Making "Push on Green" a Reality

DANIEL V. KLEIN, DINA M. BETSER, AND MATHEW G. MONROE

Daniel Klein has been a Site Reliability Engineer at Google's Pittsburgh office for 3.5 years. His goal is to automate himself out of a job, so that he can get on with the work of looking for new things to do at Google. Prior to Google, he bored more easily and did a myriad different things (look on the Web for details). dvk@google.com

Dina Betser is a Site Reliability Engineer who has worked on projects such as Google Calendar and Google's large machine learning systems that maintain high quality ads. As an SRE, she often works on ensuring that products behave reliably with as little manual intervention as possible. She studied computer science as an undergraduate and master's student at MIT. dinabetser@google.com

Mathew Monroe is a Site Reliability Engineer who has worked on both the payments and anti-malvertising systems at Google. When not trying to make the Internet a safer and better place, he is trying to make running Internet services a magical experience. He has a master's in software engineering from Carnegie Mellon University and worked in distributed file systems and computer security before coming to Google. onet@google.com

Updating production software is a process that may require dozens, if not hundreds, of steps. These include creating and testing new code, building new binaries and packages, associating the packages with a versioned release, updating the jobs in production datacenters, possibly modifying database schemata, and testing and verifying the results. There are boxes to check and approvals to seek, and the more automated the process, the easier it becomes. When releases can be made faster, it is possible to release more often, and, organizationally, one becomes less afraid to "release early, release often" [6, 7]. And that's what we describe in this article—making rollouts as easy and as automated as possible. When a "green" condition is detected, we can more quickly perform a new rollout. Humans are still needed somewhere in the loop, but we strive to reduce the purely mechanical toil they need to perform.

We, Site Reliability Engineers working on several different ads and commerce services at Google, share information on how we do this, and enable other organizations to do the same. We define "Push on Green" and describe the development and deployment of best practices that serve as a foundation for this kind of undertaking. Using a "sample service" at Google as an example, we look at the historical development of the mechanization of the rollout process, and discuss the steps taken to further automate it. We then examine the steps remaining, both near and long-term, as we continue to gain experience and advance the process towards full automation. We conclude with a set of concrete recommendations for other groups wishing to implement a Push on Green system that keeps production systems not only up-and-running, but also updated with as little engineer-involvement and user-visible downtime as possible.

Push on Green

A common understanding of Push on Green is "if the tests are good, the build is good, go push it!" but we define Push on Green in three distinct ways:

1. A pushmaster says "this build is ready to go—push it." The criteria for this judgment may be based on a predefined push/freeze schedule, may have political or compliance-related considerations, may need to be coordinated with other projects, etc. Automated testing may occur, but the human is the ultimate arbiter.
2. In a continuous-build system (also known as "continuous deployment" or "continuous delivery"), a collection of smoke tests (simple tests that examine high-level functionality) and regression tests for a current build all pass at a given revision. That revision is "green" and may be pushed to production. The testing framework is the ultimate arbiter.
3. A change to a configuration file is made (which may enable or disable an existing feature, alter capacity or provisioning, etc.). This rollout may likely reuse an already green build, so the incremental tests and approvals are substantially simpler, and the reviewers and the testing framework are together the arbiters.

Other definitions are certainly possible (including the current state of the production system, so that we can consider a green-to-green transition), but above are the three that we use in this article. In all cases, we consider a system supported by Site Reliability Engineers (SRE) who are responsible for both the manual steps and the construction of the automated processes in the rollout.

Development and Deployment Best Practices

With the complexity and interconnectedness of modern systems, some development/rollout best practices have evolved which attempt to minimize problems and downtime [2, 3]. To better understand the issues involved in creating a Push on Green system, an outline of a typical Google development environment and deployment process provides a useful introduction.

Development

All code must be peer reviewed prior to submitting to the main branch to ensure that changes make sense, adhere to the overall project plan, and that bug fixes are sanely and reasonably implemented. All changes must be accompanied by tests that ensure the correct execution of the code both under expected and unexpected conditions [5]. As new libraries, APIs, and standards are introduced, old code is migrated to use them. To provide as clean and succinct an interface as possible for developers, libraries are updated and old APIs are removed as new ones are introduced [8]. The next push after a library update, then, has the same chance of breaking production as a local developer change.

