Dataflow for Real-Time Gaming Analytics

https://github.com/GoogleCloudPlatform/dataflow-solution-guides/blob/main/use_cases/gaming_analytics_dataflow_guide.md

Reference Architecture

Overview

Real-time gaming analytics refers to collecting, processing, and analyzing player data as it is generated during gameplay. This provides immediate insights into player behavior, game performance, and overall player performance. Incorporating real-time feedback into a game could enrich & enhance both single-player & multi-player experience and improve your revenue per user when integrated with your existing application architecture. While this solution is geared towards one single industry (gaming) on the surface, gamification is now pervasive across many verticals. These principles can be applied to industries such as education, healthcare, and e-commerce, and should be viewed as opportunities to maximize your return on investment on the data you collect.

Business Impact

Combining real-time data with your gaming infrastructure can deliver business impact in the following ways:

- **Enhanced multi-player experiences:** Deliver richer experiences for multi-player games, including real-time scoreboards and in-game actions & offers based on current rankings. Adjust balance game difficulty based on real-time player data.
- **Personalized gameplay:** Present tailored experiences for gameplayers based on real-time gameplay data. Adapt game difficulty as player succeeds & fails at various stages of gameplay. Deliver user-specific notifications & reminders to nudge users to continue gameplay.
- **Targeted monetization:** Identify high-value players and tailor in-app purchase offers or advertisements to their specific interests and behaviors, increasing revenue.
- **Improved player retention:** Detect early signs of player churn and intervene with personalized offers, challenges, or communication to keep players engaged.
- **Curb bad actors:** Quickly find players who demonstrate abusive behaviors (e.g. exploiting glitches, verbal abuse, harmful conduct) and remove them from gameplay in real-time

This business impact can be realized across various industries:

- **Gaming:** Dynamically adjust game difficulty as player(s) proceed through sequential levels of gameplay. Integrate real-time leaderboards and just-in-time notifications & reminders to sustain player engagement. Deliver tailored offers & ads for high-level players.
- **Education:** Apply gaming principles to learning paths for virtual learners. Adapt questions & exercises in real-time to keep student engaged and skill up on a given subject.
- **Healthcare:** Celebrate milestones with tailored notifications & positve messages during a patient's treatment plan (e.g. present a congratulatory message for following a treatment plan for four consecutive weeks). Use-real time data to display timely notifications for when to take medications or take actions on treatment.
- **Retail & E-Commerce:** Create head-to-head experiences or leaderboard tracking to reward loyal customers for certain actions (i.e. reaching a spend threshold, sharing a promotion on social media, referring new customers).

Customer Stories

- Nintendo deployed Pub/Sub, Dataflow, and BigQuery to collect log data from Super Mario Run, which reached 40 millions downloads in four days without any major issues
- Niantic's data science teams worked with BigQuery, BigTable, Dataflow, and Pub/Sub for data analytics use cases to support the launch of Pokemon GO, which scales to close a million transactions per second in a matter of minutes
- King / Activision Blizzard migrated from on-premise systems to Dataflow to ingest data and support its massive data archive

