Google Site Reliability Engineering

Reliable Data Processing with Minimal Toil

Oct 12,2021
by Pieter Coucke (Google) and Rita Sodt (Google) with Julia Lee (Slack), Rich Feit (Google), Athena Vawda (Google), Betsy Beyer (Google), and John Lunney (Google)

How to minimize the manual work involved in updating a batch job safely by implementing **automated rollouts, validations, and canarying**, thereby limiting potential damage caused by new versions.

Background

As the Google Workspace[1] user base and product portfolio grew rapidly over the past several years, so did our need to process data reliably with minimal toil. To provide for user trust and safe systems, sometimes that meant switching to real-time/streaming data architecture when it made sense, and sometimes that meant increasing the safety of existing batch jobs— the focus of this paper. We wanted to support growth while minimizing manual work in order to keep data processing reliable.

Batch jobs typically do one thing well and run reliably for several years. As a consequence, **people tend to forget about batch jobs,** and they become unsafe "haunted graveyards"[2]: places where you don't really care to venture, where ugly surprises are likely to bite you in the back.

A couple more factors made teams even more hesitant to visit the haunted graveyard of batch jobs. Making changes to these batch jobs was a largely **manual and time-consuming process**. And whereas user-facing services[3] are built on an internal framework that provides monitoring, alerting, rollouts, and so forth, the teams responsible for batch jobs were left to set up most of this infrastructure themselves. Engineers had to carefully roll out a new version, monitor logs and metrics (such as the percentage of records changed by a run) during the rollout, and lacked basic canarying[4] capabilities.

A delayed batch job rarely causes immediate user-visible impact. At Google, batch jobs are **often not SRE-supported**, so an alert triggered by a delayed batch job typically doesn't page an on-call SRE, and is investigated by the development team as a non-urgent issue. However, a bug can be problematic and require manual intervention or even data restores. A job that removes data from an inactive account after 60 days has the potential to remove data for every user if the selection criteria has a wrong value. A compliance-related job that doesn't run on time can have legal implications. So batch jobs can break your services in interesting ways, even in the absence of a rollout.

When we realized that our batch jobs were problematic, we temporarily declared an emergency stop on rolling out changes to batch jobs. This stop led to yet another outage: dependencies (for example, a dependency on a deprecated API version) tended to break after some period of time. It was especially hard to detect what went wrong in these situations, since nothing was actually deployed.

To add to an already complicated scenario, **logging and monitoring** was hugely different between teams, adding to the time needed to root-cause an incident or even to add basic alerting.

---

[1] Google Workspace includes Gmail, Drive, Meet, Calendar, Docs, and more.
[2] Term coined by John Reese in No Haunted Graveyards.
[3] Like a frontend server or API backend.
[4] We'll discuss canarying further in Limiting Impact of Issues with Canarying.

Clearly, something needed to change. We implemented a two-part approach:

- A **continuous integration** system with automated testing, which is standard procedure for all our user-facing services.
- **Validation tests**, which are easier to reproduce than someone performing one-off tests during an initial fire-and-forget launch. Additionally, they document what we consider correct behavior.

This article describes an approach for reliable data processing that produces **correct** and **fresh** results while still allowing continuous integration. This approach increases developer velocity, as developers need to spend less time on manual verification.

We first focus on the specific data pipeline use case of batch jobs. We'll also address the broader case of asynchronous, event-based data processing pipelines.

What is Reliability for Batch Jobs?

What is a Batch Job?

Imagine that you're writing a program that removes files from Google Drive's trash after 30 days. One way to do this is with a batch job: a process that starts daily and scans over all users to remove files in their trash. The files to be removed can pile up during the day, and you can schedule the job to run at night. The job queries the database and iterates through the files to delete, performing actions like removing the file from disk, purging permissions, and deleting the database row.

We define a batch job as **a job that performs some finite amount of work** (typically by scanning over data or storage), **then terminates**. A batch job can be scheduled to run periodically (daily or weekly) but can also be a one-off— for example, to fix data corruption. Or a batch job can be triggered by some process that requires results from the batch job.

Benefits

Batch jobs are a **compute- and often cost-efficient** way to process **large amounts of work** when near real-time data is not needed. They often run during **off-peak hours** on **unused resources**, and are therefore essentially free[5] given a large enough deployment of compute power. They can run on resources provisioned specifically for the batch job or on cheaper preemptible instances. As a result, batch jobs are well-suited for analysis and for performing simple transformations of data like pre-processing a machine learning model. Other use cases include report generation, indexing, correctness checks, and bulk imports.