The at-times draconian review process can slow down release of new code, but it ensures that whatever code is released is as likely as possible to perform as desired. And, because we live in the real world, we also extensively test our code.

Tests

Everyone at Google uses tests—the developers have unit-level, component-level, and end-to-end tests of the systems they write in order to verify system correctness. SREs have deployment tests and may call upon other tests to ensure that the newly rolled-out production system behaves the same way as it did in a testing environment. Occasionally, tests are simply a human looking at the graphs and/or logs and confirming that the expected behavior is indeed occurring. Running a production service is a compromise between an ideal world of fully testable systems and the reality of deadlines, upgrades, and human failings [7].

When developing code, all of the existing tests must continue to pass, and if new functional units are introduced, there must also be tests associated with them. Tests should guarantee that not only does the code behave well with expected inputs, but also behaves predictably with unexpected inputs.

When a bug is found, the general rule is that test-driven development is favored. That is, someone first crafts a test that triggers the buggy behavior; then the bug is fixed, verifying that the previously failing test no longer fails. The notion of "fixing the bug" may simply mean "the system no longer crashes," but a better, more laudable behavior is "appropriately adjusts for the erroneous input" (e.g., logging the problem, correcting or rejecting the data, reporting the problem back to the developers for further debugging, etc.).

We acknowledge that mistakes happen and that they happen all the time. When someone makes a mistake that adversely affects production, it becomes their responsibility to lead the postmortem analysis to help prevent future occurrences. Sometimes, a fix can be made that checks for mistakes before they happen, and at other times, changes to the underlying assumptions or processes are put into effect. For example, it was assumed that adding a column to a certain database would be harmless. A postmortem discovered that a rollback of software also required a rollback of that database, which lost the data in the new compliance-required column. This resulted in a change in procedure, where schema changes were made visible in the release prior to the one in which the code changes are visible, making rollbacks separable.

Monitoring

At Google, we extensively monitor our services. Using monitoring data, we continually strive to make our services better and to notice, through alerting, when things go wrong.

The most effective alerting looks for symptoms (and not their causes), allowing the human in the loop to diagnose the problem based on symptoms. While extensive monitoring provides great insights into the interrelation of various components of the system, ultimately all alerting should be performed in defense of a service level agreement (SLA), instead of trying to predict what may cause a service level failure [1]. Real world failures have a way of setting their own terms and conditions, and by setting (and monitoring) appropriate SLAs, it is possible to notify on "failure to meet expectations." After SLA-based alerting has been triggered, extensive monitoring enables drill-down and detailed root-cause analysis.

When this style of monitoring and alerting is in place, then not only is it possible to alert under extraordinary circumstances (e.g., surges in activity or failure of a remote system), but it is also possible to alert quickly when a new version of the software is unable to meet the demands of ordinary circumstance. Thus, an automated rollout procedure can easily incorporate monitoring signals to determine whether a rollout is good or whether the system should be rolled back to a previous version.

Updates and Rollbacks

Rolling out a new version of software is often coordinated under the supervision of a pushmaster, and may involve updating a single job, a single datacenter, or an entire service. Where possible, canaries are used to test proper functioning of revised software. Named for the proverbial canary in a coal mine, the canary instances are brought up before the main job is updated, in one last pre-rollout test of the software. The canary typically receives only a small fraction (perhaps 1% or 0.1%) of production traffic; if it fails or otherwise misbehaves, it can be quickly shut off, leaving the rest of the code as-yet not updated, and returning the service to normal operation for all users. Canarying can also be done in stages, per job, by orders of magnitude, by datacenter, or by geographic region served.

Services handle updates in a few different ways. Disparate code changes must be integrated into "builds" (where the binaries are created from various sources and libraries), and the timing of the release of these builds is often planned well in advance. A production binary not only comprises the directly edited code of the team, but also those libraries released by teams that run supporting services utilized by the service. Many .jar/.so files are statically associated into a binary package, and there is no universally followed release cycle; each team produces new versions on their own timetable. Therefore, whenever a new binary is built for release, the changes that comprise it may come from far and wide.