Technical Benefits

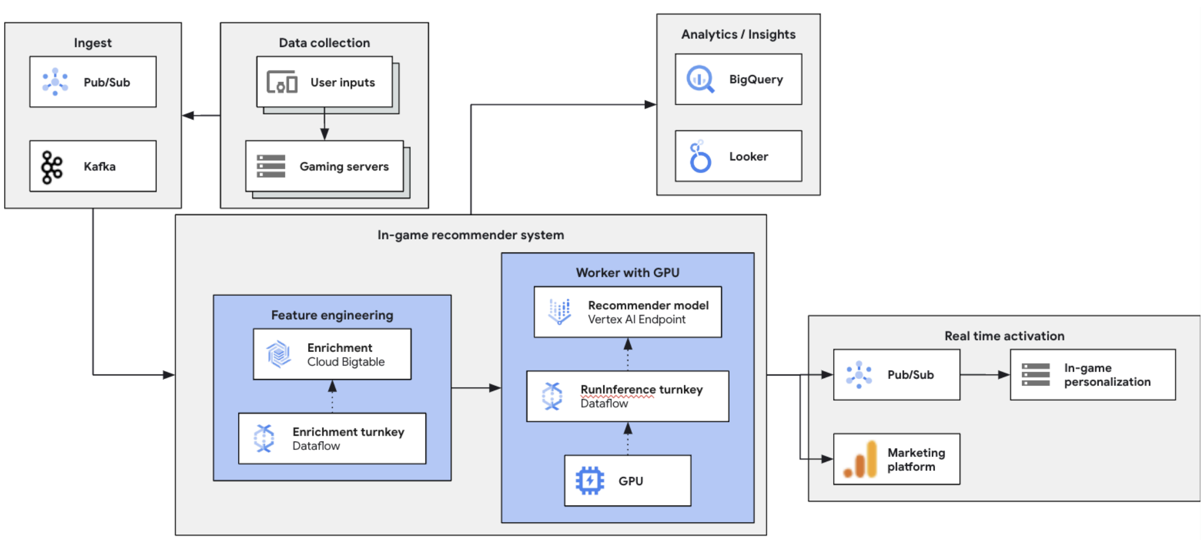
- **Streaming AI:** Enhance user experience with streaming predictions. Call for predictions from your Gemini model in Vertex AI using RunInference, and deliver personalized gameplaying experiences in real-time.
- **Horizontal autoscaling:** Gaming workloads can oscillate over the course of a day, week, or launch cycles. Instead of worrying about overprovisioning your infrastructure for a big release, or underprovisioning during times of anticipated low usage, Dataflow's horizontal autosclaing will ensure that you have enough resources for where ever your usage levels are.
- **Event time model:** Apache Beam provides users rich semantics for analyzing based on event time as opposed to processing time, which preserves analytical accuracy even if your users are disconnected during gameplay.

- **Deep I/O integration:** Dataflow's read performance from streaming data systems like Pub/Sub and Kafka can handle effortless scaling beyond 10 GB/s, capable of handling workloads of any scale.
- **Global availability:** Dataflow's availability across the world backed by Google's industry-leading network infrastructure ensure that your stream processing can respond gracefully to spikes in traffic, no matter where it might be occuring in the world.

Architecture

The architecture diagram shows how to apply real time AI/ML for an in-game model recommender. The architecture is shown here to write output to an output marketing platform. It also writes to BigQuery for analytics purposes. This could be also a separate pipeline reading from the Pub/Sub output topic, to decouple the analytics pipeline from the in-game activation pipeline.

However, that's not the most important part. The architecture enables continuous injection of recommendations back into the game, by using Pub/Sub as output. This is an essential property. Players engagement depends on the time-to-activation.



The data collection comes from the gaming platform servers and ingest data continuously in either Pub/Sub and Kafka. There is a single Dataflow pipeline, represented in the box "In-game recommender system". This pipeline leverages GPUs for real time predictions, although other options are also possible (e.g. model deployed in Vertex AI, and Dataflow using CPUs and calling the Vertex AI endpoint).

In terms of infrastructure, this architecture needs the input topic (in Pub/Sub or Kafka), a dataset in BigQuery for the analytics pipeline (optional), the Dataflow job (probably with GPUs) and the output Pub/Sub topic (or Kafka topic).

The architecture has the following components:

- Pub/Sub or Kafka topics. A topic per incoming data source.
- A Bigtable table, or a Vertex AI Feature Store, containing features data.
- A Dataflow job, possibly using GPUs, to enable fast inference in streaming.
- A destination BigQuery dataset, to write output for analytics purposes.
- A destination Pubsub topic, to buffer predictions to be sent back to the gaming servers.
- Optionally: Networking setup for the Dataflow job to reach the different services and publish back any activation. This may require using Cloud NAT, public IPs, or other alternatives.

