Maximizing Performance, Reliability, and Scalability with Dataflow Streaming

Apache Beam Summit

1:00 PM Tuesday, 23 June 2026

Tom Stepp & Ryan Wigglesworth, Google

Table of Contents

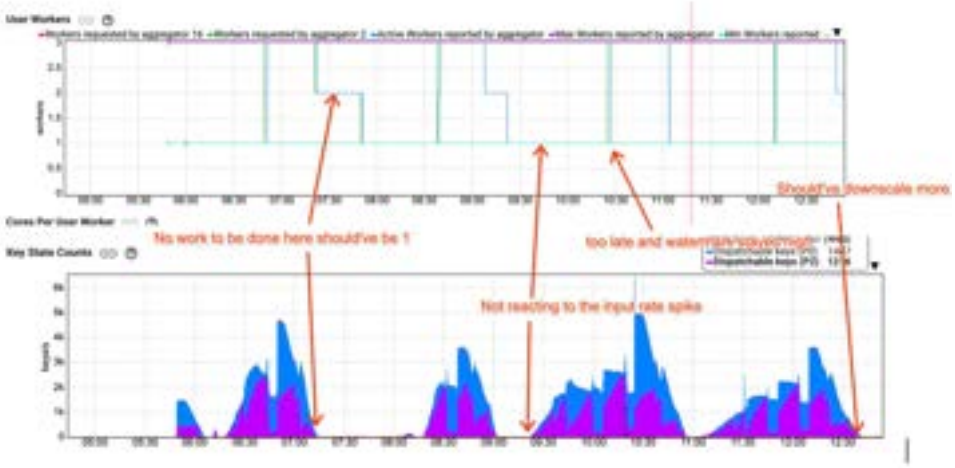
Streaming ML

Async DoFn

- For long running Python or Java Transforms
- Prevents retries allowing long running transforms to finish without premature interruptions from requests hedging
- Also added asyncio support in Python to allow reusing threads waiting on remote calls, increase parallelism by up to 500x

GPU Autoscaling

- Legacy autoscaling is designed for operations that take milliseconds to seconds, rather than seconds to minutes like many ML steps do.
- Autoscaling needs special tuning for this case



How to use GPU Autoscaling

- Marking computation
  - Using resource hints and passing in max_parallelism_per_worker at the same time.
  - Example usage:
    - pcoll | MyPTransform().with_resource_hints(

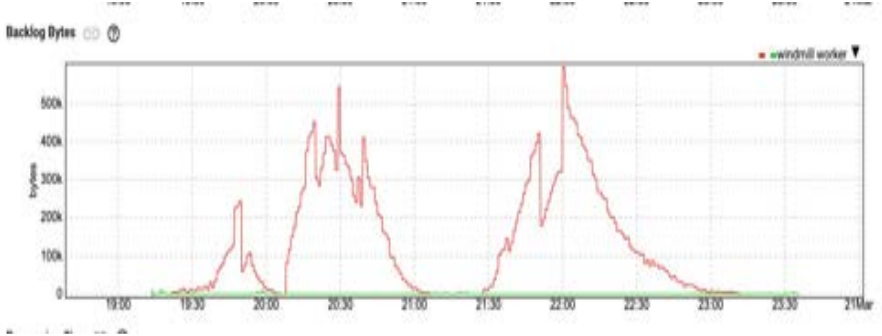
      min_ram="4GB",

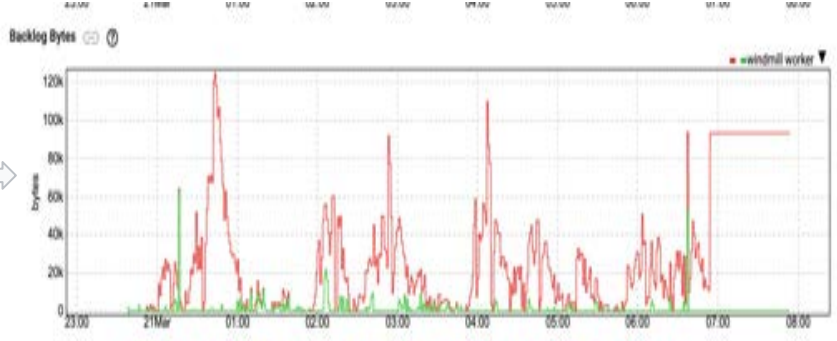
      accelerator="type:nvidia-tesla-l4;count:1;install-nvidia-driver",

      max_parallelism_per_worker=2,

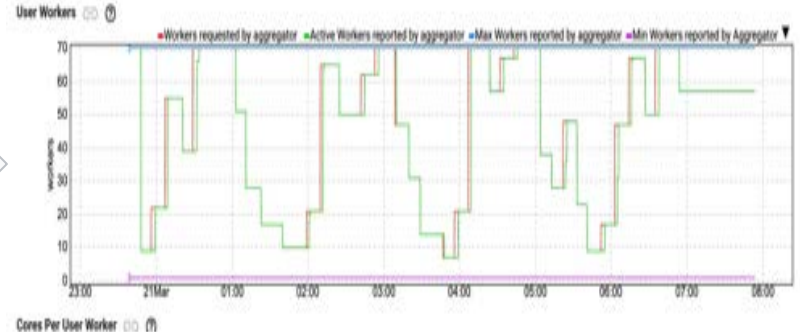
)

Experiment


Backlog Bytes
windmill worker
bytes
500k
400k
300k
200k
100k
0
19:00
19:30
20:00
20:30
21:00
21:30
22:00
22:30
23:00
23:30
21Mar

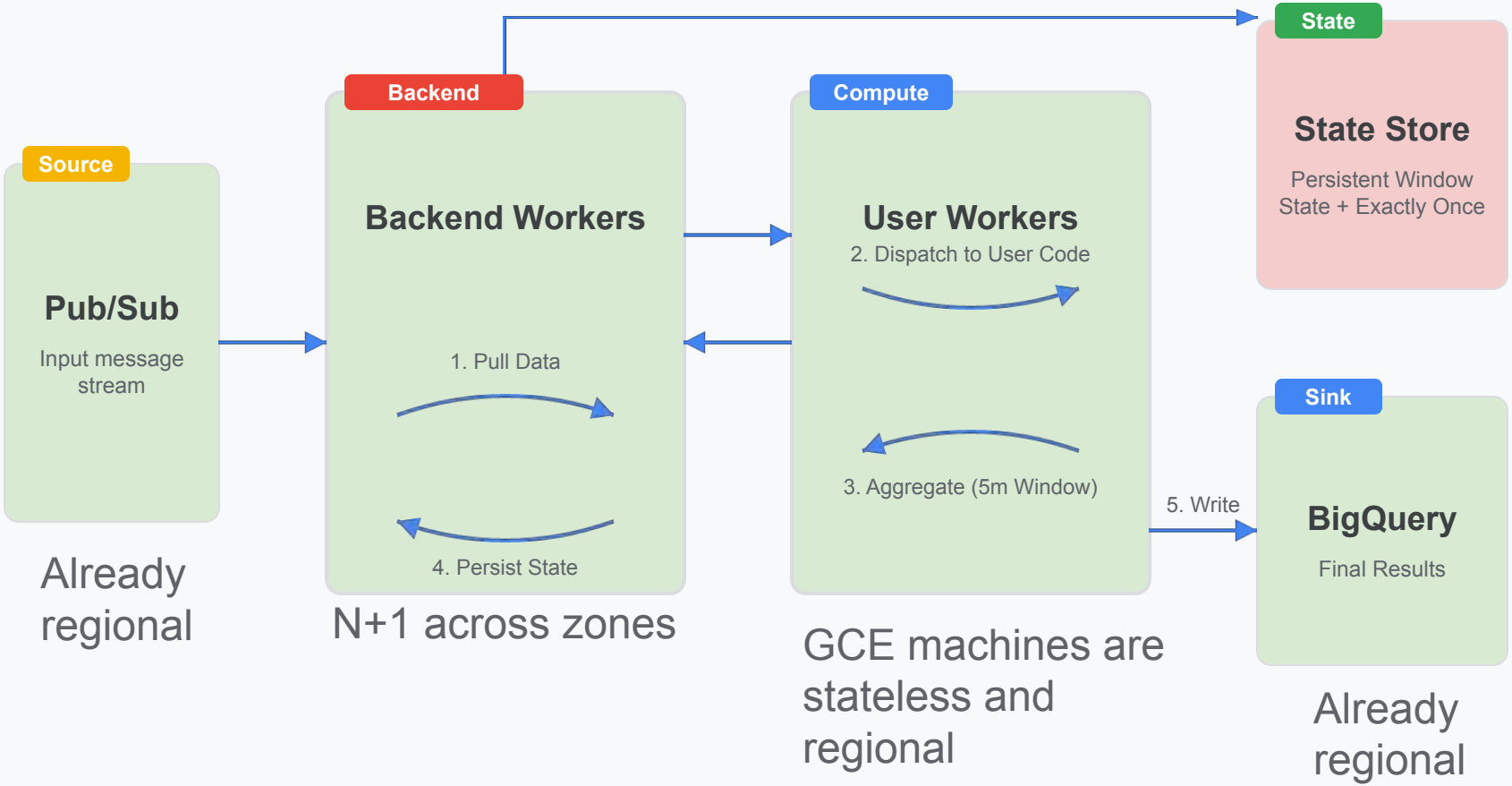

Backlog Bytes
windmill worker
bytes
120k
100k
80k
60k
40k
20k
0
23:00
21Mar
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00


User Workers
Workers requested by aggregator
Active Workers reported by aggregator
Max Workers reported by aggregator
Min Workers reported by Aggregator
workers
70
60
50
40
30
20
10
0
19:00
19:30
20:00
20:30
21:00
21:30
22:00
22:30
23:00
23:30
21Mar


User Workers
Workers requested by aggregator
Active Workers reported by aggregator
Max Workers reported by aggregator
Min Workers reported by Aggregator
workers
70
60
50
40
30
20
10
0
23:00
21Mar
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00
Cores Per User Worker

High Availability

Jobs are at risk of state store zonal failure



Dataflow HA Advantages

Fully Managed

Fits with our fully managed position versus competitors.