Configuration changes are also considered rollouts. These may be in the form of runtime flags specified on the command line, or in configuration files read on startup; both require a job to be restarted to take effect. There may also be database updates or changes that impact the behavior of a system without restarting it. Configuration changes have the same potential to induce failure, but they also benefit from the same types of automation.

Safely Introducing Changes

Consider how you would add a new feature to a service. One common practice incorporates the following steps:

1. Create a new runtime configuration directive for a new feature, with the default value set to "disabled." Write the code that uses the new feature, and guard that code with the new directive (so that the new code is present but is disabled by default).
2. Release the new binaries with no references to the new directive in any configuration file. The feature should remain inactive, and failed rollout requires a rollback to the previous version of the binaries.
3. Update the configuration files to include the presence of the new directive (but explicitly specify that it is disabled), and restart the current system. The feature should continue to remain inactive, and a failed rollout simply requires a rollback to the previous version of the configuration files.
4. Update the system configuration files to enable the new directive in the canary jobs only, and restart the current version of the binaries in the canaries. A failed rollout simply requires turning off the canaries and later rolling back to the previous version of the configuration files.
5. Update the remainder of the jobs with the directive enabled. Failures are less likely at this stage since the canaries have not died or caused problems, but failure simply requires a rollback to the previous version of the configuration files. At this point, the new feature is enabled.
6. In a subsequent release, alter the code so that the directive is now enabled by default. Because the directive is currently enabled in the configuration file, changing the default flag value to match the specified configuration value should have no effect on behavior, so rolling out this change is usually deferred to occur along with a collection of other changes. However, a failed rollout requires a rollback to the previous version of the binaries.
7. Update the system configuration files to make no further reference to the directive—it is "live" by default. A failed rollout simply requires a rollback to the previous version of the configuration files.
8. Edit all conditional execution of code to always execute, since that is now the implicit behavior. A failed rollout requires a rollback to the previous version of the binaries.
9. Delete the now-unused definition of the directive in the code. A failed rollout at this stage is almost certainly due to a configuration file error, because the directive itself should not have been used since step 7—so a binary rollback is probably not needed.

Requiring nine steps to fully add a new feature may seem like overkill, but it ensures the safe release of new code. Additionally, the steps involved can take place over many months and many disparate releases. Complicating this process is the fact there may be dozens of such changes occurring in parallel, some simultaneously in-flight but starting and ending at widely different times. Automating as much of the rollout process as possible can help mitigate the overhead of keeping track of changes.

Types of Configuration Changes

We consider two kinds of configuration changes:

1. Changes to configuration directives that require job restart.
2. Changes to configuration directives that are automatically picked up by jobs.

There are advantages and disadvantages to both. When job-restart is required, one type of job can be updated with the configuration directive, regardless of which other jobs have the

directives available to them. This yields fine-grained control, but also requires that all restarts be tightly coordinated, so that an unrelated job restart does not pick up unintended configuration changes.

When jobs automatically pick up changes, configuration changes are more global in scope. While this has the advantage of easily automating changes on a large scale, it also means that greater care must be taken in hierarchically specified configuration-files to ensure that only the intended jobs are changed. In a real-world system with thousands of options across hundreds of jobs, it is easy for the hierarchy to break down or become unmanageable, riddled with special cases.

In both cases, great care must be taken to restrict the inadvertent propagation of unintended changes. Simplicity and flexibility are at odds using either scheme, while reliability and configurability are the goal of both.

Towards Push on Green

Much of the danger in releasing new code can be mitigated, but the process still has a large amount of mechanical human oversight, and the purpose of the Push on Green project is to minimize as much of this toil as possible.

Historical State of the Practice

We begin by examining a "sample service" at Google. The rollout process starts with a push request being filed against the current on-call, detailing the parameters of the rollout (the version number, people involved, and whether the push is for canary, production, or some other environment).

Previously, this service had a largely manual rollout process, comprising a collection of scripts that were manually invoked following a rollout run-book. The first step towards Push on Green was to replace this with a more automated process that effectively performed the same steps.