Design considerations

This architecture could have one or two pipelines:

- In-game activation pipeline
- Analytics pipeline

The in-game activation pipeline takes data from Pub/Sub or Kafka, and applies different transformations. This includes feature engineering using the Enrichment transform. The features can be stored either in Bigtable (shown in diagram) or Vertex AI Feature Store. You can also apply many other transforms for typical ML related tasks (e.g. NLP). For the inference, the model can be loaded locally and applied using the RunInference transform. For the lowest prediction latency, we recommend using GPUs for that step.

Alternatively, the model can be stored externally to Dataflow, for instance deployed in a Vertex AI Endpoint. The RunInference transform allows calling the model if it is deployed in an endpoint. In this setup, the GPU is not necessary.

The output should be written to Pub/Sub or Kafka, for immediate activation back in the game. This ensures very low latency in-game recommendations.

The analytics pipeline could be part of this main pipeline, or could be deployed as a separate pipeline. The diagram shows everything as a single pipeline for simplicity, but we advise the deployment of separate pipelines to avoid performance interferences between both.

For the output to a marketing platform, Apache Beam does not have specific connectors. We recommend using the Web APIs I/O connector to publish the activation results in any marketing platform using its HTTP REST API:

- https://beam.apache.org/documentation/io/built-in/webapis/

In summary, you should take into consideration the following design decisions:

Step	Description
Real time input and output (Extract)	The architecture assumes Cloud Pub/Sub or Kafka for receiving data from gaming servers. Setting up the real time export from those systems into Pub/Sub is out of the scope of this design.
Enrichment (Transform)	We are assuming that the upstream data sources will only populate a "ids", and that any other feature required for the inference process is stored in Bigtable, Vertex AI Feature Store, or any other fast-lookup database. In the above architecture, we assume it is Bigtable, since it is supported by the turnkey "Enrichment" transform of Apache Beam. The same would work for Vertex AI Feature store. For any other database, the design may require the development of custom code for lookups. We recommend using the Web APIs I/O connector in those cases. Combining the Enrichment and Web APIs I/O may require using cross language pipelines with the Dataflow runner v2 (see instructions for Python and for Java).
Inference (Transform)	Once the data is hydrated and all the necessary features are extracted, we can apply ML/AI models to our data. The RunInference turnkey transform works both with external models deployed and with local models leveraging a GPU.

	The external models could be deployed in a Vertex AI endpoint, as a HuggingFace pipeline, as a GenAI API such as Gemini or Gemma, and many other options.
Output (Load)	Once the prediction is obtained, the corresponding action should be injected back into the output platforms. For this purpose, we recommended using the Web APIs I/O connector, if the platform does not have a specific connector (e.g. GoogleAds has a Beam connector). Alternatively, the output can also be sent to a Pub/Sub topic to buffer the predictions before they are sent back to the gaming platform.

Planning your pipelines

We will cover key considerations when planning your Dataflow pipelines in this section.

Service level objectives & indicators (SLOs and SLIs)

An important measure of performance is how well your pipeline meets your business requirements. Service level objectives (SLOs) provide tangible definitions of performance that you can compare against acceptable thresholds. For example, you might define the following example SLOs for your system:

- **Data freshness:** generate 90% of product recommendations from user website activity that occurred no later than 3 minutes ago.
- **Data correctness:** within a calendar month, less than 0.5% of customer invoices contain errors.
- **Data isolation/load balancing:** within a business day, process all high-priority payments within 10 minutes of lodgement, and complete standard-priority payments by the next business day.

You can use service level indicators (SLIs) to measure SLO compliance. SLIs are quantifiable metrics that indicate how well your system is meeting a given SLO. For example, you can measure the example data-freshness SLO by using the age of the most recently processed user activity as an SLI. If your pipeline generates recommendations from user activity events, and if your SLI reports a 4-minute delay between the event time and the time the event is processed, the recommendations don't consider a user's website activity from earlier than 4 minutes. If a pipeline that processes streaming data exceeds a system latency of 4 minutes, you know that the SLO is not met.

Sources & sinks

To process data, a data pipeline needs to be integrated with other systems. Those systems are referred to as sources and sinks. Data pipelines read data from sources and write data to sinks. In addition to sources and sinks, data pipelines might interact with external systems for data enrichment, filtering, or calling external business logic within a processing step.