- Single setup option
- Automatic failover
- No data loss
- ~1 min pause only

Cost Efficient

Only replicates the necessary portions of the infrastructure.

- Just replication
- ~10% of total costs
- Avoids full pipeline duplication

Deep Integration

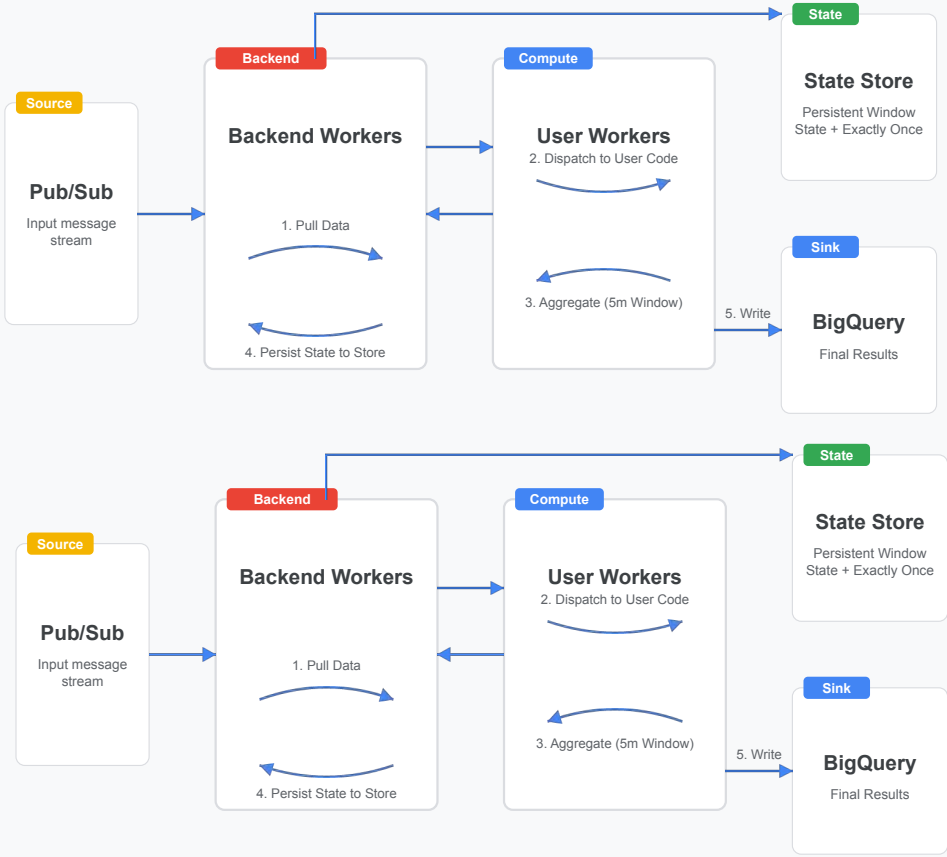
Works seamlessly with Google Cloud's regional ecosystem.

- Regional sources/sinks
- Fits better with the rest of BQ system

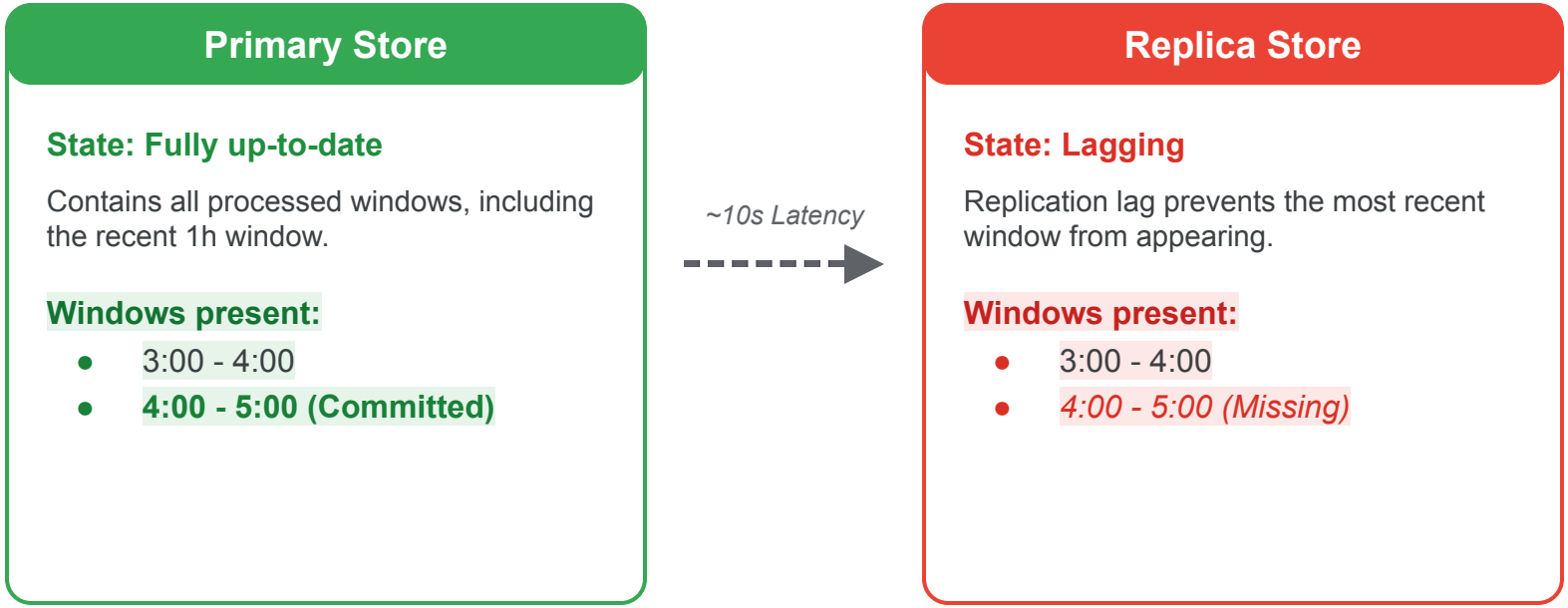
Targeting GA in Q4

Customer Implementation Requires Duplicating Resources

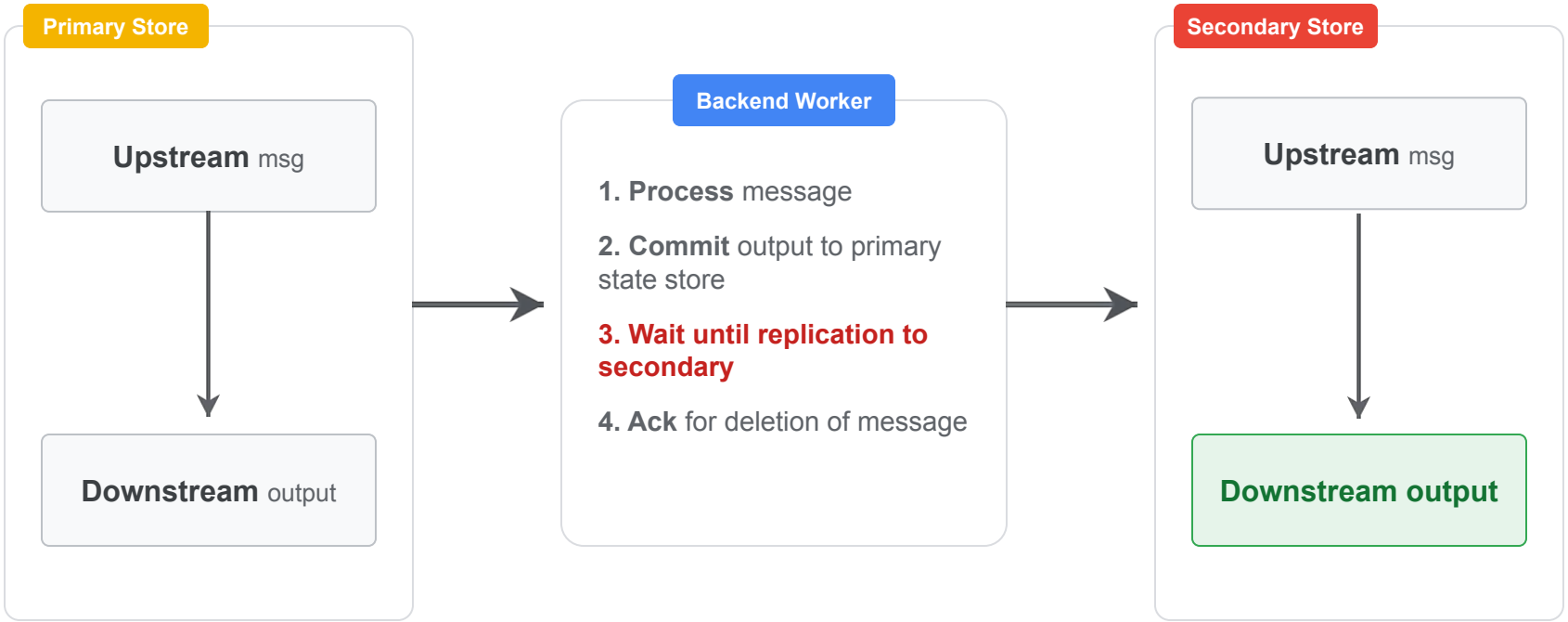
- Customers who want regional availability have to duplicate all resources.
- Both the Dataflow pipeline itself but also the source and sink data.
- Managing the failover between the primary and secondary is non-trivial as well.



