Canary Analysis Service

AUTOMATED CANARYING QUICKENS DEVELOPMENT, IMPROVES PRODUCTION SAFETY, AND HELPS PREVENT OUTAGES.

ŠTĚPÁN DAVIDOVIČ WITH BETSY BEYER

In 1913, the Scottish physiologist John Scott Haldane proposed the idea of bringing a caged canary into a mine to detect dangerous gases. More than 100 years later, Haldane's canary-in-the-coal-mine approach is also applied in software testing.

In this article, the term *canarying* refers to a partial and time-limited deployment of a change in a service, followed by an evaluation of whether the service change is safe. The production change process may then roll forward, roll back, alert a human, or do something else. Effective canarying involves many decisions—for example, how to deploy the partial service change or choose meaningful metrics—and deserves a separate discussion.

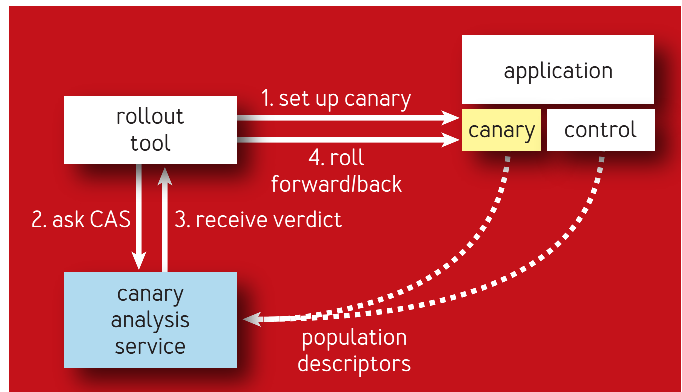
Google has deployed a shared centralized service called CAS (Canary Analysis Service) that offers automatic (and often autoconfigured) analysis of key metrics during a production change. CAS is used to analyze new versions of binaries, configuration changes, data-set changes, and other production changes. CAS evaluates hundreds of thousands of production changes every day at Google.

WORKFLOW WITH CAS

CAS requires a very strict separation between modifying and analyzing production. It is a purely passive observer: it never changes any part of the production system. Related tasks such as canary setup are performed outside of CAS.

In a typical CAS workflow (shown in figure 1), the rollout tool responsible for the production change deploys a change to a certain subset of a service. It may perform some basic health checks of its own. For example, if pushing a new version of an HTTP server causes a process

FIGURE 1: **DIAGRAM OF TYPICAL CAS WORKFLOW**



restart, the rollout tool might wait until the server marks itself as able to serve before proceeding. (This may also inform the deployment speed of the production change. This rollout tool behavior is not specific to canarying.)

This subset of production now constitutes the *canary population*. By conducting an A/B test compared to a control population, CAS answers the question, "Is the canary meaningfully worse?" The control population is a (possibly strict) subset of the remainder of the service. Importantly, CAS is not trying to establish absolute health.

The population should be as fine-grained as possible. For example, an application update can use a global identifier of that particular process, which at Google would be a BNS (Borg Naming Service) path. A BNS path is structured as /bns/<cluster>/<user>/<job name>/<task number>. The job name is a logical name of the application, and task number is the identifier of a particular instance.[2] For a kernel update, the identifier might be machine hostname: clearly, multiple processes can run on the same machine, but (modulo virtualization) you are limited to a single running kernel, so granularity is defined at the machine level. Granularity allows the caller to slice and dice the overall service with no restrictions at runtime, and it makes a static preexisting canary setup unnecessary.

Once the canary population is set up, the rollout tool requests a verdict from CAS. This request specifies the canary and control populations, as well as a time range for each member of that population. An entity's canary status can be ephemeral: the canary only became a canary at some specific point in time; before that time, it did not have the tested production change.

Granularity allows the caller to slice and dice the overall service with no restrictions at runtime, and it makes a static preexisting canary setup unnecessary.

The request also contains a reference to a user-supplied configuration, if one exists. As discussed in the following section, CAS tries to provide value, even without external configuration, by enforcing things we hold generally true.