Challenges

The aforementioned benefits come with some challenges that can become dangerous when not considered properly:

- **Large blast radius:** Because they can modify the entire dataset in one go.
- **Data corruption**: For example, if a batch job overwrites data with empty or broken files.
- **Downstream delay**: The output of a batch job is often the input of another batch job. One batch job may collect a list of actions to perform, and then the subsequent batch job performs those actions. The output (for example, the modified storage) of a batch job may also be used by user-facing serving jobs. Corrupt or delayed data from a single job can rapidly propagate through a system, which makes repairs difficult and time-intensive.
- **Staleness of results**: When the job takes too long.
- **Overloaded downstream services**: If the job makes too many requests in a short period of time.

[5] See details about Borg best-effort batch tier job priorities in Borg: the Next Generation.

- **High cost** when operating on an entire corpus, as opposed to operating on a targeted set such as only recently updated items.
- **Duplicated and divergent logic**: For batch and live servers. Or, there might be version skew between the batch process and the server (for example, the server expects v2, while the batch job still writes v1).

This article focuses on the first four challenges, as the other challenges are solvable by implementing usage quotas and enforcement, or via software architecture decisions.

Reliable and Safe Data Processing

To apply an SRE approach, we can declare data processing to be reliable if well-reasoned SLOs are met. The **freshness SLO** ("Did the job complete in time?") is fairly straightforward to measure: it is a measure of how long we can wait for the data to be available. For example, we might want to make Google Analytics data available to the webmaster within an hour. This article will first focus on the **correctness SLO**[6] ("Did the job produce the correct results?"). Even when correctness is covered by high-quality tests, data corruption can creep in. In some systems, even the smallest amount of corruption may be unacceptable.

The Road to More Reliability

Our approach to the first three challenges above— to limit the blast radius, avoid data corruption, and limit propagation of errors— builds on reliability best practices like canaries and release qualification that are well understood for servers but are under-utilized for batch processing. As a case study from Slack illustrates, many reliability best practices for data pipelines are relevant for both batch jobs and asynchronous event-driven processing. We address the fourth challenge above— staleness of results— later in this article, in Ensuring Data is Available on Time.

We first organized batch jobs according to how risky they were, and then defined stricter procedures for more dangerous jobs. By creating a release pipeline with build stages and build promotions gated by automated validations, we could detect problems before they reached production. We took this a step further by canarying changes in production, first on a subset of the data, and later on a target population of our least risky users. By standardizing on a common platform, multiple teams could benefit from these features with minimal integration work.

We increased reliability for our data pipelines with the following approach:

1. We asked all teams to install processes that would allow **best-effort identification of risky changes** up front.
2. We didn't allow any new batch jobs that touched the whole dataset in one go.
3. We scoped our effort to **prioritize work on the riskiest** user-impacting jobs.
4. We **scaled up** by creating tools to make conformance easier.

Teams already know reliability best practices from servers well, but these best practices are under-utilized for batch processing.

[6] Jobs that run reliably according to a freshness SLO aren't necessarily safe— in other words, they might not meet the correctness SLO.

Safety Levels

We expanded on a policy that teams already knew for user-facing services: the enforcement of gradual rollout best practices when making production changes to code, configs, and databases. This policy was originally defined in relation to making changes to continuously running user-facing services. We built upon this existing well-known policy to describe which portion of the data is modified in one run of an updated batch job.

As shown in Table 1, we defined four safety levels, each describing how risky a change is based upon how much data is modified in one run of a new version. A higher (and thus safer) level indicates that a rollout has a smaller blast radius. The safer rollout is more gradual, with each step in the rollout modifying an increasingly larger part of the total data.

Table 1: *Four safety levels for changes*

Level	Impact of a change
Level 0	The **entire dataset** is affected in a single run of the job.
Level 1	Changes are **canaried** and don't affect the entire dataset. The canary can be manual or automated.
Level 2	Changes are gradually rolled out, first to **less risky populations** (such as internal, beta, or freemium user populations), then globally.
Level 3	Level 1 and 2 criteria are met and **no humans are involved** in the phased rollout.

Making Changes

Each safety level has a policy attached that describes which manual verifications are required in order to proceed with a change. A team with a batch job at level 3 has fully automated gradual rollouts and requires no manual verification. A batch job that falls in a lower (and therefore riskier) safety level requires more manual verifications for each change. This system incentivizes teams to modify their jobs to comply with the highest safety level, since doing so reduces their toil and increases release velocity by removing obstacles.