We can replicate the state but not lossless



Dataflow exactly once



To achieve zero data loss, Dataflow ensures replication before issuing upstream deletes. If failover occurs, we are guaranteed that input exists or output was replicated. If both exist then standard deduplication resolves the conflict.

Iceberg Ingestion

Benchmarking Results

- **Throughput**: Dataflow Streaming initially took 3 hours to clear 1 TB of backlog.
- **File Size**: Originally produced parquet files of average 38 MB per file.
    - Note: Writing large files (e.g. 500 MB) is desirable since they are more optimal for reads.
- **Resources**: Dataflow had poor CPU utilization versus similar workloads with other IOs.

Improvements

- **Compression** → Use table-defined compression algorithm (defaulting to better zstd over gzip).
- **Metadata Caching** → Improve inefficient metadata lookups & prevent quota exhaustion for BigLake Metastore.
- **Direct Writes** → Avoid expensive GroupIntoBatches (GIB) by directly writing large bundles.
- **Autosharding** → Default range-based algorithm forces many shards, resulting in undesirable small files.

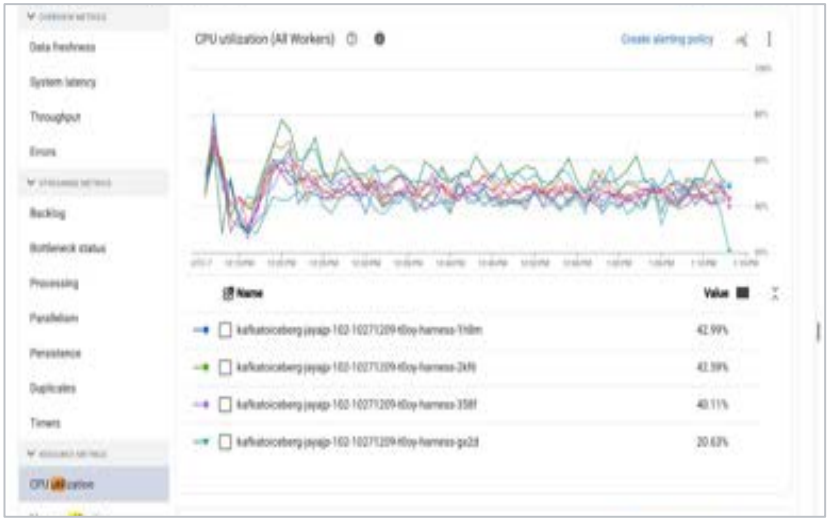
Compression: Overview

	Prior: gzip	Current: zstd
Year Released	1992	2016
Performance	Most reports show that zstd outperforms gzip on the following factors - compression ratio, CPU utilization, wall-time.	
Behavior with incompressible data	Still attempt to apply its compression algorithms leading to increased CPU usage and potentially larger files.	Zstd can detect incompressible data and switch to a "passthrough" mode.
Usage	Used on Beam before 2.70.0	Used on Beam after 2.70.0

https://github.com/apache/beam/pull/36542

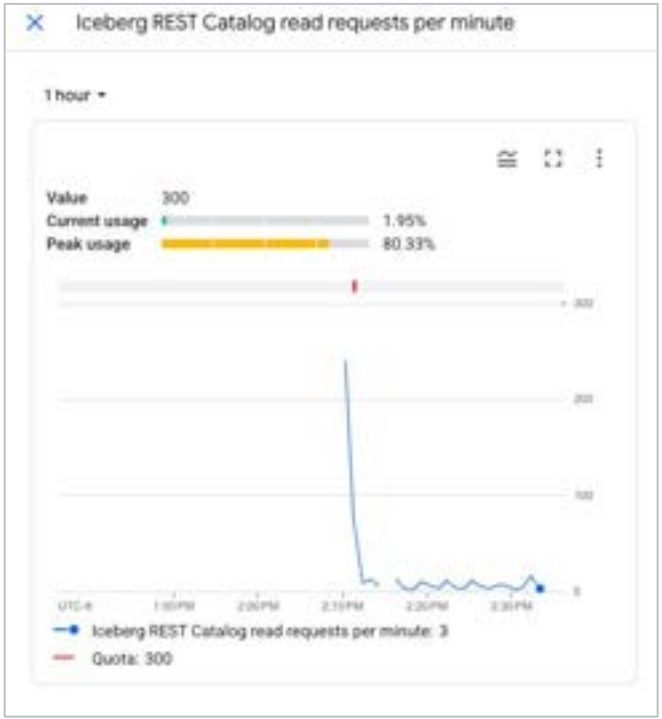
Compression Load Tests

Time to clear backlog 1.2tb Kafka backlog: 3 hours -> 1 hour 40 minutes



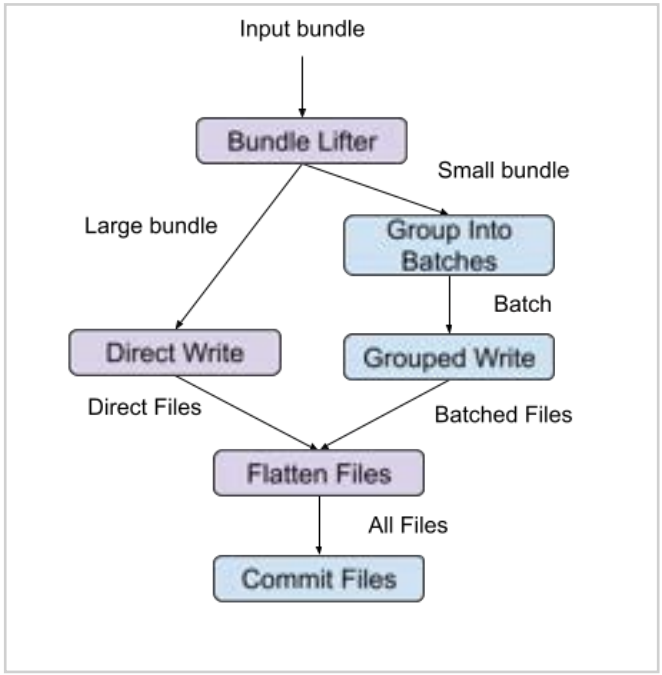
Metadata Caching

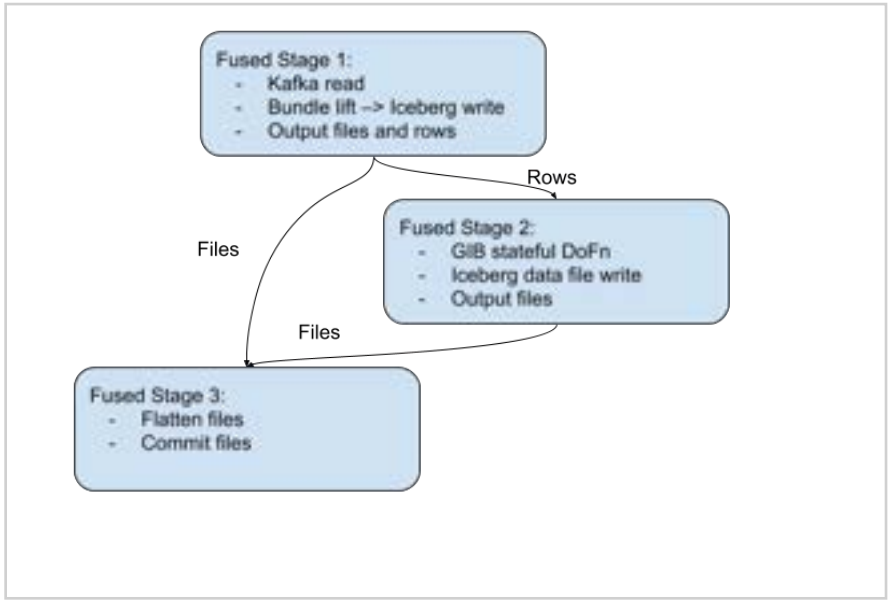
- Testing quickly exhausted BigLake Metastore Read Quota due to refreshing table metadata on each input.
- This blocked processing and reduced performance.
- Optimized by caching and refreshing periodically.