CAS provides a point-in-time verdict after evaluation: a simple PASS or FAIL, meaning the system is performing either the same or in a dangerously anomalous way. (A third option, NONE, is also possible if underlying infrastructure was unavailable and CAS could not reach a verdict. Clients commonly treat this the same as if they could not reach CAS.) The signal must be clear and unambiguous for the rollout tool to take an action such as rolling forward, rolling back, or alerting a human. CAS intentionally does not provide a confidence score, p-value, or the like: that would imply that the rollout tool has logic to determine when to take a real-world action. Keeping this decision centralized allows better reuse and removes the risk of creating artificial confidence scores from a meaningless heuristic.

Default tooling integration and zero configuration option

CAS quickly gained extensive coverage across all of Google by integrating with all major tools used to change production, including tools to roll out new binaries, configurations, and data sets. Widespread integration required a conservative integration approach: in some cases, concessions in the analysis quality had to be made in favor of not inconveniencing users.

Yet another barrier to entry was removed by not requiring a canary setup in order to start using CAS. If the user does not specify a configuration, default analyses are performed across metrics that can be reasoned about

across the board. CAS auto-discovers features of canaried systems, such as whether the binary is C++ or Java, or which RPC (remote procedure call) methods receive significant traffic, then chooses analyses to run (e.g., RPC error ratio) for those features. Google's infrastructure homogeneity makes this largely successful.

SERVICE DESIGN

CAS's relatively simple high-level interactions are enabled by a fairly complex system under the hood.

Public API in detail

CAS's API has two RPC calls: `Evaluate()` and `GetResult()`.

`Evaluate()` is given one or more trials and returns a unique string identifier, `Evaluation ID`, which is a fully qualified URL to the CAS user interface. This simple trick has made it quite trivial to insert these links into various rollout tools, since it means they do not need any additional client-side logic to figure out how to turn an identifier into a URL. Trials are pairs of canary and control populations and the time range during which they should be compared; if the end time of the time range is left unset, CAS is free to decide how much time the evaluation needs. This means striking a balance between delaying the evaluation too much and not having enough data to reach a meaningful conclusion. In practice, at least five minutes of data are required. The call is reasonably fast (typically under a second), and the API retains the resulting data indefinitely (or at least until garbage collection of evaluations that are clearly no longer relevant). The client sends one RPC for a logical evaluation.

The CAS API does not promise that the evaluation will start when the `Evaluate()` call is sent. Instead, analysis starts on `GetResult()`, since that indicates someone is interested in the result. As an optimization, the analysis actually starts on `Evaluate()`, but in order to set appropriate expectations with the client, this optimization is not part of the API definition.

`GetResult()` takes one parameter: the `Evaluation ID`. This RPC blocks until the analysis process is finished, which can take between a few seconds and a few minutes after the end time of the request; `GetResult()` is idempotent.

For the sake of reliability, CAS developers designed the system with two calls. This setup allows the system to resume processing a request without requiring complex client cooperation. This reliability strategy played out in practice when a bug in a library made all CAS processes crash every five to ten minutes. CAS was still able to serve all user requests, thanks to the robust API.

There are some obvious alternatives to this design. CAS developers decided against using a single long-running RPC: since these calls are fundamentally point-to-point connections between two Unix processes, disruption (e.g., because one process restarted) would lead to a full retry from the client side. The original design doc included a large number of options, each with tradeoffs tied to nuanced properties of Google's infrastructure and requirements.

Evaluation structure

While the RPC returns only a simple PASS/FAIL verdict, the underlying analysis consists of several components.

The lowest-level unit is a *check*, a combination of time series from the canary population, time series from the control population, and a statistical function that turns both time series into an unambiguous PASS/FAIL verdict. Some example checks might be:

- Crash rate of the canary is not significantly greater than the control.
- RPC error ratio is not significantly greater than the control.
- Size of data set loaded in memory is similar between canary and control.

As mentioned in the API description, each evaluation request can define multiple trials (i.e, pairs of canary and control populations). Evaluation of each trial results in a collection of checks. If any check in any trial fails, the entire evaluation is declared a failure, and FAIL is returned.