For the production jobs, the following steps are executed for binary pushes or command-line flag changes. The automated rollout procedure updates the push request at each step.

1. Silence alerts that will be known to fire during a rollout (for example, warnings about "mixed versions in production").
2. Push the new binaries or configuration changes to the canary jobs in a datacenter currently serving traffic.
3. Run the smoke tests, probers, and verify rollout health.
   a. If the tests fail, notify the on-call, who may initiate a canary rollback or bring down the canaries.
   b. Some health-check failure conditions are the result of an accumulation of errors, so some services require that tests can only pass after a sufficient amount of time is allowed for the binary to "soak."
4. Push the binaries to the remainder of the jobs in that datacenter.
5. Unsilence the previously silenced alerts.
6. Rerun smoke tests (step 3); if the tests pass, repeat steps 2–5 for each of the other datacenters.

This process still entails a lot of manual work. A push request must be filed for each rollout, and the binaries for each of the jobs must be built. The binary packages must be tagged, annotated, or accounted for in some way (so that the rollout pushes the correct binaries), and there are assorted handoffs between the release engineer, test engineers, and Site Reliability Engineers that limit the number of rollouts per day to only one or two. Although alerting in case of problems is largely automated, the entire push process must still be baby-sat to ensure correct functioning.

State of the Art—Recent Developments

Once the rollout process was made to be a largely push-button operation, steps were taken to make it more automated with even fewer buttons to push. These steps included:

RECURRING ROLLOUTS FOR INFRASTRUCTURE JOBS

Our services consist of jobs that are specific to the operation of the service and jobs and software packages that are maintained by other teams but that we configure for our service. For example, the monitoring and alerting jobs are standardized jobs that are custom-configured. The monitoring teams update the binaries that are available to use, but it is the responsibility of each service to periodically restart their jobs with new binaries at a time that is safe for the service involved.

Our recurring rollout updates those jobs maintained by other teams on a daily basis, keeping them current, even when there are service-specific production freezes. This recurring rollout was the first step to Push on Green automation.

ROLLOUT LOCKING

Some rollout steps have the potential to interfere with other rollouts. For example, if we are doing a production rollout, we do not want to simultaneously do an infrastructure rollout, so that we know which rollout to blame in case of a problem. With inter-rollout locking, we can also provide an inter-rollout delay, so that the effects of each rollout are clearly delineated from each other.

ROLLOUT REDUNDANCY

Reliability of the rollout system is just as important as reliability of the system being supported. A rollout that fails part way through due to a failure in the rollout automation can leave a production service in an unstable and unknown state. As such, the rollout systems should run in a replicated production configuration to guard against failure.

TIME RESTRICTIONS
We have on-call teams spanning multiple continents and separated by anywhere between five and ten time zones. The next step towards Push on Green was to provide a rollout schedule that took into account waking hours, weekends, and holidays (with different cultural norms). The software that recognizes these holidays was made to be easily configurable so other teams could reuse it for similar automation that includes other countries.

ROLLOUT MONITORING
The on-call must often consult a collection of logs to determine when a rollout started and ended, and attempt to correlate that data with problems that are reflected in latency and availability graphs.

Push on Green avoids manual searches of disparate sources of information, so another automation component was creating variables in the rollout system that could be queried by the monitoring and alerting system. This has enabled us to overlay a graph that visually displays the start and end of rollout components on top of the latency and availability graphs, so it is easy to see whether an inflection point in a graph exactly corresponds to a rollout.

AUTOMATIC ROLLOUT OF CONFIGURATION CHANGES
Adding a new configuration option requires nine discrete steps, and half of these are manual processes. The next step in automation is to have a single recurring rollout simply look for changes in the revision control system which match two specific criteria:

- Affect a specific set of files
- Have approvals from the right people

The rollout then automatically creates and annotates a new push request, and processes the rollout steps. When this is combined with rollout locking and time restrictions, we have an automatic Push-On-Green system (according to our third definition in "What is Push on Green"), dramatically reducing engineer toil. This cautious first step does not eliminate the human component of arbitration but, instead, removes much of the checklist labor that needs to be done.