https://github.com/apache/beam/pull/37102

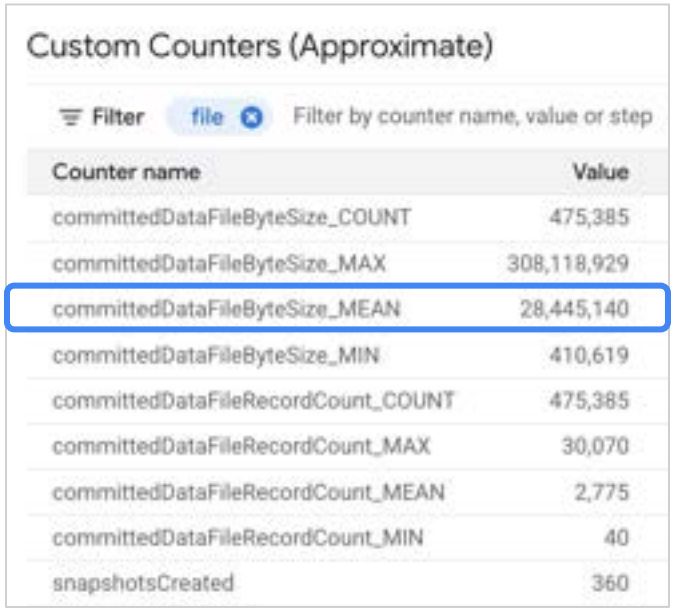
Direct Writes: Overview





https://github.com/apache/beam/pull/36720

Direct Writes: File Sizes

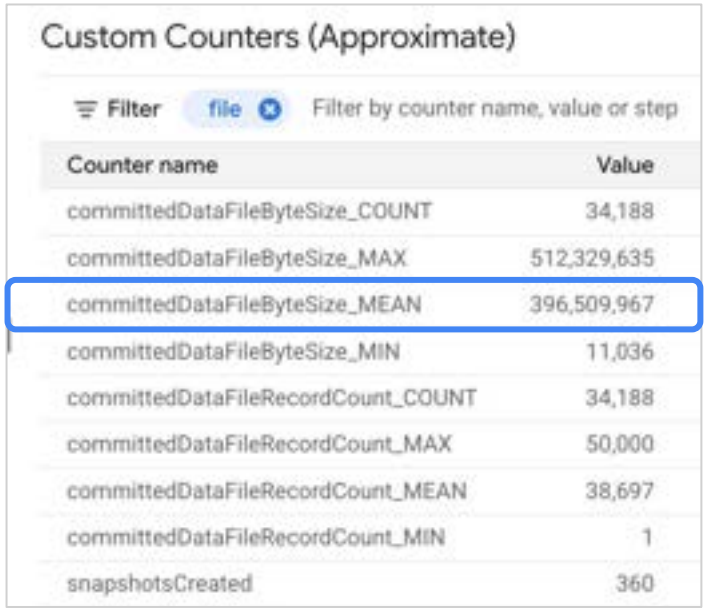


Custom Counters (Approximate)

Filter | file | Filter by counter name, value or step

Counter name	Value
committedDataFileByteSize_COUNT	475,385
committedDataFileByteSize_MAX	308,118,929
committedDataFileByteSize_MEAN	28,445,140
committedDataFileByteSize_MIN	410,619
committedDataFileRecordCount_COUNT	475,385
committedDataFileRecordCount_MAX	30,070
committedDataFileRecordCount_MEAN	2,775
committedDataFileRecordCount_MIN	40
snapshotsCreated	360

Baseline



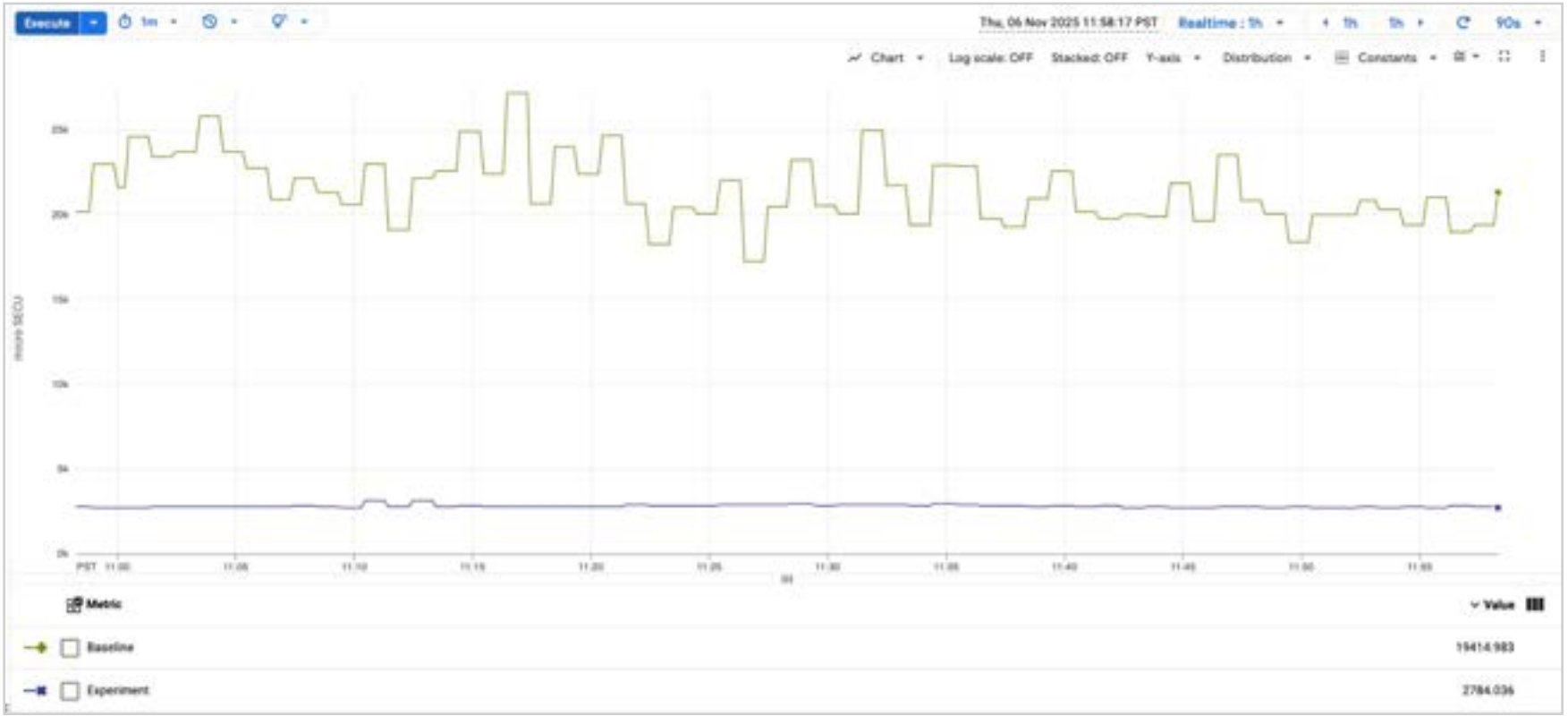
Custom Counters (Approximate)

Filter | file | Filter by counter name, value or step

Counter name	Value
committedDataFileByteSize_COUNT	34,188
committedDataFileByteSize_MAX	512,329,635
committedDataFileByteSize_MEAN	396,509,967
committedDataFileByteSize_MIN	11,036
committedDataFileRecordCount_COUNT	34,188
committedDataFileRecordCount_MAX	50,000
committedDataFileRecordCount_MEAN	38,697
committedDataFileRecordCount_MIN	1
snapshotsCreated	360

Experiment

Direct Writes: Cost



Motivation for Autosharding in Iceberg

- Iceberg Sink is Over-Parallelized
  - Each worker tries to 40 parallel files at once. Results in small files & slow queries.
- Throughput-based autosharding is a better fit.
  - Allows Iceberg sink to dynamically adjust parallelism.
  - It is already used for BigQuery and GCS IO sinks.

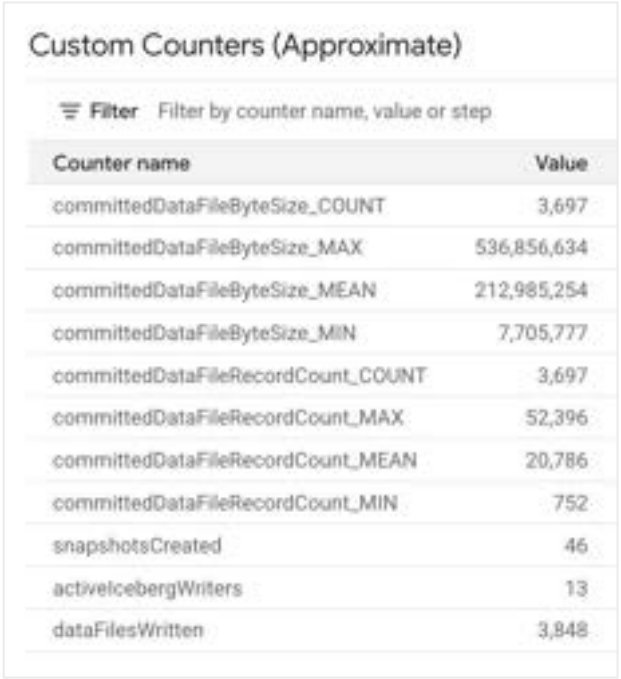
Iceberg Autosharding: File Sizes

Custom Counters (Approximate)

Filter Filter by counter name, value or step

Counter name	Value
committedDataFileByteSize_COUNT	196,056
committedDataFileByteSize_MAX	48,638,921
committedDataFileByteSize_MEAN	3,946,035
committedDataFileByteSize_MIN	10,996
committedDataFileRecordCount_COUNT	196,056
committedDataFileRecordCount_MAX	4,747
committedDataFileRecordCount_MEAN	385
committedDataFileRecordCount_MIN	1
snapshotsCreated	46
activeIcebergWriters	15
dataFilesWritten	196,885

Baseline



Custom Counters (Approximate)