Currently, trials are implemented to be fairly independent, though a given evaluation request might have multiple trials if they look at two related but different components. For example, consider an application with a front end and a back end. Changes on the front end can trigger bad behavior on the back end, so you need to compare both:

- The canary front end to the production front end.
- The back end receiving traffic from the *canary* front end to a back end receiving traffic from the *production* front end.

These are different populations, possibly with different metrics, but failure on either side is a potential problem.

Configuration structure

What exactly does a user-defined configuration

entail? While the design phase of CAS involved lengthy philosophical discussion about the nature of configuration, the primary aim was simplicity. The CAS developers didn't want to force users to learn implementation details in order to encode their high-level goals into a configuration. The intent was to ask users only a few questions, as close to the user's view of the world as possible.

The individual checks that should be executed for each matching trial define what information is needed. For each check, the user specifies:

- What it should be called.
- How to get the time series for the particular metric.
- How to turn these time series into a verdict.

The user can also include optional pieces of information, such as a long-form description.

Monarch is the typical source of monitoring data for time series.[1] The user specifies an abstract query, and the canary and control populations are determined at *runtime* in the RPC that requests evaluation. CAS has a flexible automatic query rewrite mechanism: at runtime, it rewrites an abstract query to specialize it to fetch data only for a particular population.Say a user configures a query, "Get CPU usage rate." At runtime, CAS rewrites that query as "Get CPU usage rate for job foo-server replicas 0, 1, 2." This rewrite happens for both the canary population and the control, resulting in two queries.

It is possible, although uncommon, to specify different queries for the canary and the control. The queries are still subject to rewriting, which guarantees that they will fetch data only for the objects that are actually being evaluated.

To simplify configuration, there are also *common*

queries. These are canned queries curated by the CAS team, such as crash rate, RPC server error ratio, and CPU utilization. These offer known semantics, for which CAS can provide better quality analysis.

Finally, there needs to be a way to turn the time series (possibly multiple streams) obtained by running the Monarch query for the canary and control populations into an unambiguous verdict. The user can choose from a family of tests. Some tests (such as Student's t-test) have a clear statistical origin, while others contain custom heuristics that attempt to mimic how a human would evaluate two graphs.

As discussed later, automatic analyses are applied if a user chooses the default configuration, as well as on user-supplied queries if the user does not specify a statistical test.

SYSTEM COMPONENTS AND REQUEST FLOW

Figure 2 illustrates the components of the CAS system. This section describes the role of each component and the CAS request flow.

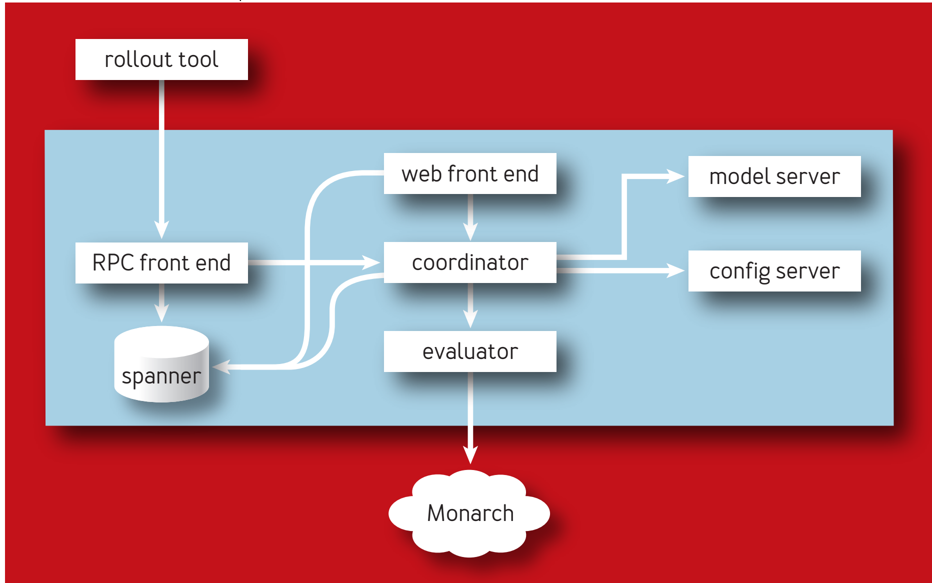
Spanner database

The Spanner database is a shared synchronization point for the evaluation flow; almost all components write to it. It is the canonical storage for evaluation progress and final status.