The lower the safety level, the more manual verifications we ask a team to perform for a change.

It became clear that manually upgrading each service to Safety Level 3 would require a lot of repeated work to configure rollouts, canarying, validations, and monitoring— many teams would have to perform the exact same tasks. And if we wanted to introduce a new safety level or add extra conditions to an existing level in the future, we'd have to ask every single team to implement the change.

While we could have written our own framework inside Google Workspace, we partnered with the recently initiated central Batch Platform Team to join their alpha program. The improvements made to that software are now available Google-wide— a nice example of how looking outside the walls of your team can make everybody's life better.

Benefits of Standardization

by Athena Vawda, Google Batch Platform Lead

Managing batch jobs in production is a complex endeavor. Google's internal technology stack provides the building blocks for production management, but integrating them can be a daunting task, and individual teams often end up duplicating work that has already been done by others.

Our solution to the duplication problem is the **Batch Platform**: a standard system for production management that integrates these building blocks into a unified platform.

A developer or SRE provides the platform with a minimal description of how their batch job should be run in production. From there, the platform configures all of the infrastructure needed to accomplish that goal, following the principle of convention over configuration so that the user can benefit from sensible defaults rather than specifying every last detail of their setup.

A standardized platform provides many benefits to a large software organization:

Scaling expertise

In a platform-less world, each team builds its own production setup, so a team without specialist knowledge of production ends up more susceptible to outages. With a platform, subject matter experts for each facet of production can codify their expertise into the platform's defaults and policies, so every team in the organization benefits from it.

This ability to scale out expertise has been essential to the deployment of Google Workspace's change management policy and others like it. The platform has provided a focal point for infrastructure experts to collaborate on building a Google-wide system for progressive rollouts, saving individual product teams from having to develop their own solutions to this problem.

Increased productivity

As the developer of a user-facing system, dealing with production can feel like an impediment, getting in the way of your actual goals. By providing a facade over the complexities of production, the platform allows you to focus on what you care about, rather than having to worry about the minutiae of how to configure various pieces of infrastructure.

As an SRE managing a collection of jobs, using a platform frees you from the repetitive work of configuring the same piece of infrastructure for N different jobs, allowing you to focus more on system-level reliability improvements.

Easier incident response

Without a platform, every team's jobs tend to be set up differently in production, leading to high cognitive load when debugging a problem that spans multiple teams. By providing a standard way to configure production, the platform ensures that different teams' jobs are set up uniformly. This lets incident responders focus on the important details of an incident instead of getting bogged down in accidental complexity, thereby reducing time to recovery.

Structured data

> By generating the details of a job's production configuration from a minimal description, the platform can also make this data available to other software in a structured way. This allows us to provide a rich UI with information about each job, its dependencies, and the infrastructure it uses. Structured data also provides a basis for other teams to build tools on top of the platform, which users can then use without needing to write additional configuration.

These benefits are already well known inside Google for user-facing services. The Batch Platform is much newer, but we're already making life easier for over 450 teams across the company, and we have our sights set on turning production management for batch jobs into a solved problem.

Cloud Solutions that Can Help with Standardization

If you're considering implementing a similar approach at your organization, you might consider the following:

- For data manipulation, Cloud solutions like BigQuery and Dataflow can save you the trouble of provisioning servers.
- BigQuery can query data that is ingested in batch or continuous modes.
- You can use Dataflow in real-time, continuous, or batch mode.
- Cloud Scheduler helps with the surprisingly complex task of reliably launching batch jobs at scale.

Setting up the Release Pipeline

Similar to how we handle user-facing services, we don't deploy changes to batch processes immediately to production. Instead, we first go through stages ("environments") in a release pipeline that check if the new version compiles, passes tests, and behaves as expected.

We typically define **three release stages**, detailed in Table 2: *Autopush, Staging, and Production*. The release is promoted to the next stage on a fixed schedule when release certifications at that stage pass.

We also classify data into different **datasets**. Our standard *Prod* dataset contains live data. The *Test* dataset is a copy, subset, or manually curated[7] dataset that's typically shared by all release stages before reaching the *Production* release stage. Nothing prevents you from creating a completely separate dataset for each release stage as you see fit. Some systems do not have an available test data set. While we strongly encourage the use of test data, it is not a requirement. Without test data to run the batch job on, it's all the more important to launch with a smaller production canary set to limit the blast radius.