For scalability, Dataflow runs the stages of your pipeline in parallel across multiple workers. Factors that are outside your pipeline code and the Dataflow service also impact the scalability of your pipeline. These factors might include the following:

- **Scalability of external systems:** external systems that your pipeline interacts with can constrain performance and can form the upper bound of scalability. For example, an Apache Kafka topic configured with an insufficient number of partitions for the read throughput that you need can affect your pipeline's performance. To help ensure that the pipeline and its components meet your performance targets, refer to the best practices documentation for the external systems that you're using. You can also simplify infrastructure capacity planning by using Google Cloud services that provide built-in scalability. For more information, see Using Google Cloud managed sources and sinks on this page.
- **Choice of data formats**: certain data formats might be faster to read than others. For example, using data formats that support parallelizable reads, such as Avro, is usually faster than using CSV files that have embedded newlines in fields, and is faster than using compressed files.
- **Data location and network topology**: the geographic proximity and networking characteristics of data sources and sinks in relation to the data pipeline might impact performance.

Regional considerations

Dataflow is offered as a managed service in multiple Google Cloud regions When choosing a region to use to run your jobs, consider the following factors:

- The location of data sources and sinks
- Preferences or restrictions on data processing locations

- Dataflow features that are offered only in specific regions
- The region that's used to manage execution of a given job
- The zone that's used for the job's workers

For a given job, the region setting that you use for the job and for the workers can differ. For more information, including when to specify regions and zones, see the Dataflow regions documentation.
By specifying regions to run your Dataflow jobs, you can plan around regional considerations for high availability and disaster recovery. For more information, see High availability and geographic redundancy.

Security

As a fully managed service, Dataflow automatically encrypts data that moves through your data pipeline using Google-managed encryption keys for both in-flight data and at-rest data. Instead of using Google-managed encryption keys, you might prefer to manage your own encryption keys. For that case, Dataflow supports customer-managed encryption keys (CMEK) using the Cloud Key Management Service (KMS). You can also use Cloud HSM, a cloud-hosted hardware security module (HSM) service that allows you to host encryption keys and perform cryptographic operations in a cluster of FIPS 140-2 Level 3 certified HSMs.

When you use CMEK, Dataflow uses your Cloud KMS key to encrypt the data, except for data-key-based operations such as windowing, grouping, and joining. If data keys contain sensitive data, such as personally identifiable information (PII), you must hash or otherwise transform the keys before they enter the Dataflow pipeline.

Networking

Your networking and security requirements might mandate that VM-based workloads such as Dataflow jobs use only private IP addresses. Dataflow lets you specify that workers use private IP addresses for all network communication. If public IPs are disabled, you must enable Private Google Access on the subnetwork so that Dataflow workers can reach Google APIs and services.