RPC front end

The rollout tool sends `Evaluate()` calls to the RPC front end, which is intentionally very simple. The front end generates a unique identifier for the evaluation, stores the

FIGURE 2: **DIAGRAM OF MAIN COMPONENTS OF CAS**



entire evaluation request in the database (with the unique identifier as primary key), and returns the identifier.

`GetResult()` calls also land on the RPC front end, which queries the database to see if a coordinator is already working on the evaluation. If so, the RPC front end sends an `AwaitEvaluation()` RPC to the coordinator, which blocks until the evaluation is complete. If the coordinator isn't tracking the evaluation (e.g., if a restart resulted in lost state) or if no coordinator is assigned, the RPC front end chooses a coordinator, stores that information in the database, and calls `AwaitEvaluation()`. These retries are limited.

If the evaluation has already finished, the RPC front end does not contact the coordinator and immediately returns

the results from the database to the caller.

It is very cheap for the RPC front end to handle parallel `GetResult()`s. Selecting one coordinator avoids duplication of expensive work unless the client requests two duplicate and independent evaluations.

Coordinator

The coordinator keeps all evaluations it's currently processing in memory. Upon `AwaitEvaluation()`, the coordinator checks whether the evaluation is being processed. If so, the coordinator simply adds this RPC to the set of RPCs awaiting the result.

If the evaluation is not being processed, the coordinator transactionally takes ownership of the evaluation in the database. This transaction can fail if another coordinator (for whatever reason, such as a race condition) independently takes ownership, in which case the coordinator pushes back to the RPC front end, which then contacts the new canonical coordinator.

Upon receiving a new evaluation, the coordinator does the following:

1. Retrieves fully qualified and unambiguous expanded configuration from the config server. The coordinator now has the full set of all checks to run.
2. Fans out each check to evaluators.
3. Calls the model server to obtain predicted behavior for checks, simultaneously reporting the results of the checks in the current evaluation.
4. Responds to all waiting `AwaitEvaluation()` RPCs with the final verdict.

The coordinator checkpoints progress to the database

throughout. Checkpoints occur after a coordinator receives a fully qualified configuration and asynchronously as evaluators return check-evaluation requests. If the coordinator dies, a new one takes over, reads progress from the database, and continues from the last coordinator's checkpoint.

Configuration server

The configuration server looks up and fully expands a configuration that matches an evaluation.

When the configuration is explicitly referenced in a request, lookup is trivial. If the configuration isn't explicitly referenced, a set of automatic lookup rules search for the user's *default config*. These lookup rules are based on features such as who owns the canaried service.

The CAS-submitted configuration is generic: it might say something like "Fetch HTTP error rate," without specifying where to fetch the error rate. In the typical flow, the rollout tool identifies the current canary and passes this information along to CAS when the evaluation is requested. As a result, the configuration author cannot necessarily predict the canary population.

To support this flexibility, the configuration server expands configuration and canary/control population definitions to specify exactly what data is requested. For example, the user's "Fetch HTTP error rate" becomes "Fetch HTTP error rate from these three processes for canary data, and from these ten processes for control data." From a user's point of view, after configuring the generic variant, the "right thing" happens automatically, removing any need to define a dedicated canary setup

before canarying (although users can define such a setup if they have other reasons to do so).

Besides evaluations, the configuration server also receives configuration updates, validates updates for correctness and ACLs (access control lists), and stores these updates in the database.

Evaluator

The evaluator receives a fully defined configuration (after the expansion already mentioned) for each check, with each check in a separate RPC. The evaluator then:

1. Fetches time series for both canary and control data from the appropriate time series store.
2. Runs statistical tests to turn the resulting pair of sets of time series into a single PASS/FAIL verdict for each statistical test (pair because of canary/control; sets because it's possible, for example, to have a time series per running process and have many processes in the canary or control groups).

If a user configures a statistical test, then the evaluator runs only that test. If the user opts for autoconfiguration, however, the evaluator may run dozens of tests with various parameters, which generate data that feeds into the model server.