State of the Art—Future Plans
Some of what follows is work in progress, and some of it is still in the planning stages, but all of it logically follows the work that has been accomplished so far in that it advances the automation of our rollout processes.

- Rollback-feasibility rollout step: use the testing environment to roll out a new binary, then roll it back after some traffic has been routed to the new jobs. If the smoke and regression tests still confirm job health, then the rollout can safely proceed in production jobs.
- Continuous delivery: automatically create push requests for versioned builds that pass the required tests, taking the "push early, push often" philosophy to its logical extreme. We can then use the monitoring data in the staging environment to ascertain which builds we believe are safe to push to production.
- Rollout quotas: we may want to limit the number of rollouts per day, or limit the number of rollouts that a single user can initiate, etc.
- Pushing Green-on-Green: perform a correlative analysis of a collection of indicators to determine overall system health before performing a new push. The system may not be out of SLA, but it might be dangerously close to the edge. Unless a service is currently green, it is a bad idea to automatically perform a new rollout.

We Still Need Manual Overrides!
Regardless of how much we want to automate everything, some processes will stay manual for now:

- We will always need the ability to do a manually induced rollback.
- Some tests are flaky. It may be the case that a build is not green due to a flaky test or that the system is healthy but the tests say otherwise. We need to be able to manually override the requirement for "green"; sometimes we believe that the problem being reported does not exist, or the problem exists but the rollout will fix it.
- Every automated system needs a big red button to halt execution. If we have the required means of switching off automatic rollouts, then we still need a way to do manual rollouts of any kind.

Real Numbers, Real Improvement
Since introduction of Push on Green, the on-calls in our service have experienced the improvements seen in Table 1.

We have increased the number of rollouts by an order of magnitude in two years, while at the same time saving almost a whole SRE FTE (and freeing the developers from much of their involvement in rollouts). Once our rollout system begins automatically detecting green conditions, we expect that the number of rollouts will increase even more, and the level of engineer engagement will continue to decrease.

Towards Push on Green: Recommendations
The fundamental components of an automated rollout system are as follows:

Monitoring and Alerting
If you don't know how your system is behaving in a measurable way, you only have assumptions to deal with. While naked-eye observation can tell you the gross symptoms ("the page loads

	Original (manual) rollouts 2012	Mechanized rollouts 2013	Semi-automated rollouts 2014
Rollouts/ month	12–20	60	160
Time saved for on-call/ month	0 hours (baseline)	20 hours	50–60 hours

Table 1: Rollouts have increased by an order of magnitude over two years, while time spent on them has decreased.

slowly" or "that looks better"), you need automated monitoring at a fine-grained component level to give you insights as to *why* problems are happening or whether changes have had the desired effect.

- Monitoring needs to be performed on production systems, and monitoring is different from profiling. Profiling helps you find hotspots in your code with canned data; monitoring tracks the responsiveness of a running system with live data.
- Monitoring needs to be combined with alerting so that exceptional conditions are rapidly brought to the attention of production staff. Although it is possible to eyeball a collection of graphs [4], a well-monitored system has far more variables than any human can reasonably scan by eye (in a Google production service, it is not unusual to monitor millions of conditions with tens of thousands of alerting rules).
- The state of monitored variables and resulting alerts must be available in a form that allows programmatic queries, so that external systems can determine the current and previous state. Both these data points are needed to make determinations as to whether things have improved or degraded.
- If at all possible, separately monitor canary versions of jobs and their non-canary production counterparts. If all other things are equal (traffic, load, data sets, etc.), then it is possible to assess the health and quality of a canary job relative to the previous version of production.

Builds and Versions

A repeatable mechanism for building new binary releases must be part of the overall release cycle. Static builds ensure that what you build and test today is exactly the same as what you push next week. While different components may have different build procedures, they all must be regularized into some standard format.