Filter Filter by counter name, value or step

Counter name	Value
committedDataFileByteSize_COUNT	3,697
committedDataFileByteSize_MAX	536,856,634
committedDataFileByteSize_MEAN	212,985,254
committedDataFileByteSize_MIN	7,705,777
committedDataFileRecordCount_COUNT	3,697
committedDataFileRecordCount_MAX	52,396
committedDataFileRecordCount_MEAN	20,786
committedDataFileRecordCount_MIN	752
snapshotsCreated	46
activeIcebergWriters	13
dataFilesWritten	3,848

Experiment

Iceberg Autosharding: File Size Sanity Check

Offset Deduplication

Offset Deduplication

- Problem:
  - Redistribute is expensive due to exactly-once cost and latency.
  - Can’t use allow duplicates optimization because you need exactly-once.
- Solution:
  - Utilize source metadata, such as message offsets, to make shuffle cheaper.
  - Enabled for KafkaIO.Read.withRedistribute() on Beam 2.70+

Implementation

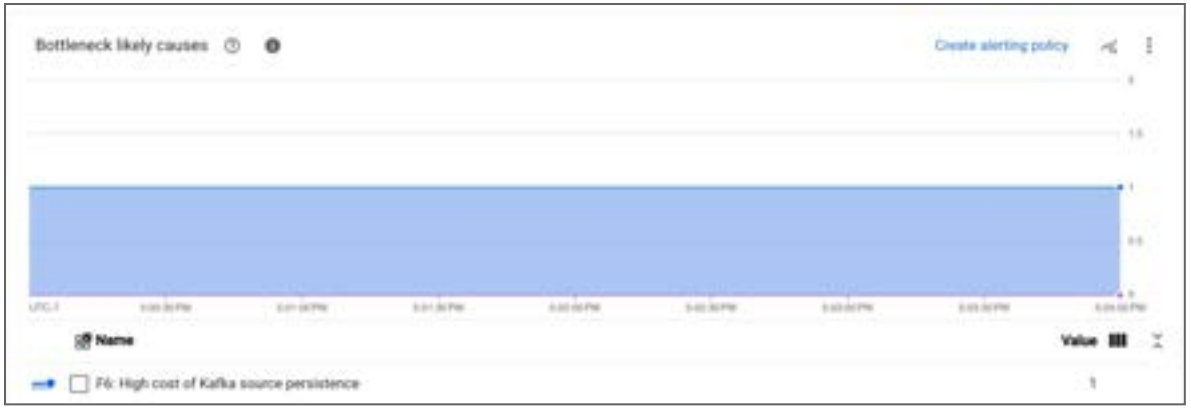
- Beam SDK
  - Reader interface to provide per record offsets to Dataflow runner.
  - Source interface to provide offset limit for checkpoints.
- Beam Runner
  - Message offsets for records committed to backend.
  - Offset limit for source states committed to backend.

Results

Cost

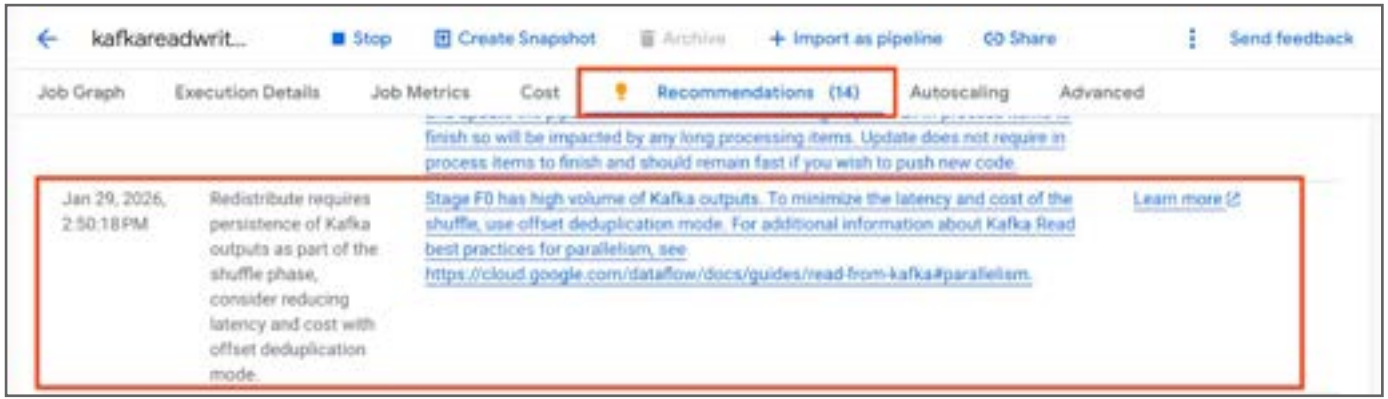
Latency

Observability



Bottleneck Detector
Helps identify when the Kafka source persistence is slow and the docs linked below help guide to use the new feature.

Recommendations
Recommendation also shown based on the analysis of bottleneck detector.



High Cardinality

High Cardinality Improvements

Strengths

Dataflow Streaming's ability to handle high scale and cardinality is a key competitive advantage.

Key Customers

- Top customers run large stateful jobs.
- **Throughput:** Millions of messages/s
- **Cardinality:** 100s of million keys

Challenges

- Some of these jobs stress the limits of our backend system.
- Resource utilization on large jobs is a concern.
- Customers want better performance at a lower cost

High Cardinality Improvements - Projects

Multi-Key Bundles

Small bundle sizes hinder performance of streaming pipelines.

Enable large cross key bundles, significantly improving batching and combiner lifting efficiency.

Reduce IO & Resource usage

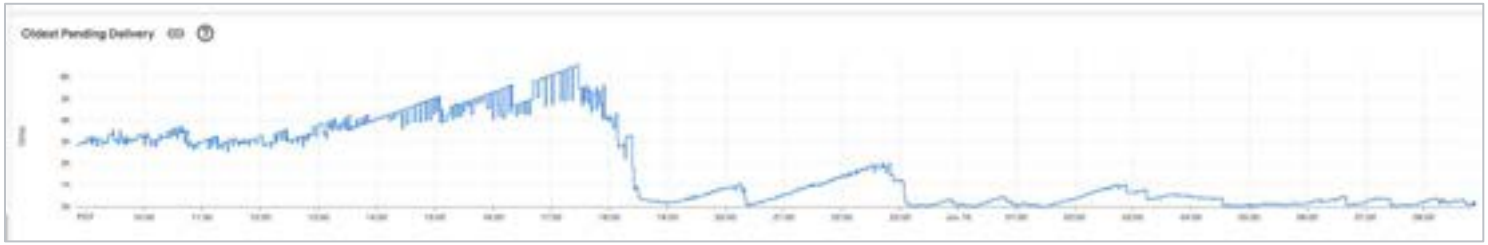
- Reducing state reads by improving state cache.
- Optimizing user worker budgeting for better utilization.

Shuffle Enhancements

Shuffle Enhancements

- Problem:
  - Upstream backend worker not aware of downstream worker resource limitations.
  - Sending individual requests on new RPCs use more resources.
  - Ordering limitations with individual requests.
- Solution:
  - Using long-lived connections reduces resource usage, provides simple way to communicate budgets between pairs, and allows pipelining

First Milestone Results



Parallel Updates

Streaming Job Updates

Current Challenges in Dataflow Streaming Job Updates:

- **Extended Downtime:** In-place updates currently result in substantial service interruptions, typically requiring a 15-minute maintenance window.
- **Job Incompatibility:** Frequent compatibility conflicts require specialized, manual workarounds during the update process.

Project Objective:

- To optimize customer onboarding by reducing manual intervention and lowering operational overhead. Provide declarative APIs.

Parallel Update Features

- Automated stop and replace pipelines streaming job update.
- Automated parallel pipeline updates have been extended with
  - An option to auto cancel the previous job after a drain timeout.
  - Allow creating new parallel jobs with the same name.
- Ability to choose job update strategy: in place vs parallel job update.

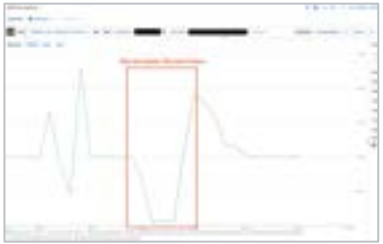
Stop and replace pipelines:

Stop the old job and then start the new job with preallocated compute resources

`parallel_replace_job_max_stop_duration=DRAIN_TIMEOUT_DURATION`

Optional drain timeout, set based on production usage.

Use case: Auto cancel stuck drain e.g. currently looping timers can cause drain to never finish.



`parallel_replace_job_cancel_on_drain_timeout=true`

Auto cancel on drain timeout. **Default true but only valid if drain timeout set.**

Must not set `parallel_replace_job_min_parallel_pipelines_duration` option.

- For now the `parallel_replace_job_preallocate_compute_resources=true` is default.
- New Job state remains `RUNNING` even during the old job is stopped.

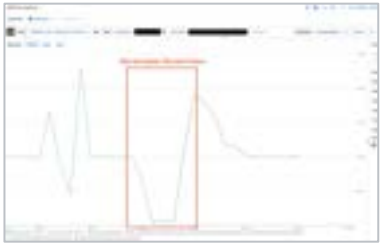
Stop and replace pipelines:

Stop the old job and then start the new job without preallocate compute resources

`parallel_replace_job_max_stop_duration=DRAIN_TIMEOUT_DURATION`

Optional drain timeout, set based on production usage.

Use case: Auto cancel stuck drain e.g. currently looping timers can cause drain to never finish.



`parallel_replace_job_cancel_on_drain_timeout=true`

Auto cancel on drain timeout. **Default true but only valid if drain timeout set.**

Must not set `parallel_replace_job_min_parallel_pipelines_duration` option.