The evaluator returns the data from tests and any potential metadata (such as errors talking to time-series stores) to the coordinator.

Model server

The model server performs automatic data analysis. After evaluation, the coordinator asks the model server for

predictions. The request contains information about the evaluation and all observed verdicts from the evaluator.

For each observed verdict, the model server returns its expected verdict for that particular evaluation. It returns this information to the coordinator, which ignores results of statistical functions for predicted failures when deciding the overall verdict. If the model server predicts failure because said failure is typical behavior, this behavior is deemed a property of the evaluated system and not a failure of this particular canary evaluation.

AUTOCONFIGURATION

Canarying properly is a complex process, as the user needs to accomplish these nuanced tasks:

- Correctly identify a meaningful canary deployment that creates a representative canary population with respect to the evaluation metrics.
- Choose appropriate evaluation metrics.
- Decide how to evaluate canaries as passing or failing.

CAS eases the burden by removing the most daunting of these tasks: evaluating what it means for a time-series pair to pass or fail. CAS builds upon the underlying argument that running reliable systems shouldn't require in-depth knowledge of statistics or constant tuning of statistical functions' parameters.

CAS uses behavior learning that's slightly different from the general problem of anomaly detection for monitoring. In the CAS scenario, you already know that a service is being changed, and exactly where and when that change takes place; there is also a running control population to use as a baseline for analysis. Whereas anomaly detection

for monitoring triggers user alerts (possibly at 4 a.m.), bad CAS-related rollouts are far less intrusive—typically resulting in a pause or a rollback.

Users can opt out of autoconfiguration by specifying a test and its parameters manually.

Online behavior learning

In the simplest terms, we want to determine the typical behavior of the system being evaluated during similar production changes. The high-level assumption is that bad behavior is rare.

This process takes place online, since it must be possible to adapt quickly: if a behavior is anomalous but desirable, CAS fails the rollout; when the push is retried, CAS needs to adapt.

Adaptive behavior poses a risk if a user keeps retrying a push when an anomaly is actually dangerous: CAS eventually starts treating this risky behavior as the new norm and no longer flags it as problematic. This risk becomes less severe as the automation becomes more mature and reliable, as users are less inclined to blindly retry (assuming an incorrect evaluation) and more inclined to actually debug when CAS reports a failure.

Offline supporting processes can supplement the standard online learning.

Breakdown of observations

Intuitively, you know that comparing the same metrics across different binaries may yield different results. Even if you look at the same metric (RPC latency, for example), a stateful service such as BigTable may behave

quite differently from a stateless web-search back end. Depending on the binary being evaluated, you may want to choose different parameters from the statistical tests, or even different statistical tests altogether.

Rather than attempting to perform in-depth discovery of potential functional dependencies, CAS breaks down observations across dimensions based upon past experiences with running production systems. You may well discover other relevant dimensions over time.

Currently, the system groups observations by the following factors:

- *Data source.* Are you observing process crash rate, RPC latency, or something else? Each data source is assigned a unique identifier by fingerprinting the configuration and some minor heuristics to remove common sources of unimportant differences.
- *Statistical function and parameters.* This could mean, for example, a t-test with significance level 0.05. Each distinct statistical function and parameter set is assigned a unique identifier.
- *Application binary.*
- *Geographical location.* This refers to the locations of the canary and control.
- *Process age.* Has the process recently restarted? This helps distinguish a configuration push (which might not restart the process) from a binary update (which likely would).
- *Additional breakdowns,* such as different RPC methods. For example, reading a row in BigTable may behave very differently from deleting the entire table. This breakdown depends on the supplied metric.

- *Time of observation*. This is kept at daily granularity for system efficiency.

These factors combine with the count of each observed verdict to make a *model*. A model knows *only* identifiers—it has no understanding of the data source, statistical functions, or their parameters.

Prediction selection

All models pertaining to a particular binary are fetched across all statistical functions for which there is an observation, and across all data sources.

For each statistical function and each data source, the weighted sum of the past observed behaviors is calculated for each possible result. Similarity is weighted both by heuristic similarity of features (process age and geographical location) and by the age of the model. Because additional breakdowns such as RPC methods don't have a usable similarity metric, the additional matching breakdowns are simply filtered in, with no further weighting.