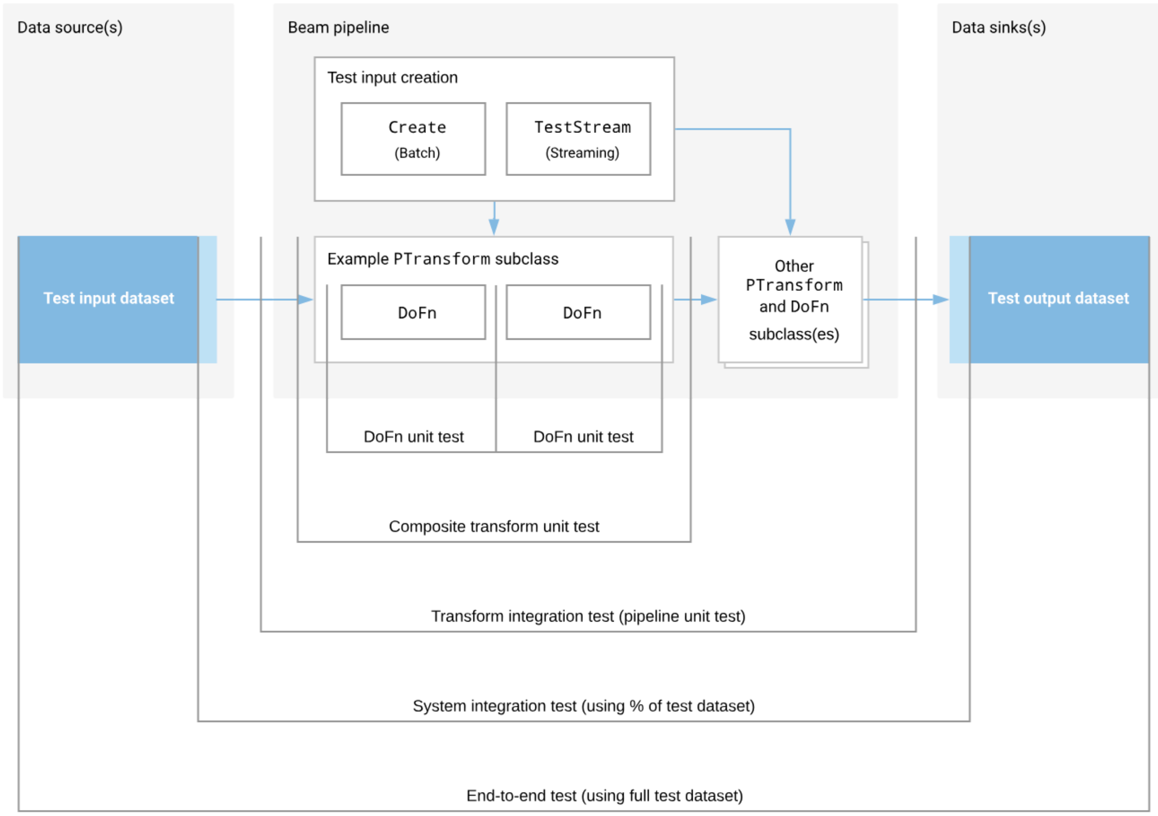
We recommend that you disable public IPs for Dataflow workers, unless your Dataflow jobs require public IPs to access network resources outside of Google Cloud. Disabling public IPs prevents Dataflow workers from accessing resources that are outside the subnetwork or from accessing peer VPC networks. Similarly, network access to VM workers from outside the subnetwork or peer VPC networks is prevented.

For more information about using the --usePublicIps pipeline option to specify whether workers should have only private IPs, see Pipeline options.

Developing your pipelines

The way that the code for your pipeline is implemented has a significant influence on how well the pipeline performs in production. In order to develop and deploy battle-tested code, we recommend the following:

- **Pipeline runners:** Use different Apache Beam runners to run pipeline code. The Apache Beam SDK provides a Direct Runner for local development and testing. You can use the Dataflow Runner for ad hoc development testing and for end-to-end pipeline tests.
- **Deployment environments:** Create deployment environments to separate users, data, code, and other resources across different stages of development. Run pipeline locally for development and rapid testing using the Direct Runner. Create a pre-production environment for for development phases that need to run in production-like conditions, such as end-to-end testing. The production environment should be a dedicated Google Cloud project, where continuousl delivery copies deployment artifacts to the production environment when all end-to-end tests have passed.
- **Leverage open-source:** Apache Beam provides a rich set of pipeline examples in its directories for developers to copy from. The Dataflow Cookbook provides a library of transformations and common patterns in Java, Scala, Python, and Go. There are also guides for common use-case patterns found in our documentation. Google-provided templates are open source under the Apache License 2.0, so you can use them as the basis for new pipelines. The templates are also useful as code examples for reference.
- **Test pipeline code:** Use unit tests, integration tests, and end-to-end tests when applicable. The Apache Beam SDK provides functionality to enable these tests. The Apache Beam SDK provides functionality to enable these tests. Ideally, each type of test targets a different deployment environment. The following diagram illustrates how unit tests, integration tests, and end-to-end tests apply to different parts of your pipeline and data.



Apache Beam best practices

In addition to the guidance listed above, Apache Beam and Dataflow provides features that complement these best practices for improved production readiness.