- Explicitly set the `parallel_replace_job_preallocate_compute_resources=false` option.
- New Job state remains `PENDING` until old job is stopped.

Run parallel pipelines (continued):

Run the new job in parallel with the old one. Stop the old job once the new one is ready

`parallel_replace_job_max_stop_duration=DRAIN_TIMEOUT_DURATION`

Optional drain timeout, set based on production usage.

Use case: Auto cancel stuck drain e.g. currently looping timers can cause drain to never finish.

`parallel_replace_job_cancel_on_drain_timeout=true`

Auto cancel on drain timeout. **Default true but only valid if drain timeout set.**

`parallel_replace_job_min_parallel_pipelines_duration=DURATION`

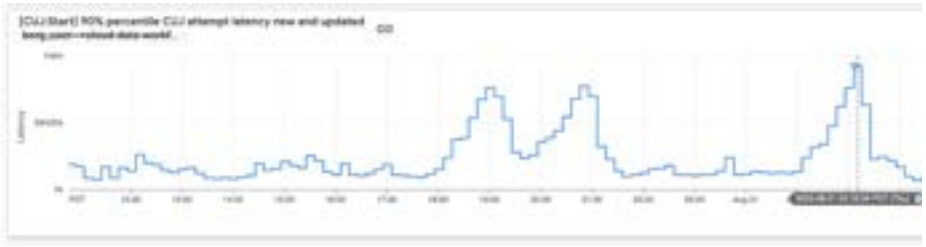
Recommended value `0s`. This example is for non-zero duration.

Two Jobs will be in `running` state for `DURATION` before previous job starts `draining`.

Streaming Engine Reliability

Streaming Engine Reliability Work

- Better support for launching large numbers(O(thousands)) of jobs at once
  - 73% faster P90 job start time



Observability

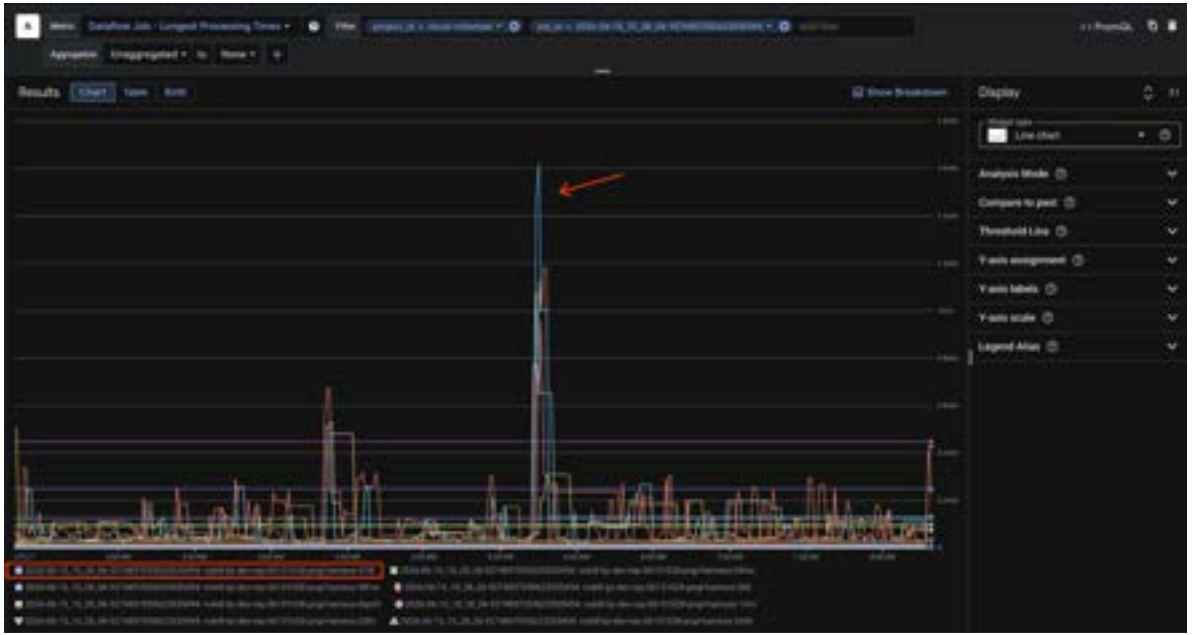
Metrics

Observability: Metrics

- `job/backlogged_keys:` The number of backlogged keys for a bottleneck stage.
- `job/recommended_parallelism:` The recommended parallelism for a stage to reduce bottlenecking.
- `job/is_bottleneck:` Whether a stage is a bottleneck, and the bottleneck kind and likely cause for it.
- `job/longest_processing_time:` The time in microseconds of the longest processing time for a user worker.

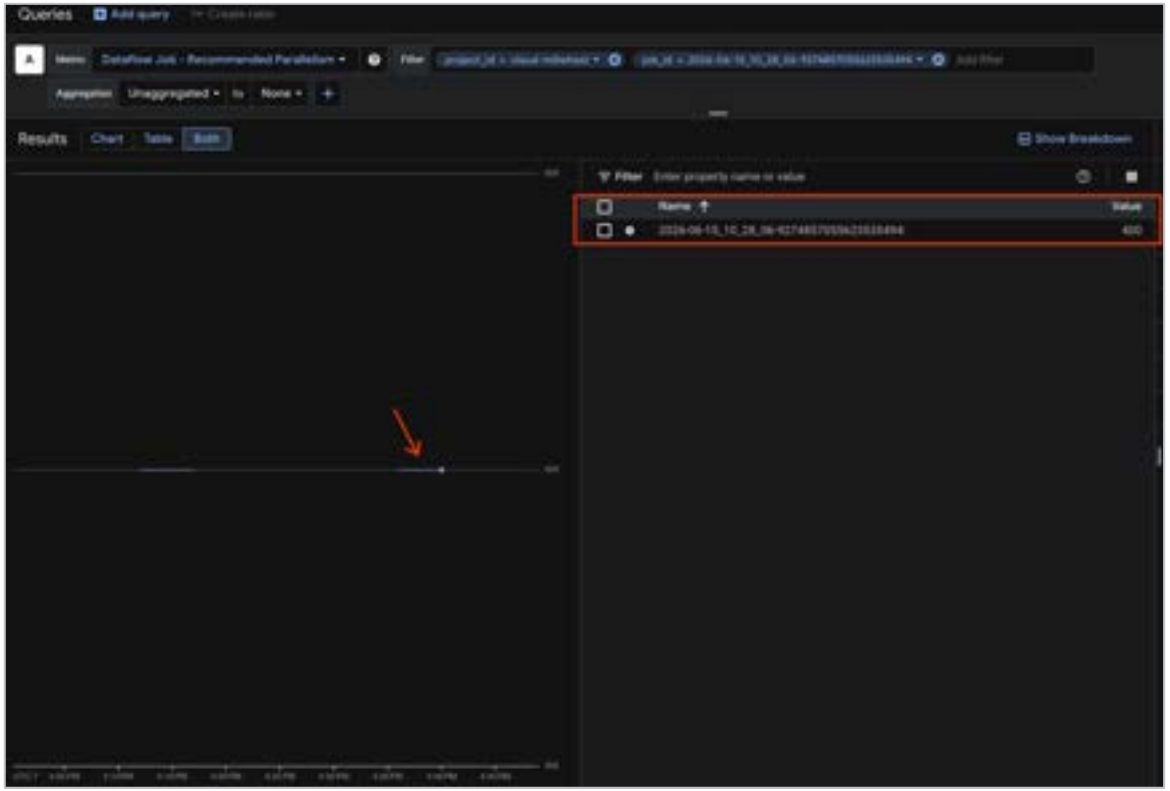
Observability: Metrics

Which worker is slowest?



Observability: Metrics

What parallelism is recommended for the job?