For a single statistical function and a single data source, we generate a score for each possible verdict (PASS, FAIL, or NONE). We calculate this score from a weighted sum of past observations. Weighting is based upon factors like age of the observation and similarity of the observation to the current situation (for example, do both observations pertain to the same geographic location?).

Each statistical function has a minimum pass ratio. The ratio *sum[PASS] / (sum[PASS] + sum[FAIL] + sum[NONE])* must be greater than the minimum for a PASS prediction. Otherwise, the prediction is FAIL.

This ratio allows CAS to impose a notion of strictness on various functions, while being tolerant of "normal" volatile behavior. For example, consider two statistical functions: one that tolerates only 1 percent deviation between canary and control, and one that tolerates 10 percent. The former can be given a very high minimum pass ratio, and the latter a lower one. If the metric fluctuates more than 1 percent in normal operation, CAS quickly learns that behavior and stops flagging it. If that fluctuation is a one-off, CAS flags it, the system recovers, and over time CAS relearns that normal behavior includes only deviations under 1 percent. CAS intentionally takes longer to learn normal behavior for larger tolerated fluctuations, so in this example, CAS will learn at a slower rate for the 10 percent case.

Bootstrapping

When a user initially submits a configuration that evaluates a metric, no past behavior exists to use for prediction. To bootstrap such cases, CAS looks for past evaluations that *could have* used this config and runs those evaluations to collect observations for the model server. With enough recent evaluations, CAS will already have useful data the first time a user requests an evaluation.

If such bootstrapping is not possible, the model server reverts to the most generous behavior possible.

Arbitrary input analysis

The behavior-prediction mechanism is also the first attempt at *arbitrary input analysis*, which allows modeling behavior for tests when there is no prior knowledge of what they are about.

When a user configures canarying on RPC error ratio,

CAS knows in advance that the values are between 0.0 and 1.0, and that higher is worse. For a user-supplied query against the monitoring data, CAS has no such knowledge and can only apply a battery of tests and observe the differences.

Despite some significant issues (discussed later), the CAS development team chose this approach because they were confident that it would have relatively few unexpected risks. It still greatly improves automated canarying. The developers are actively working on improvements.

FUTURE WORK

Time series aggregate models

While the meta-analysis of the results of hard-coded statistical functions has worked well for the initial launch of automatic configuration, this approach is crude and inflexible. Rather than storing results of statistical tests without any knowledge about the time series that caused them, CAS could store data about the time series.

Each statistical function that CAS supports requires different data from the time series. We could attempt to extract constant-size aggregate views on this data, one for each statistical test. For example, a Student's t-test view on the time series could be the mean value for both populations, the population sizes, and variance estimation.

This aggregated view from many past observations would allow synthesizing a single test for each statistical function, with the correct parameters chosen based on past data and some policy.

This work would essentially replace half of the current autoconfiguration system.

Further observation breakdowns

Observation breakdowns turned out to be the biggest contribution of the model server to CAS as a whole, so the development team plans to expand this feature. Adding more breakdowns entails additional computational/storage costs and, therefore, needs to be undertaken carefully given CAS's large scale.

While CAS currently has breakdowns based on the object of evaluation, this could be expanded to breakdowns by type of canarying. Anecdotally, there have been major differences in canary behavior when observed using before/after tests versus simultaneous tests of two populations. The size of the canary population in relation to the control population and the absolute sizes of the populations can also provide meaningful breakdowns.

Future work could determine if these additional breakdowns are worthwhile, and at what granularity to perform them. Automatically generated decision trees may also be an option.

Priming with steady state data

CAS sees only production changes. Currently, it does not learn that a particular metric is erratic even in steady state.

Data about metric behavior outside of production changes could be used to define the typical noise in the data. CAS would fail a canary only if the deviation is above this typical noise level. The noise data could come from analyzing only the control population for every evaluation, because the control population is expected to have no production changes.

KNOWN ISSUES

Same environment overfitting

CAS autoconfiguration's most significant issue is overfitting data when there is already a rich history of past observations in exactly the same environment. In this scenario, only the historical data for that environment is used.