A **dry run** means the job skips the writing phase and does not produce any changes that affect other parts of the system. This ensures that an error stays limited to a particular binary. You can configure a dry run with a parameter in the script that starts the job. If writing is an essential part of the job, you can configure the job to write to a temp location not consumed by any other process (a different storage bucket or folder). You can enforce this setup with permissions on the dataset to ensure nothing can be modified accidentally.

The **code deployment schedule** is completely different from the job run schedule. For example, it is common in production to have a daily run but only a weekly release.

[7] For example, a manually created dataset containing all exception cases discovered through previous bugs.

Table 2 shows more details about each release stage. A version moves ("promotes") to the next stage when it is considered stable.

Table 2: *Release stages*

Release stage	Dry run?	Purpose
Autopush	Yes	Every two hours checks **"Does this compile, pass tests, and run?"** by building and deploying a fresh release based on the latest checked-in revision that passes continuous integration. Runs on a *Test* dataset.
Staging	Yes	Ensures that the actual **mutations between the versions are expected and equivalent** before writing data. Launches two instances of the job at the new and old version for an A/B test (see A/B testing, below) of data output or counter comparisons. These dry run jobs typically read from the *Test* dataset, but sometimes read from *Prod* datasets, too. Reading from the *Prod* dataset is allowed as long as these *Staging* jobs cannot modify the Prod environment. Batch jobs accessing *Prod* datasets in the Staging release stage are subject to the same security and data compliance rules as production to ensure that the (derived) data is not accessed or stored outside the production system.
	No	Output may be consumed by other processes that also operate on the *Test* dataset. Output **continuously runs and is actively monitored** for delays or issues in other downstream jobs. Typically, new versions remain in this stage for days or a week to allow time for issues to arise before rolling out further with the production release. Non-prod jobs shouldn't touch production, so these read-write staging jobs should never use the *Prod* dataset.
Production	No	Where a release progresses after it successfully completes all previous stages. This stage supports multiple arbitrary canary stages, explained later in Limiting Impact of Issues with Canarying.

Removing Manual Checks with Automated Validations

In the previous section, we set up our release pipeline with stages[8] running next to each other. But when do we consider a new version ready to progress from one stage to the next?

An important part of gradual releases is the go/no-go decision made at each stage before promoting the new version to the next stage. For user-facing services, we typically rely on signals from the requesting side (for example, a change in client errors or latency). For batch jobs, we use signals on the generated output.

A promotion decision comes down to **determining whether the version is considered stable enough to move ("promote") to the next stage**. For less critical batch jobs, you might consider a job stable after one successful run. For extra safety, you can require two or more jobs to run successfully (or you might even require a full week of jobs, to cover a weekend peak) before moving to the next stage.

A promotion decision comes down to determining whether the version is considered stable enough to move to the next stage.

We can automate these stability checks with automated validations. You should aim to minimize **false positives** (meaning that a job is incorrectly labeled as unstable), as they require manual intervention to investigate and therefore introduce rollout delays. The approaches below measure stability and minimize the number of false positives.

The two-phase mutation design pattern is particularly suited to implement some of these validations. This approach entails storing candidates (like IDs) somewhere, and then performing API calls with these IDs in a separate process. This split allows validations, dry-runs, and A/B testing on the list of IDs without making actual changes. A note of caution: if there is ever a chance that these unique identifiers have private user information encoded in them, ensure that your batch pipeline protects this information correctly, even between different phases of the mutation.

In addition to being useful when deciding if a version is stable enough to promote to the next version, you can use this pattern on every job that runs when the version is already fully deployed on production. This pattern allows you to continuously monitor the health of your jobs with alerting when violations occur.

Start-up Tests

Start-up tests validate basic conditions at the beginning of the batch job before reading or writing any data. Check if the data you need is available and in the correct format, and that config files are not empty and don't contain overly broad values (for example, * or / for file paths).

The two-phase mutation design pattern can also be seen as a start-up test. The first phase (start-up) reads some data, decides what it's going to do, and validates the actions. Only then does the second phase of mutations occur.

Process Exit Code

The process exit code is the very basic sign something went wrong. A 0 (zero) indicates success, while any other code indicates an error. As a first step, you can check the exit code after the process runs, and deploy this version in the next stage after it successfully runs at least once.

[8] Learn more about Release Engineering in the SRE Book.

Counter Validation

A next step is to compare counters that the process logged during runtime. A counter might be the number of records processed, number of files written, processing time, percentage of records processed vs. total records, or basically anything that can be used as a metric.

We've found evidence that looking for counter anomalies (for example, large increases in some particular counter, or a counter increasing from zero) could have prevented major issues. Basic validation like checking that at least one row has been processed or at least one file was written can help detect issues early. So start with these basic validations before spending too much time on finding the perfect metric and range. There are cases where unexpectedly high counter diffs can occur and break this validation setup— for example, a big spike for a World Cup game.