Turnkey Alerts

Observability: Turnkey Alerts

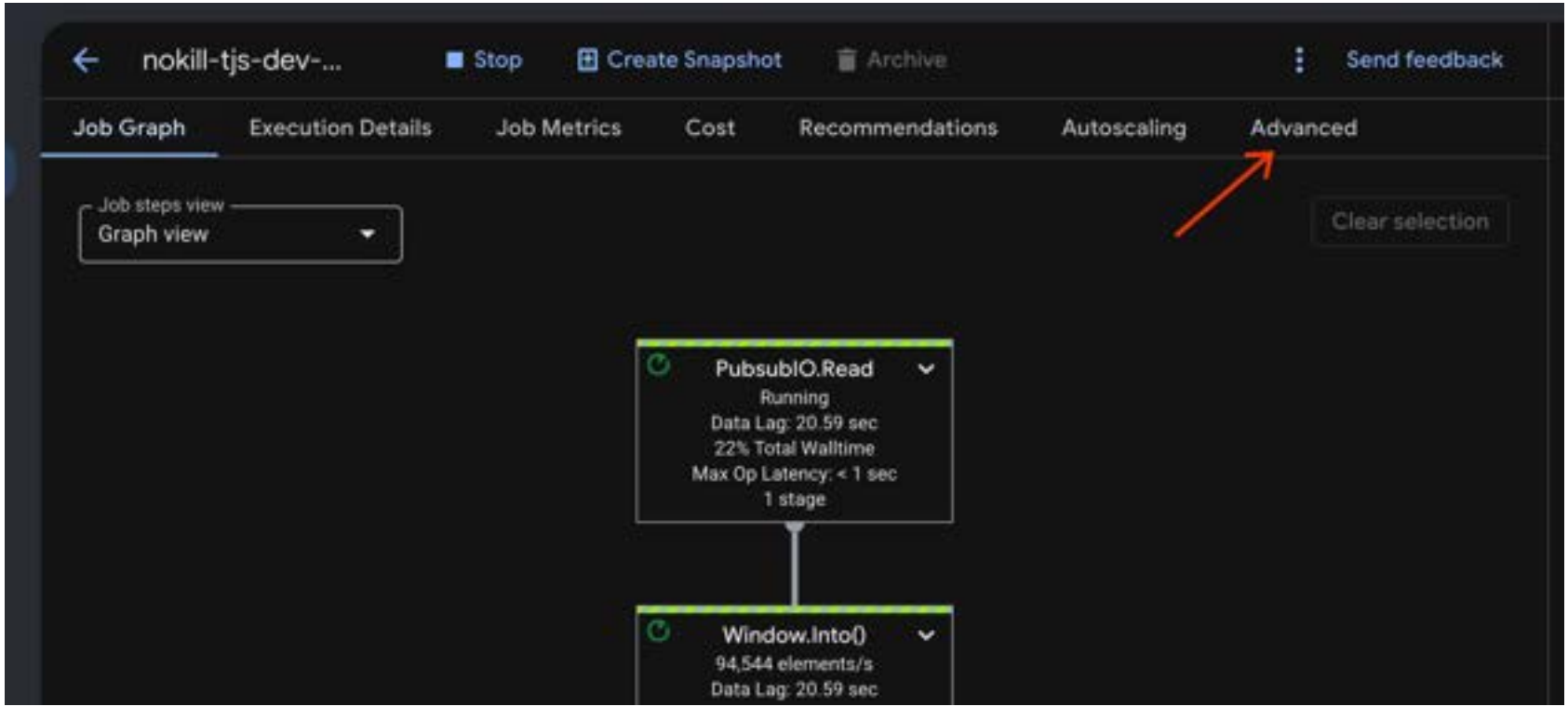
- Predefined suite of alerts to help monitoring your pipelines
- Launching with 2 alerts
  - **Default Policy: Estimated Backlog:** Alerts when the `job/estimated_backlog_processing_time` exceeds a threshold (e.g., 30 minutes).
  - **Default Policy: SECU Usage:** Alerts on significant relative changes (e.g., 50% change) in `job/total_secu_usage` to help monitor costs.
- Alerts can be customized in the Cloud Monitoring console
  - Change parameters or add/subtract new alert queries.
- Launch a job with turnkey alerts via the UI or the cli (`--enable-turnkey-alerts`).
  - Adding turnkey alerts to running jobs is also supported

Advanced Tab

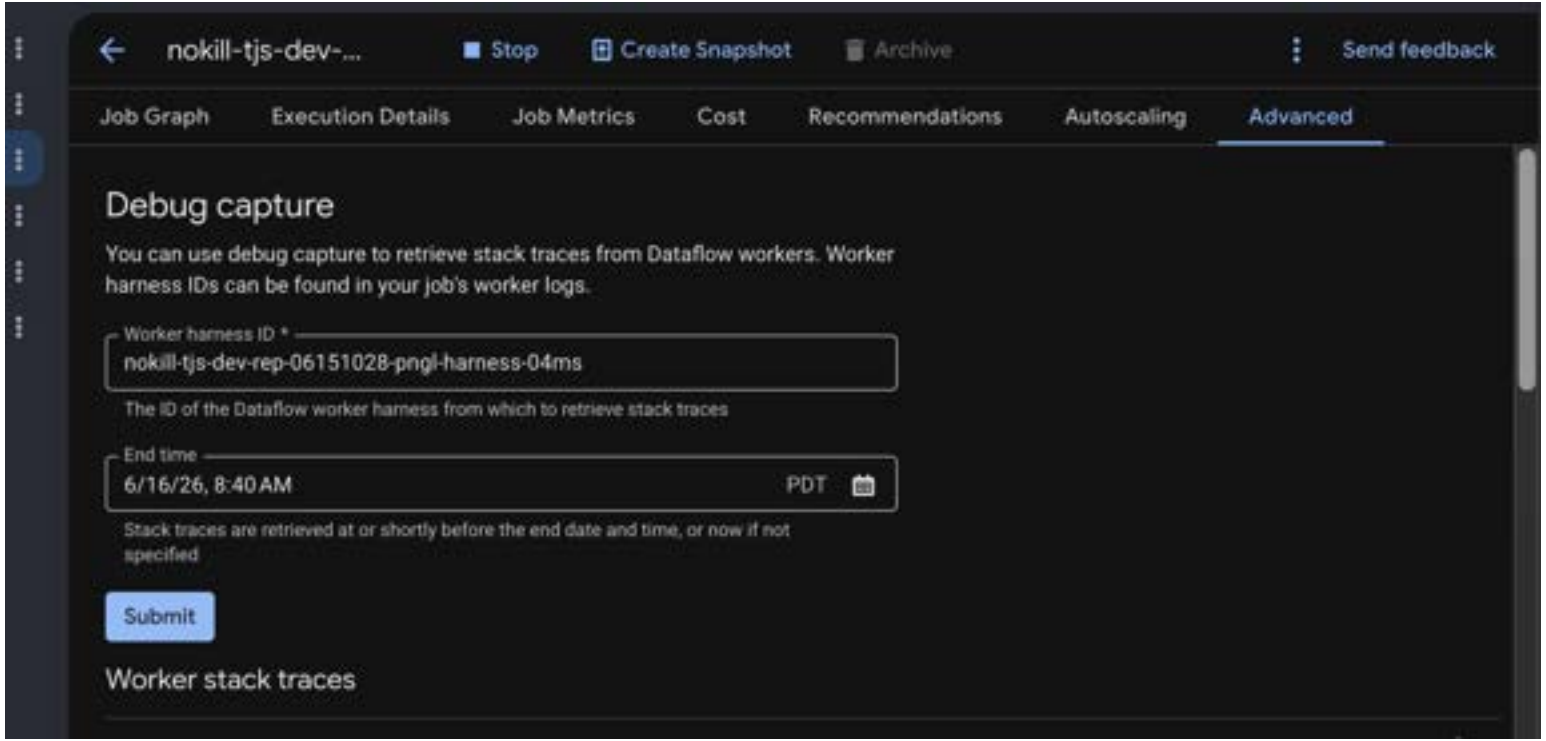
Observability: Advanced Debugging

- Context: Processing was slow by a worker and I want more details.
- Problem: Currently need to dig through logs, only >5m processing is logged.
- Solution: Provide user worker thread stacks in the Job UI

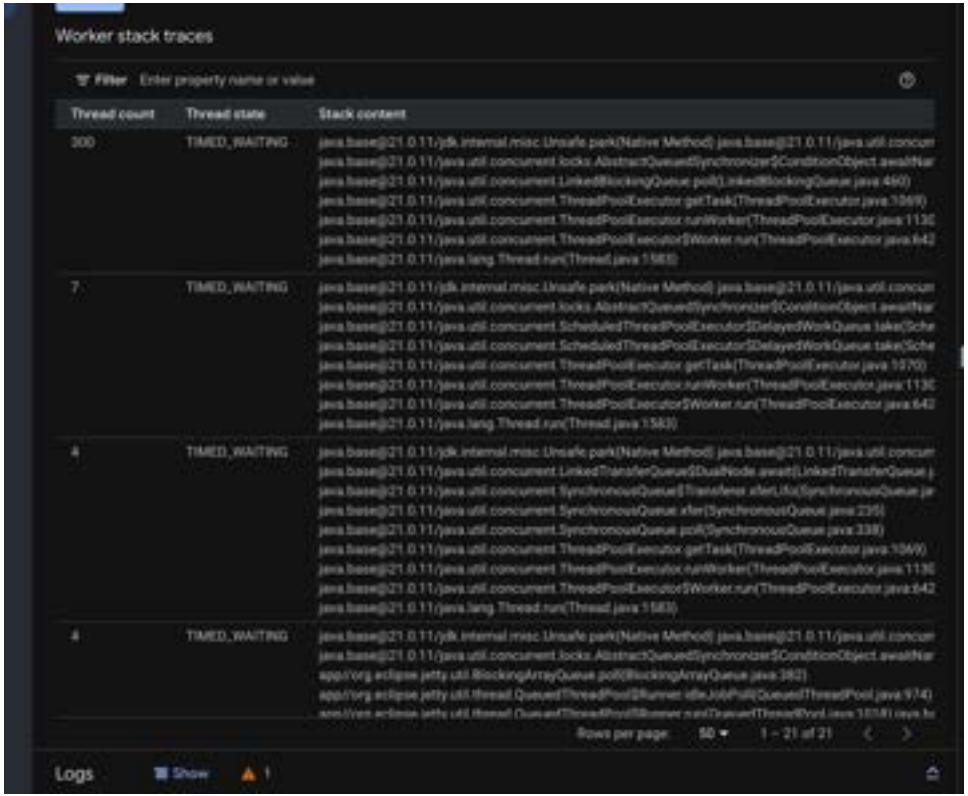
Observability: Advanced Debugging



Observability: Advanced Debugging



Observability: Advanced Debugging



Hot Key Detection

New for this year: Opt-in logging of unencoded Hot Keys

- The Problem:
  - A Key is the fundamental unit of parallelism in Dataflow. As a consequence, processing for a single key is single threaded.
  - This means that if one key is overloaded compared to the others, the pipeline may fall behind since we are not able to parallelize efficiently.