This behavior has some caveats. Consider a rollout of a new version of a system that takes twice as long to handle each RPC call but does a significantly better job. CAS would flag the longer RPC handling time as anomalous behavior for each geographical location of the rollout, causing the release owner undue toil. The mitigation is to adjust the heuristics carefully in selecting relevant environments to include data beyond the perfect match.

User mistrust

CAS is useful but far from perfect. It has experienced incidents when users disregarded a canary failure and pushed a broken release. User mistrust of complex automation is at the root of many of these issues.

The CAS developers are tackling this mistrust by explicitly explaining, in human-friendly terms that don't require knowledge of statistics, why CAS reaches a particular conclusion. This includes both textual explanation and graphical hints.

Relative comparisons only

Because the model server stores only the outcomes of statistical functions without knowing the input values, CAS doesn't know the typical values for a time series.

Not knowing the semantics of the data implies that the tests being run are purely relative comparisons, such as having a t-test with null hypothesis that the metric didn't increase by more than 5 percent. While relative comparisons are easy to reason about, they behave extremely poorly if the provided time series value is typically zero, or if a large relative change occurs in absolute numbers too small to be important to the service owner.

This is a significant limitation of the mechanism. While it has not had much practical impact in real-world operation, especially given existing trivial workarounds, it merits improvement. Numerous improvements can be made to this mechanism, some quite simple. In addition to the future work mentioned previously, candidates include standard deviation analysis and looking at past observed behavior of the metric.

Scale limitations on input values

As CAS uses only a hard-coded set of statistical functions and their parameters, the system is somewhat inflexible about analyzing inputs outside of the expected input scale. For example, if the range of 1 percent through 100 percent difference is covered, what about the systems and metrics where a difference of 200 percent is normal? What if even a 1 percent difference is unacceptable?

CAS developers did not anticipate this to be a significant limitation in practice, which thankfully proved true. Most metrics meriting canary analysis turn out to contain some noise; conversely, most of our A/B testing hopes to see little difference between the two populations, so large differences are unexpected and therefore noticed.

LESSONS LEARNED

Good health metrics are surprisingly rare

The best way to use CAS is to employ a few high-quality metrics that are clear indicators of system health: suitable metrics are stable when healthy, and they drastically change when unhealthy.

Often, the best canarying strategy is to choose metrics tied to SLOs (service-level objectives). CAS automatically integrates with an SLO tracking system to apply service-wide SLOs and some heuristics to scale them appropriately to the canary size.

Setting an SLO is a complex process connected to business needs, and SLOs often cover an entire service rather than individual components. Even if a canary of a single component misbehaves in the extreme, its impact on a service's overall SLO can be small. Therefore, key metrics need to be identified (or introduced) for each component.

It's tempting to feed a computer all the metrics exported by a service. While Google systems offer vast amounts of telemetry, much of it is useful only for debugging narrow problems. For example, many BigTable client library metrics are not a direct indication that a system is healthy. In practice, using weakly relevant metrics leads to poor results. Some teams at Google have performed analysis that justifies using a large number of metrics, but unless you perform similarly detailed data analysis, using only a few key metrics yields much better results.

Perfect is the enemy of good

Canarying is a very useful method of increasing production

safety, but it is not a panacea. It should not replace unit testing, integration testing, or monitoring.

Attempting a "perfectly accurate" canary setup can lead to a rigid configuration, which blocks releases that have acceptable changes in behavior. When a system inherently does not lend itself to a sophisticated canary, it's tempting to forego canarying altogether.

Attempts at hyper-accurate canary setups often fail because the rigid configuration causes too much toil during regular releases. While some systems don't canary easily, they're rarely *impossible* to canary, though the impact of a having a canary process for that system may be lower. In both cases, switching to a strategy of gradual onboarding of canarying, starting with low-hanging fruit, will help.

Impact analysis is very hard

Early on, the CAS team asked, "Is providing a centralized automatic canarying system worth it?" and struggled to find a answer. If CAS actually prevents an outage, how do you know the impact of the outage and, therefore, the impact of CAS?

