any issues. This is where "Always Be Deploying" can return a development project to effective form. Frequent release trains allow for minimal divergence from a known good position, with the recency of changes aiding in resolving issues. But how can a team ensure that the complexity inherent with a large and quickly expanding codebase doesn't weigh down progress?

On Google Maps, we take the perspective that features are very important, but only very seldom is any feature so important that a release should be held for it. If releases are frequent, the pain a feature feels for missing a release is small in comparison to the pain all the new features in a release feel for a delay, and especially the pain users can feel if a not-quite-ready feature is rushed to be included.

One release responsibility is to protect the product from the developers.

When making trade-offs, the passion and urgency a developer feels about launching a new feature can never trump the user experience with an existing product. This means that new features must be isolated from other components via interfaces with strong contracts, separation of concerns, rigorous testing, communication early and often, and conventions for new feature acceptance.

Conclusion

Over the years and across all of our software products, we’ve found that, counterintuitively, faster is safer. The health of your product and the speed of development are not actually in opposition to each other, and products that release more frequently and in small batches have better quality outcomes. They adapt faster to bugs encountered in the wild and to unexpected market shifts. Not only that, faster is *cheaper*, because having a predictable, frequent release train forces you to drive down the cost of each release and makes the cost of any abandoned release very low.

Simply having the structures in place that *enable* continuous deployment generates the majority of the value, *even if you don’t actually push those releases out to users*. What do we mean? We don’t actually release a wildly different version of Search, Maps, or YouTube every day, but to be able to do so requires a robust, well-documented continuous deployment process, accurate and real-time metrics on user satisfaction and product health, and a coordinated team with clear policies on what makes it in or out and why. In practice, getting this right often also requires binaries that can be configured in production, configuration managed like code (in version control), and a toolchain that allows safety measures like dry-run verification, rollback/rollforward mechanisms, and reliable patching.

TL;DRs

- *Velocity is a team sport*: The optimal workflow for a large team that develops code collaboratively requires modularity of architecture and near-continuous integration.
- *Evaluate changes in isolation*: Flag guard any features to be able to isolate problems early.
- *Make reality your benchmark*: Use a staged rollout to address device diversity and the breadth of the userbase. Release qualification in a synthetic environment that isn’t similar to the production environment can lead to late surprises.
- *Ship only what gets used*: Monitor the cost and value of any feature in the wild to know whether it’s still relevant and delivering sufficient user value.

- *Shift left*: Enable faster, more data-driven decision making earlier on all changes through CI and continuous deployment.
- *Faster is safer*: Ship early and often and in small batches to reduce the risk of each release and to minimize time to market.

1 Remember the SRE “error-budget” formulation: perfection is rarely the best goal. Understand how much room for error is acceptable and how much of that budget has been spent recently and use that to adjust the trade-off between velocity and stability.

2 Dan Siroker and Pete Koomen, *A/B Testing: The Most Powerful Way to Turn Clicks Into Customers* (Hoboken: Wiley, 2013).