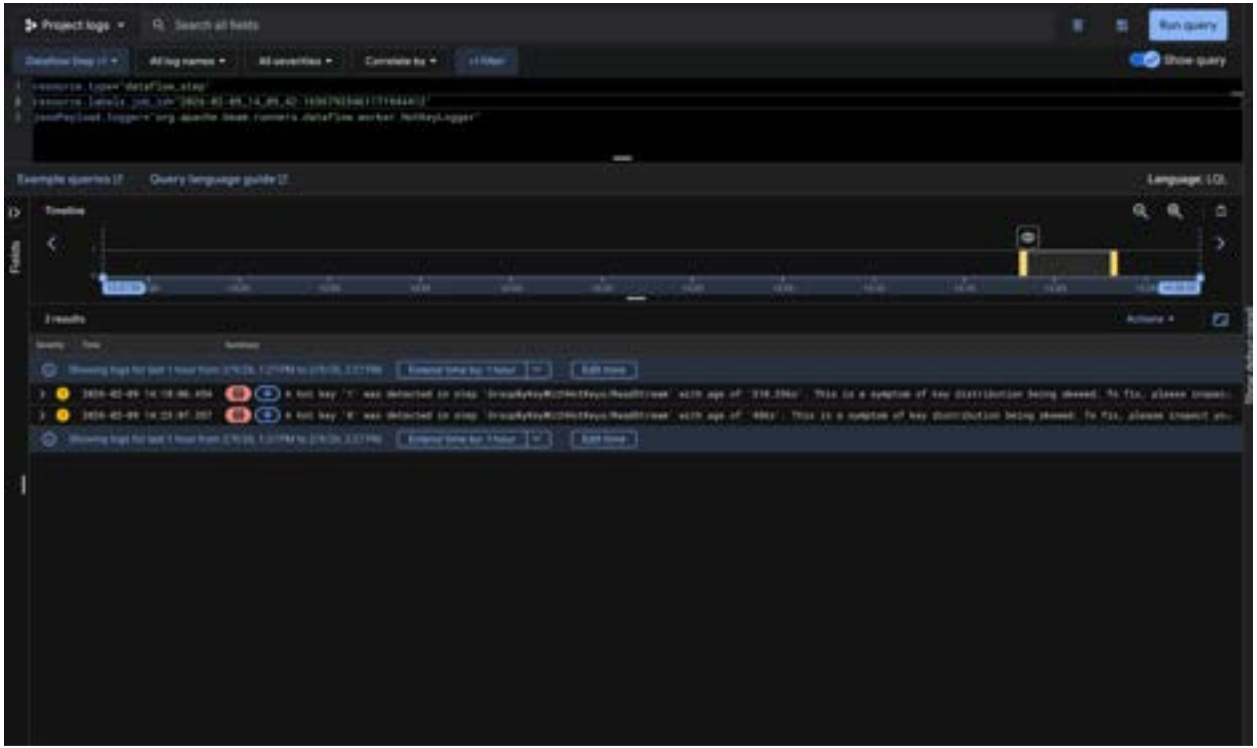
Opt-in Logging of Hot Keys

- The way to fix hot keys is normally by either updating pipeline code or by updating upstream data sources but the challenge had been figuring out what the Hot Key is.
- By default, Dataflow shows a hashed version of the key.
- Now, opt a job into Hot-Key Logging with

```
--hotKeyLoggingEnabled=true
```

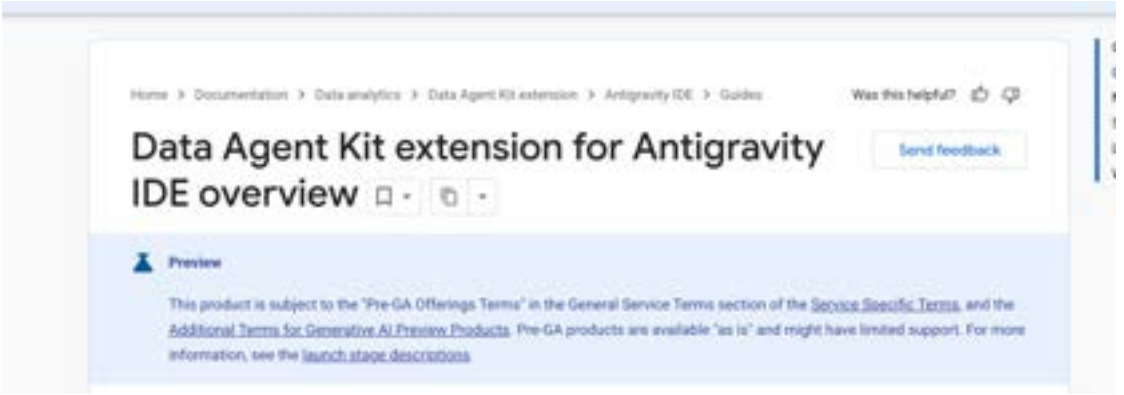
Hot Key Logging

Hot-Key Logging



AI Pipeline Writing/Debugging

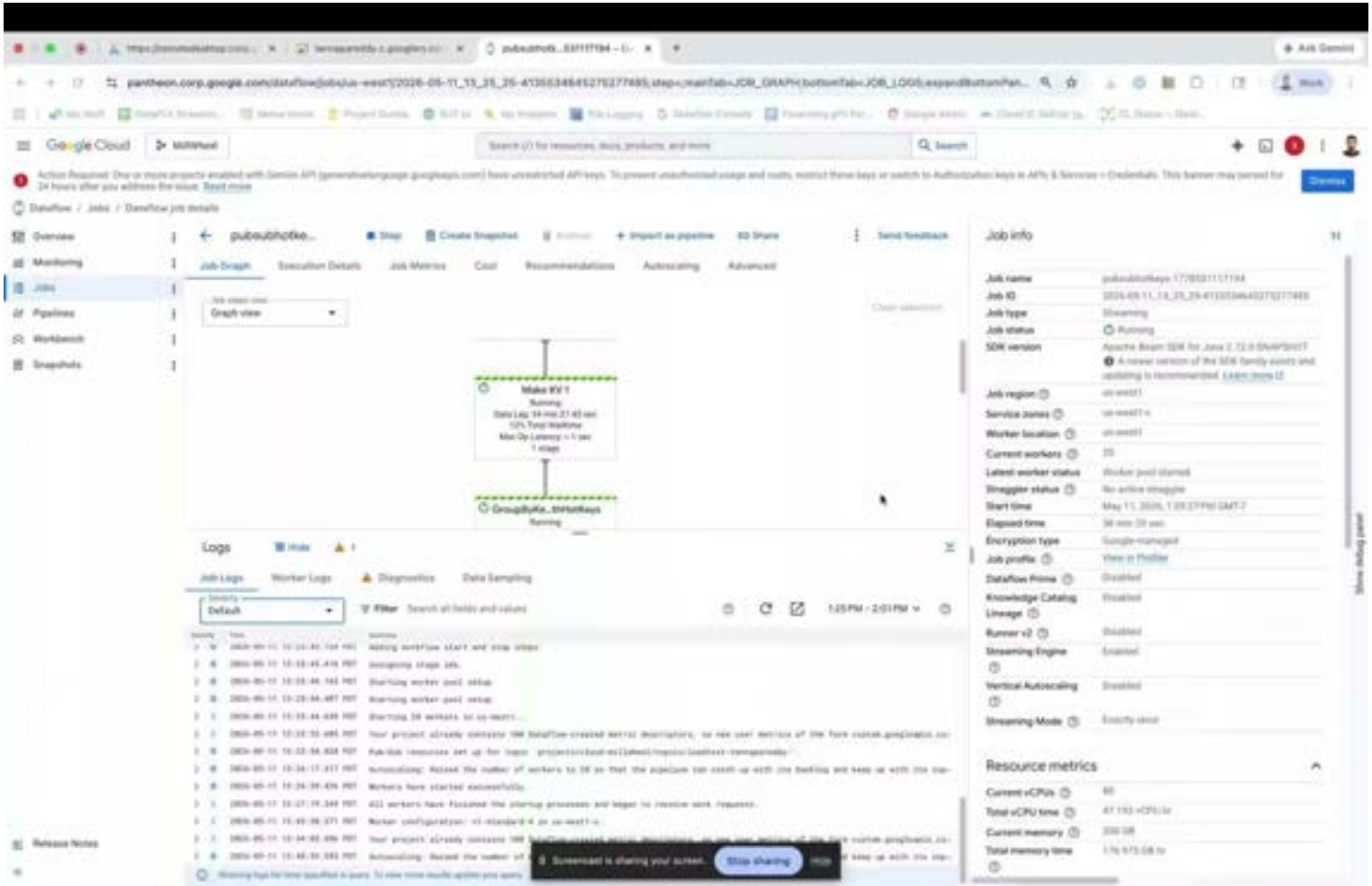
What is it?



- Data Agent Kit extension
- Brand new way to write and monitor your pipelines
- Get the wisdom of the whole team at your fingertips
- Programmed to intelligently traverse through documentation and Pantheon signals
- Built for Antigravity

Demo time

- Example use case of asking Antigravity to help debug a poorly performing pipeline.


Google Cloud
Job info
Job graph
Execution details
Job metrics
Cost
Recommendations
Autoscaling
Advanced
Logs
Job logs
Worker logs
Diagnostics
Data sampling
Resource metrics
Screencast is sharing your screen.
Stop sharing
Hide

Thank you!

Questions? Feel free to reach out!

linkedin.com/in/tomstepp
linkedin.com/in/ryan-wigg