- Versions must be tracked, preferably in a way that makes it easy for both the developers and release engineers to correlate a given build with a specific production service. Rather than sequential version numbers (like v3.0.7.2a), we recommend versions that incorporate the date and some other distinguishing nomenclature (such as tool_20140402-rc3) so that a human can readily correlate versions.
- Versions should be tagged, annotated, or otherwise consistently accounted for. This means that rather than "push version X to production," you should "mark version X with the production tag" and then "push the current production version." This allows for separation of responsibilities (developers build releases, release engineers tag them, and production engineers update the jobs) while still maintaining a coherent view of the service.
- Builds should be tested at a number of levels, from unit tests through to end-to-end tests and live production tests. Finding problems earlier in the process helps eliminate them faster.

Scripted and Parameterized Rollouts and Rollbacks

A systematized and regularized rollout procedure must exist. If automated steps are interspersed with manual steps, there must also be checks to ensure that all of the manual steps have been properly performed.

- As many steps in the rollout process as possible should be fully automated. If customizations need to be done on a per-rollout basis, these should be specified in a configuration file so that nothing is left to the memory of the person starting the rollout. This is especially important when rolling back to some previous version.
- Rollouts steps (and thus the entire rollout) should be idempotent. A rollout should not suffer from any step being performed twice. If a rollout fails, rerunning the rollout should either fail in the same way or cure the problem.
- A rollback should be the same as a rollout to a previous version. This is more of an ideal goal than a practical reality—there will always be some special case where a schema change cannot be rolled back. However, the more regular the rollout process, the less likely this will happen, and the more likely it is that developers will avoid changes that cannot be rolled back.

Process Automation and Tracking

Once the basic infrastructure is in place for scripted rollouts, you can contemplate automatically detecting and rolling out new versions.

- Once the process of versioning has been regularized, you can query the build system to determine when a new version is available, and roll it out when the appropriate preconditions have been met (i.e., build and tests are green, the version number is incrementally larger, production is green, etc.).
- When a new version is rolled out, the monitoring and alerting should be queried at various stages in the rollout to ascertain

whether there are any problems. If any anomalous behavior is detected, the rollout should be paused at whatever state it is in until a determination of overall rollout health can be made (since rollouts should be idempotent, it is also valid to abort the rollout, with the expectation that it can be restarted later).

- An anomaly in your monitored data may be the responsibility of your just-pushed system, but it may also be the result of some dependent system having been coincidentally just pushed. Coordinating rollouts with dependent teams can avoid this problem.

Conclusions

Building our Push on Green rollout system has been an evolutionary process. It has worked well because of the great degree of caution that we have exercised in incrementally adding functionality and automation. Although we are all in favor of having computers do our jobs for us, we are also averse to the disasters that mindless automation can bring.

We are only one of many teams at Google who are automating their rollout process. The needs of the project and the constraints of the production environments influence how each of these teams perform their jobs. However, regardless of the particulars, each group has addressed the same concerns and exercised the same degree of caution in handing over the reins to an automated system. With apologies to Euripides, "The mills of process automation grind exceedingly slow, but grind exceedingly fine..." Anyone can do it—just be prepared for a long haul.

References

[1] Alain Andrieux, Karl Czajkowski, Asit Dan et al., "Web Services Agreement Specification (WS-Agreement)," September 2005: http://www.ogf.org/documents/GFD.107.pdf.

[2] J. R. Erenkrantz, "Release Management within Open Source Projects," in *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, May 2003.

[3] J. Humble, C. Read, D. North, "The Deployment Production Line," in *Proceedings of the IEEE Agile Conference*, July 2006.

[4] D. V. Klein, "A Forensic Analysis of a Distributed Two-Stage Web-Based Spam Attack," in *Proceedings of the 20th Large Installation System Administration Conference*, December 2006.

[5] D. R. Wallace and R. U. Fujii, "Software Verification and Validation: An Overview," *IEEE Software*, vol. 6, no. 3 (May 1989), pp. 10, 17.

[6] H. K. Wright, "Release Engineering Processes, Their Faults and Failures," (Section 7.2.2.2) PhD Thesis, University of Texas at Austin, 2012.

[7] H. K. Wright and D. E. Perry, "Release Engineering Practices and Pitfalls," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)* (IEEE, 2012), pp. 1281-1284.

[8] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, "Large-Scale Automated Refactoring Using ClangMR," in *Proceedings of the 29th International Conference on Software Maintenance (IEEE, 2013)*, pp. 548–551.