- **Leverage turnkey transforms:** Turnkey transformations provide a utility for developers to accomplish common business logic patterns in the convenience of a transform. These transformations abstract away unnecessary overhead that can take dozens, if not hundreds of lines, to manage.
  - **Enrichment**: When you enrich data, you augment the raw data from one source by adding related data from a second source. The additional data can come from a variety of sources, such as Bigtable or BigQuery. The Apache Beam enrichment transform uses a key-value lookup to connect the additional data to the raw data.
  - **RunInference**: Users oftentimes needs to make a call for a prediction from an ML model stored externally to the pipeline. The RunInference API enables you to run models as part of your pipeline in a way that is optimized for machine learning inference. To reduce the number of steps in your pipeline, RunInference supports features like batching.

- **Micro-batch calls to external services:** When you call external services, you can reduce per-call overheads by using the GroupIntoBatches transform to create batches of elements of a specified size. Batching sends elements to an external service as one payload instead of individually. In combination with batching, you can limit the maximum number of parallel (concurrent) calls to the external service by choosing appropriate keys to partition the incoming data. The number of partitions determines the maximum parallelization. For example, if every element is given the same key, a downstream transform for calling the external service does not run in parallel.
- **Queue unprocessable data:** Your pipeline might encounter situations where it's not possible to process elements. This situation can occur for different reasons, but a common cause is data issues. use a pattern called a dead-letter queue (or dead-letter file). Catch exceptions in the DoFn.ProcessElement method and log errors as you normally would. Instead of dropping the failed element, use branching outputs to write failed elements into a separate PCollection object. These elements are then written to a data sink for later inspection and handling by using a separate transform.

Deploying your pipelines

Deployment

Pipeline development involves different stages and tasks, such as code development, testing, and delivery into production. Deploying a pipeline within a robust continuous integration & continuous delivery framework can ensure safe rollouts that do not introduce regressions.

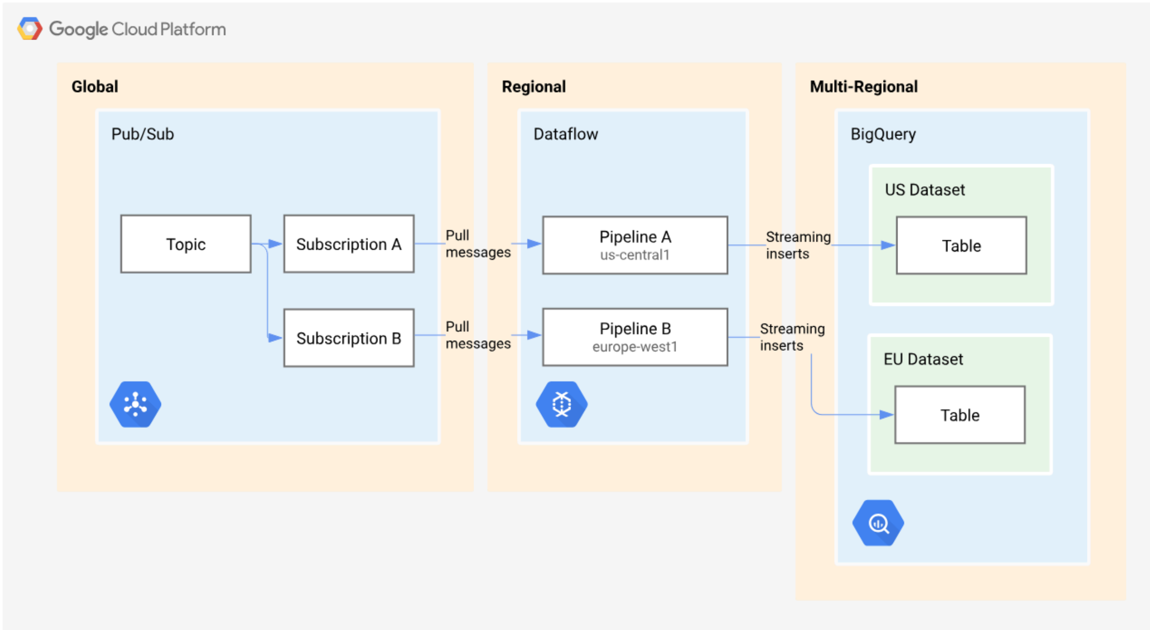
The number and types of deployment artifacts created from a passing build varies depending on how pipelines are launched. Using the Apache Beam Java SDK, you can package your pipeline code into a self-executing JAR file. You can then store the JAR file in a bucket that is hosted in the project for a deployment environment, such as the preproduction or production Google Cloud project. If you use Classic Templates (a type of templated execution), the deployment artifacts include a JSON template file, the JAR file for your pipeline, and an optional metadata template. You can then deploy the artifacts into different deployment environments using continuous delivery.