Comparing these metrics with previous runs in the same stage (or even from other stages) provides an indication of stability. The tricky part here is that variability (between stages, runs, a growing dataset, and even the time when a batch job runs) can cause substantial differences in these numbers. Maybe the reason for the difference is exactly the reason why you built a new version (for example, to fix a bug that missed a part of the dataset).

Exact number comparisons are sometimes too noisy. A different approach is to use ranges (for example, the value should fall between X and Y) or percentages (for example, records processed must be between 10% and 20% of the total dataset).

Exact counters do make sense if the job retains correctness when it performs the same operation several times over the same dataset. This approach would work when creating a usage billing report over a fixed time period, but would be difficult to use when deleting files.

Data Validation

To add even more checks for reliability, you can add data validation. You can implement data validation within the batch job before writing data, or as a separate process after a successful run of the batch job. Validation can take many forms— ideally, it is **application-specific consistency checks**, but you might also check the format of a file, for a non-empty file size, or for duplicate rows. When running the validation as a separate job, it's a good idea to limit the validation job's permissions to the strict (read-only) minimum, according to the principle of least privilege.

A/B testing

With A/B testing on jobs processing the exact same input, we can tackle false positives in counter and data validation. Run the new version in dry-run mode (so that you don't write to any storage) **on the exact same input** as the previous version (ideally also running in dry-run mode) and compare the counters. Or, even better, compare the data itself. One challenge of data comparison is ensuring proper handling of known data that is expected to change from run to run, such as timestamps, build labels, or machine names. If an intentional change (for example, a bug fix) caused a counter difference, you'll need to perform a manual approval so the job promotion can proceed.

Resource Overloading

A new version that generates **more load on dependencies than expected** is a good signal to detect issues. If you're monitoring your API calls, you can query[9] those calls and apply quota limits to avoid overloading downstream systems. If you don't have that monitoring in place, annotate or wrap API calls to increment a counter, then use the number of downstream calls as a metric for detecting unexpected change.

Soak Time

When running jobs on a fixed schedule, you need to make sure there's sufficient time for the worst-case runtime of the job, plus some additional verification time to detect any potential breaks down the line. A wide variability in run time (for example, due to throttling on an underlying component or because of a seasonal peak event) might mean that you set the job to run only once a week in order to make sure the job can finish in that time period. Setting a job to run every week, plus requiring at least two successful runs to promote to the next stage, equates to six weeks in a four stage setup to get the new version in production, causing a slowdown in rolling out new features.

A workaround to increase rollout velocity is to make job runs and promotions cascading. This means the start time becomes dynamic instead of fixed in a crontab. After a successful run and some verification time, the next run starts. This strategy can significantly lower the total time to get a release into production. Another alternative is to set the job run frequency to every hour, but check at startup if another job runs, and if that job completed more than X hours ago. Cascading rollouts make troubleshooting and monitoring more complicated because failures or delays in a run require checking the previous run or even the run before that. A job orchestration system can help here.

You can tweak this approach: for example, by requiring the first stage to run successfully five times, but only three times in the next stage, and then only two times for each following stage.

Limiting Impact of Issues with Canarying

So far, we described how to detect issues before deploying a new version to production. Here we go one step further by limiting the rollout to a random subset of the data using canarying. Then we'll improve upon this concept by canarying on a specific subset (a "target population") of the data that is most risk tolerant.

The Canarying Releases chapter in the Google SRE Workbook discusses how to apply canarying to a user-facing service. This approach typically focuses on the behavior of a new binary when handling a portion of traffic. For example, if the canary detects a faulty version, as indicated by an increase in errors or latency, the canary version rolls back and will not reach a broader part of production. Typically, canaries progress based upon traffic segmentation, but this concept doesn't map to a batch process. Instead, canarying for batch processes needs to happen based on segmented populations— this is typically users or customers, but can more generally be any logical objects in the data model.

As shown in Figure 1, the production release can have arbitrarily many, sometimes overlapping, canary phases. Each column represents a run cycle that propagates throughout the entire dataset but is split into different subsets. A new version (v2) promotes through the various canary phases and is blocked when there is an issue in a canary phase.

[9] For example, with statsd.