The team attempted to perform a heuristic analysis of production changes, but the diverse rollout procedures made this exercise too inaccurate to be practical. They considered an A/B approach where failures of a subset of evaluations were ignored, passing them in order to measure impact. Given the many factors that influence the magnitude of an outage, however, this approach would not be expected to provide a clear signal. (Postmortem documents often include a section such as "where we got

lucky," highlighting that many elements contribute to the severity of the outage.)

Ultimately, the team settled upon what they call *near-miss analysis*: looking at large postmortems at Google and identifying outages that CAS *could* have prevented, but did not prevent. If CAS didn't prevent an outage because of missing features, those features were identified and typically implemented. For example, if CAS could have prevented a $10M postmortem if it had an additional feature, implementing that feature proves a $10M value of CAS. This problem space continues to evolve, as we attempt other kinds of analyses. Most recently, the team has performed analysis over a (more homogeneous) portion of the company to identify trends in outages and postmortems, and has found some coarse signal.

The reusability of CAS data is limited

CAS's immense amount of information about system behaviors could potentially be put to other uses. Such extensions may be tempting at face value, but are also dangerous because of the way CAS operates (and needs to operate, at the product level).

For example, the CAS team could observe where canaries behave best and recommend that a user select only that geographical location. While the recommended location may be optimal *now*, if a user followed the advice to canary only in that location, the team's ability to provide further advice would lessen. CAS data is limited to its observations, so behavior at a local optimum might be quite different from the global optimum.

CONCLUSION

Automated canarying has repeatedly proven to improve development velocity and production safety. CAS helps prevent outages with major monetary impact caused by binary changes, configuration changes, and data pushes.

It is unreasonable to expect engineers working on product development or reliability to have statistical knowledge; removing this hurdle—even at the expense of potentially lower analysis accuracy—led to widespread CAS adoption. CAS has proven useful even for basic cases that don't need configuration, and has significantly improved Google's rollout reliability. Impact analysis shows that CAS has likely prevented hundreds of postmortem-worthy outages, and the rate of postmortems among groups that do not use CAS is noticeably higher.

CAS is evolving as its developers work to expand their scope and improve analysis quality.

Related articles

Fail at Scale
Reliability in the face of rapid change
Ben Maurer, Facebook
http://queue.acm.org/detail.cfm?id=2839461

The Verification of a Distributed System
A practitioner's guide to increasing confidence in system correctness
Caitie McCaffrey
https://queue.acm.org/detail.cfm?id=2889274

Browser Security:
Lessons from Google Chrome
Google Chrome developers focused on three key problems to shield the browser from attacks.
Charles Reis, Adam Barth, Carlos Pizano
http://queue.acm.org/detail.cfm?id=1556050

Acknowledgments

A great many people contributed key components of this work. Thanks to Alexander Malmberg, Alex Rodriguez, Brian O'Leary, Chong Su, Cody Smith, Eduardo Blanco, Eric Waters, Jarrod Todd, Konstantin

Stepanyuk, Mike Ulrich, Nina Gonova, Sabrina Farmer, Sergey Kondratyev, and many others who contributed in their own ways.

Also, thanks to Betsy Beyer, Brian O'Leary, and Chris Jones for technical review.

References

1. Banning, J. 2016. Monarch, Google's planet-scale monitoring infrastructure. Monitorama PDX 2016; https://vimeo.com/173607638.
2. Van Winkel, J. C. 2017. The production environment at Google, from the viewpoint of an SRE. https://landing.google.com/sre/book/chapters/production-environment.html.

Štěpán Davidovič *is a Site Reliability Engineer at Google. He currently works on internal infrastructure for automatic monitoring. In previous Google SRE roles, he developed Canary Analysis Service and has worked on both a wide range of shared infrastructure projects and AdSense reliability. He obtained his bachelor's degree from Czech Technical University, Prague, in 2010.*

Betsy Beyer *is a Technical Writer for Google Site Reliability Engineering in NYC, and the editor of* Site Reliability Engineering: How Google Runs Production Systems. *She has previously written documentation for Google datacenters and hardware operations teams. Before moving to New York, Betsy was a lecturer on technical writing at Stanford University. She holds degrees from Stanford and Tulane.*

Copyright © 2018 held by owner/author.