You can create a Dataflow job by using the Apache Beam SDK directly from a development environment. This type of job is called a non-templated job. Although this approach is convenient for developers, you might prefer to separate the tasks of developing and running pipelines. To make this separation, you can use Dataflow templates, which allow you to stage and run your pipelines as independent tasks. After a template is staged, other users, including non-developers, can run the jobs from the template using the Google Cloud CLI, the Google Cloud console, or the Dataflow REST API.

Reliability

It is important to consider the failures that might occur in the event of an outage. The following are a few principles to keep in mind:

- **Follow isolation principles:** A general recommendation to improve overall pipeline reliability is to follow the isolation principles behind regions and zones. Ensure that your pipelines don't have critical cross-region dependencies. If you have a pipeline that has critical dependency on services from multiple regions, a failure in any one of those regions can impact your pipeline. To help avoid this issue, deploy to multiple regions for redundancy and backup.
- **Create Dataflow snapshots:** Dataflow offers a snapshot feature that provides a backup of a pipeline's state. You can restore the pipeline snapshot into a new streaming Dataflow pipeline in another zone or region. You can then start the reprocessing of messages in the Pub/Sub or Kafka topics starting at the snapshot timestamp. If you set up regular snapshots of your pipelines, you can minimize Recovery Time Objective (RTO) time.
- **Mitigate regional outages by using high availability or failover:** For streaming jobs, depending on the fault tolerance and budget for your application, you have different options for mitigating failures. For a regional outage, the simplest and most cost-effective option is to wait until the outage ends. However, if your application is latency-sensitive or if data processing must either not be disrupted or should be resumed with minimal delay, there are a couple of architectural options to consider:
    - *High-availability: Latency sensitive with no data loss:* If your application cannot tolerate data loss, run duplicate pipelines in parallel in two different regions, and have the pipelines consume the same data. The same data sources need to be available in both regions. The downstream applications that depend on the output of these pipelines must be able to switch between the output from these two regions. Due to the duplication of resources, this option involves the highest cost compared to other options.



  - *Failover: Latency-sensitive with some potential data loss:* If your application can tolerate potential data loss, make the streaming data source available in multiple regions. For example, using Pub/Sub, maintain two independent subscriptions for the same topic, one for each region. If a regional outage occurs, start a replacement pipeline in another region, and have the pipeline consume data from the backup subscription.
- **Don't store data in the broker for long periods of time:** There is no need to do that with Dataflow for increasing robustness. Dataflow ensures exactly once processing and will not pull data more than once from the broker. Dataflow can do live updates to ensure continuity without gaps.

Operating your pipelines

Monitoring

Cloud Monitoring provides powerful logging and diagnostics. Dataflow integration with Monitoring lets you access Dataflow job metrics such as job status, element counts, system lag (for streaming jobs), and user counters from the Monitoring dashboards. You can also use Monitoring alerts to notify you of various conditions, such as long streaming system lag or failed jobs.

A streaming pipeline is a service. Start by defining a SLI and SLO of what you want to achieve with this pipeline, as mentioned when we discussed planning your pipeline. Leverage Cloud

Monitoring to measure the SLO automatically, and determine error budget and other properties without additional tooling needed.