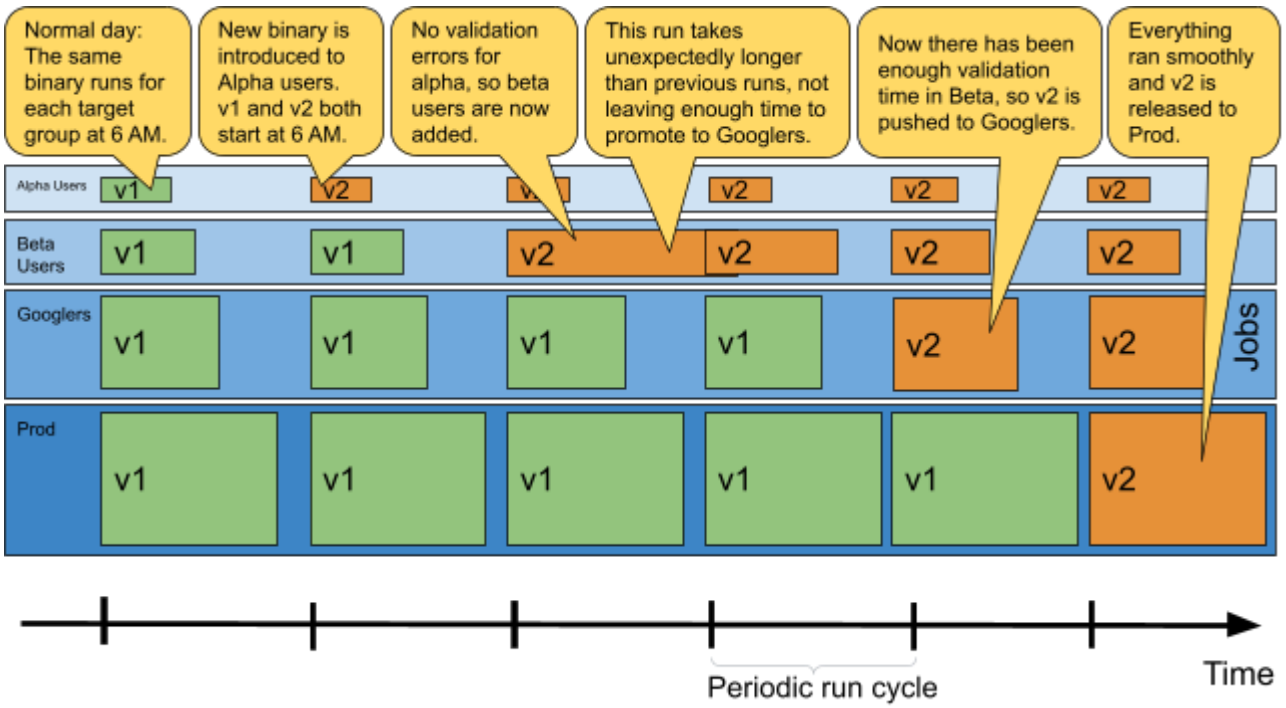

Figure 1: *Canary phases for a production release.*

Splitting the Work

Typically, to canary a user-facing service, you inspect the incoming request (for example, to identify the user) and route it through a load balancer to a certain pool of servers running the new version. After verifying that nothing breaks, that canary version is promoted to the production stage. The verification process has a **validation** component (for example, no increase in http 500 errors) and a **soak time** component (meaning that the canary has to run long enough to detect issues and should see typical peak-of-the-day load).

For batch jobs, we implemented this canarying approach by having **a set of jobs running the canary version (v2 in Figure 1), and another set of jobs running the production version (v1)**. Each job is configured with a **startup parameter**[10] that defines the subset on which it should run. It is up to the binary to interpret this parameter and implement it correctly. For the canary, this parameter might be defined as "10% prod" and implemented as a filter to process only records where `hash(userid) mod 10 == 0` to run on 10% of production users. The production version can be implemented as `hash(userid) mod 10 != 0` if the two runs should not overlap, or can simply ignore the parameter altogether if the jobs are idempotent. Having idempotent jobs is especially useful when you want to compare the outcome of the canary run with the production run.

Alternatively, you could split the work using a storage partition— for example, if you have different cloud storage buckets. Or you might partition the data according to country or region.

Extending to a Target Population

A target population is a subset of your data— for example, paying customers, or users who opted in for early access. We can extend the canary setup to target populations, allowing us to run the new version on increasingly larger groups of users.

[10] Like adding an argument *-subset=alpha-users* to the crontab entry.

Instead of having canary and production stages, we might have a *canary-for-developers-team*, followed by a *canary-for-employees*, then a *canary-production-1%-free-users*, and so on. Each of these stages represents a segment of users. Each stage is implemented the same way as above: through a single configuration flag in the startup script that determines which dataset to process.

Targeting Library

We created a library for typical user and customer selection criteria that can be reused by all batch jobs. This not only saves developers time, but ensures that the selection is the same everywhere. New criteria can be added to the library and made available to everyone. This is helpful for use cases such as when we don't want to canary stages to involve large enterprise customers.

The targeting library helps ensure the canary processes a representative production subset. To extend our large enterprise customer example, ideally you could include some large enterprise customers in the canary through opt-in.

Ensuring Data is Available on Time

So far, we've primarily discussed correctness. The freshness SLO is equally important and gives an indication of whether data produced by a batch job is fresh or stale.

The freshness SLO of a batch measures time since the last successful completion of the job. This measurement provides a signal of how much time has passed since there was a 'fresh' output of the batch job. Teams can set the threshold of the SLO according to their business needs (for example, a data wipeout job must complete successfully every X days to be compliant).

The thresholds you set for freshness depend on the use case. It is important that a cloud usage dashboard shows recent data in order to avoid billing surprises for a customer. The data pipeline underpinning this dashboard needs to meet strict freshness SLOs. On the other hand, you can probably delay removing files from the trash until after the previous job has completed.

For SREs, especially when operating at a large scale, it may feel like rare events tend to pop up at the worst possible moment. Hence, the motto *Hope is not a strategy*. For that reason, an oncall rotation ensures that someone is always ready to handle an outage and **mitigate the issue before the freshness SLO is violated**.

Jobs can take longer to run than the schedule— for example, a daily run might actually take two days. The implications here depend on the criticality of the job, and it's important to implement **alerting** to catch these scenarios. To mitigate the problem, you might add more resources to make the job complete sooner, realign requirements, or split the job into smaller tasks.

It is tempting to choose midnight as the starting time for a batch job. Doing so can lead to a resource crunch if **many jobs launch at the same time**. If you define a time period instead of specifying an exact start time, the job scheduler can optimize when to launch individual jobs.

Fresher Results with Event-based Processing

Event-based processing[11] is a setup in which a system emits an event (for example, *file-deleted*) and other systems subscribe to the event[12], which triggers those systems to perform their work. The need for fresher results and less coupling, combined with a trend towards microservices and serverless, has accelerated the adoption of event-based processing across the industry. Many[13] batch jobs can convert to this architecture.

Event-based systems have advantages when it comes to freshness, but correctness tends to be somewhat trickier, for several reasons:

- Non-critical work or **ordering of work** can block critical work.
- You need an **explicit setup for dry-runs.**
- You lose the benefit of **precomputing data** that can be used for multiple data items.
- Having processing power available all the time **can be more costly** than running off-peak on idle available capacity.
- **Corruption detection** is spread over several systems.
- Requires **a global shutdown emergency button** to mitigate issues quickly.
- A queue introduces **an external dependency** that affects your SLO. (When targeting five 9s of availability, every dependency counts.)

There are benefits and downsides of event-based processing to consider, which we won't cover in detail here, as this topic merits its own article. See the Data Processing Pipelines chapter in the SRE Workbook for a robust comparison. On the positive side, event-based processing tends to produce fresher results, is less likely to cause outages by overwhelming servers, and is often low-risk in terms of blast radius. When it comes to specific tactics, adding tasks to an asynchronous Pub/Sub queue will let you scale workers as the amount of messages on the queue increases. Using Cloud Functions, you can process queue messages without provisioning servers.

Below is a case study from Slack about how they make data processing more reliable with event queues that feature a series of deploy stages. Criticality-based routing provides further isolation between data processing jobs.

[11] Also known as async processing or incremental processing.
[12] A bit like registering for a breaking news email newsletter.
[13] Some example exceptions include jobs that need to look at the whole corpus, such as validation jobs, machine learning, and one-off repairs.

Reliable Async Compute at slack

by Julia Lee, Asynchronous Services Engineering Lead

The Asynchronous Services Engineering Team at Slack provides async computing services to engineering teams across the org. Async compute accounts for over 60% of compute for Slack application code—from fetching a preview of a URL to importing and exporting large amounts of data to and from files— so high reliability is essential.

As illustrated in Figure 2, an event-driven compute service (composed of multiple sub-services) continuously processes queued asynchronous jobs by routing them to shared pools of executor hosts, which execute the jobs themselves. Though Slack's async service is not strictly for batch processing, we can use similar architectural features to improve reliability. Specifically, we use change management and workload bucketing to reduce the blast radius of a single change to an enqueued workload.