To maintain a robust monitoring posture, the following best practices are recommended:

- **Setup relevant dashboards:** Dataflow publishes a list of metrics relevant to the performance of your pipeline. These include metrics describing resource utilization, data freshness, system latency, parallelism, and read/write throughput. Creating charts that are directly relevant to your SLOs and SLIs can simplify assessing the performance of your pipeline.
- **Inspect logs & insights:** Worker and job logs can indicate performance issues with your pipeline. Establish a practice of reviewing the diagnostics panel of the Dataflow monitoring UI, and review insights that can improve job performance, reduce cost, or troubleshoot errors.
- **Create alerts:** You can create an alerting policy directly from a metric chart. Creating alerts that align with your SLOs and SLIs can help your organization respond to a regression in your pipeline before a service outage.

Performance

Echoing the guidance regarding defining SLOs and SLIs that are relevant to the key performance indicators that you care about, there are a number of considerations that should be taken into account when optimizing the performance of your pipeline. Delivering better performance for your pipeline (defined by either lower latency or higher efficiency – or both) can generally be done by carefully observing your pipeline in production into your pipeline implementation. The following list are recommended areas to identify for performance improvements (albeit not exhaustive):

- **Region, quotas, and networking:** Confirm that the pipeline is being run in the region where resources are located. Check if networking parameters permit adequate throughput for the pipeline.
- **Excessive data shuffling:** Stages where data is shuffled (i.e. when you are using GroupByKey, CoGroupByKey, or Stateful DoFns) are generally the most computationally expensive parts of a pipeline. These stages consume the most network bandwidth and if you are not using Streaming Engine or Shuffle, will likely add a meaningful amount of processing time to your pipeline. To mitigate against this, be intentional about the data that you are shuffling. Remove as many unused attributes as possible before you invoke a grouping transformations. When possible, use a combiner instead of a GroupByKey.
- **Limited parallelism:** Certain grouping operations might be slow, and you might see “hot key warnings” displayed in the UI. This is usually a result of a skew found in the

distribution of the data, which will cause the overall processing to slow down. If possible, redesign the key partitioning or use combiners. Beam also provides several variations of “withHotKeyFanout” transformations to alleviate the ill effects of hot keys.

- **Batch External API calls:** As mentioned in the best practices section, you can reduce the impacts of making calls to external APIs by microbatching calls. Calling external APIs can sometimes be subjected to quotas that are not known to your pipeline, which can serve to slow down your own pipeline. Implement DoFn’s lifecycle methods (annotated with @Setup, @Startbundle), collect API requests in the @ProcessElement method, and call the API in the @FinishBundle method; or ose GroupIntoBatches transform to creates batches of certain sizes. It’s important to ensure that calls are idemponent, since bundles can be replayed.
- **IO Specific Recommendations:** Depending on your pipeline, review the best practices guides on IOs. Oftentimes, performance can be significantly constrained by settings configured at the source & sink.

Cost Optimization

Being intentional about upholding performance best practices will generally result in the most cost-efficient pipelines. However, there are important steps to take when monitoring your pipeline costs, as well as configurations that can reduce your overall bill without coming at the expense of your requirements:

- **Monitor Dataflow costs & set alerts:** Enable billing export into BigQuery. This is the most efficient way of observing your Dataflow costs at the projet level down the pipeline and SKU level. Implement a labeling taxonomy using the best practices found here, and add labels to Dataflow jobs. For your critical pipelines, create monitoring alerts to do real-time notifications and cost control.
- **Number of workers:** Once you have a good sense for the data volume patterns of your pipeline, setting a maximum number of workers can limit any adverse traffic spikes that might cause your pipeline to overprovision workers. On the other hand, if you know that you have a baseline number of workers that you would like to maintain in order to guarantee uptime, you can set a minimum number of workers to ensure your processing doesn’t slow down during an autoscaling event.
- **At-least-once mode:** If your pipeline does not require exactly-once processing (i.e. deduplication can be handled in your sink or destination), you can configure your pipeline to run in at-least-once mode. At-least-once mode can help optimize cost and performance by turning off our exactly-once mode, which adds to pipeline latency & cost due to the reading of metadata required to ensure we process a message no more than once.