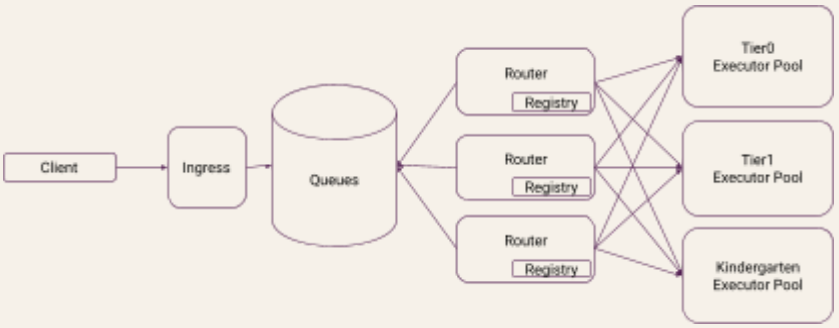

Figure 2: *Life of a job from enqueue to execution*
Enqueued jobs are fetched by routers and routed to the appropriate executor pool based on configuration in the Job Registry.

Executor **deploy stages provide an initial mechanism for incremental rollout** of job implementation changes, giving internal job owners an opportunity to identify issues before they impact all external users. Prior to deploying changes site-wide, Slack's application code is deployed to dogfood, which runs Slack's internal Slack workspaces. All jobs enqueued in a dogfood Slack workspace are completed by a dogfood-specific pool of executors. In addition to error rates, highly visible functionality like URL unfurls, which are performed by async jobs, are used as an indicator of system health during deploys.

Jobs are bucketed and routed based on criticality (kindergarten, tier1, tier0), each of which has its own isolated processing pipeline and pool of executors. When a new job type is onboarded, it starts in the kindergarten bucket. It is then promoted to tier1 and then, if considered high criticality, promoted to tier0. The kindergarten infrastructure is completely isolated from the rest of the tier1 and tier0 infrastructure and provides a proving ground to execute new workloads and understand resource requirements before sharing resources with many other production workloads. When a change is made to an existing workload, the job can be moved back to the kindergarten tier and re-promoted for safer rollout. Jobs are also assigned a delay tolerance (*immediate, soon, besteffort*), so if the queue of jobs of a given criticality tier grows too large, we can prioritize jobs sensitive to execution delays over jobs with a longer delay tolerance.

Change isolation improves reliability, but does not guarantee it; when incidents do arise, we need **tooling** to quickly restore system health. The Job Registry stores configuration for each job type and is an interface to operational tooling. Specifically, the Job Registry supports:

- Configuring job criticality and delay tolerance
- Quickly re-routing or pausing jobs by type in case of an incident
- Rate-limiting jobs by type and argument values
- Error-routing to the job owner

When a job is received by a router, the router performs a Job Registry lookup to determine which executor pool to route the job to. Committed changes to the Job Registry JSON are validated by a series of tests and can quickly be deployed, thus providing a mechanism to quickly pause or re-route and isolate a problematic job type. Paused jobs can be re-enqueued for execution when the issue is resolved.

To establish standardized health criteria and strong user ownership, the Job Registry provides support for rate limits and alerts configured by job type. It also provides standardized health metrics such as error rate. Metadata about the owning team stored in the Job Registry enables error-routing directly to job owners, who are best equipped to respond to job-specific issues.

Building isolation into the architectural design of an asynchronous computing service provides a safer environment to roll out changes to job implementations. However, until we figure out how to implement jobs that don't fail or behave unpredictably, **incorporating job ownership and incident response tooling into system design is equally critical** to running maintainable infrastructure.

Conclusion

Data processing comes with a unique set of challenges and risks, which we can mitigate using the best practices we employ for servers. We urge all batch framework developers to build canarying and release qualification support to scale more reliable data processing without the manual overhead.

If you are building or maintaining a data processing batch job, we encourage you to think about setting correctness goals (for example, no more than X rollbacks or repairs due to data corruption) and freshness SLOs (for example, how much time can lapse before data is updated). Avoiding data corruption and ensuring data freshness are both important aspects of data processing that we're focusing on at scale in Google SRE.

This article builds on the work of many who contributed through tooling, documentation, comments, and presentations. A special thanks to Alex Cebrian (Slack), Glen Sanford (Slack), Matthew Drake (Google), Salim Virji (Google), Jennifer Petoff (Google), Todd Underwood (Google), Yuval Greenfield (Google), Joe Kearney (Google), and Jason Lee (Google).