Cost optimization of Dataflow pipelines

Sergei Lilichenko

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Intro

- Solutions Architect at Google Cloud
  - I help Google Cloud's customer design and troubleshoot Data Analytics workloads
  - I publish blogs and architecture guides
- Sometimes contribute to the Apache Beam repo
- Was an enterprise architect and tech lead (FSIs, startups) in my prior life

BEAM SUMMIT

A bit about this session

- A lot of different technical topics are covered
- You will not remember everything; just make mental notes
- Slides are pretty dense, but for a reason - use them as reference material when you need to
- Remember - some Beam pipelines are the most complex pieces of software a typical company creates. Tuning cost and performance of these pipelines can take knowledge and practice.

BEAM SUMMIT

Agenda

- Systematic approach to cost optimization
- Dataflow cost structure, cost prediction and monitoring, optimization goals
- Batch and streaming pipelines optimization
- Performance optimization
- Keeping the big picture in mind
- Q and A

BEAM SUMMIT

Systematic Approach to Cost Optimization

- Understand Dataflow cost components
- Predict cost for new pipelines
- Monitor cost of production pipelines
- Define thresholds/budgets for pipelines and issue alerts when these thresholds are exceeded
- On a regular basis (monthly)
  - Identify pipelines exceeding budget/highest consumers
  - Optimize these pipelines
  - Upgrade Beam SDK and libraries used by the pipelines
- On a regular basis (quarterly) evaluate overall architecture and see what can be optimized
  - Check for new Dataflow features (e.g., auto sharding, new connector methods, accelerators)
  - Check for new products/features to improve/supplement/replace Dataflow pipelines

BEAM SUMMIT

Dataflow costs

BEAM SUMMIT

Understanding Dataflow Cost Structure

- Direct Dataflow costs. See the Dataflow Pricing page for details
  - “Classic” Dataflow
    - Cost of worker VMs (CPU, memory, GPU, disk)
    - Cost of Dataflow Shuffle (batch) or Streaming Engine (streaming)
  - Dataflow Prime
    - DCUs
  - Snapshots
- Indirect pipeline costs
  - Usage of other APIs invoked from the pipeline
    - BigQuery Storage Read/Write APIs
    - Google Cloud Storage APIs
    - BigQuery queries
    - Pub/Sub subscription consumption and publishing
  - Network egress, if any
  - …

BEAM SUMMIT

Predict Dataflow Costs

- Most of the time you have to run a pipeline at a small scale and extrapolate the expected cost of production pipeline
- Use the Cost tab on the Dataflow Job UI to get a quick estimate of a running pipeline
- Use this blog as an example of how cost prediction can be done
- Invest into creating pipeline's integration test harness. A very useful tool is the streaming data generation Dataflow template provided by Google. Working example.

BEAM SUMMIT

Monitor Dataflow Costs

- Enable Billing Export into BigQuery
- Create labeling taxonomy. Follow these best practices.
- Add labels to Dataflow jobs. Flex templates can have pre-set labels and validate that labels are provided.
- For critical pipelines, create monitoring alerts to do real-time notifications and cost control.
- Direct Dataflow cost analysis
  - Can be done directly on the Billing Export tables
- Indirect pipeline cost analysis
  - Harder to do and it depends on the APIs
  - BigQueryIO - load and query jobs run by the pipeline have particular labels
  - Always use a custom Service Account to run Dataflow jobs for security reasons. Side effect - it helps to identify API usages.

BEAM SUMMIT

Cost and Performance Optimization Goals

- The pipeline meets the required SLA in performance and load tests
- The pipeline uses the smallest worker VM machine shapes with the least amount of disk space
- The CPU utilization is consistently around 80-90% after a reasonable amount of performance tuning has been done. See scaling algorithm for details.
- Streaming pipeline properly scales up and down according to the volume of streaming data or stages in batch pipelines. Data freshness metric is a good indicator.

BEAM SUMMIT

Cost Factors

Three main factors that affect cost - pipeline “run mode” defined at the pipeline start, actual resources consumed by the pipeline (“pipeline performance”), and pipeline design dictated by SLAs.

Pipeline run mode:

- Streaming vs batch
  - Streaming engine, Shuffle Service, FlexRS
- Machine type, disk size, GPUs
- Autoscaling model
- Initial number of workers and maximum number of workers
- Prime vs Traditional, the latter with Classic Runner or Runner v2

Pipeline performance:

- SDK (Java, Go, Python)
- Sufficient parallelism
- Efficient custom transforms
- Efficient IO connectors, data location
- Efficient coders

BEAM SUMMIT

Cost Factors (cont.)

Pipeline SLAs and throughput requirements:

- Low end-to-end ingest latency requirement can affect the pipeline design
- Extremely high throughput requirements may require more expensive connectors
- Requirements to process late arriving data can affect the overall cost
- Requirements to be able to process streaming data spikes may require extra capacity

All these factors need to be considered together when optimizing the pipeline costs.

BEAM SUMMIT

Batch Pipeline Run-Mode Optimization

- Use FlexRS whenever possible. It provides around 40% savings for pipelines which tolerate pipeline start delays.
- Make sure Dataflow Shuffle is enabled and reduce the size of the persistent disk to save costs
- Try running with the smallest machine types. If needed, tune the machine memory as described in this section.
- By default, batch pipelines run in autoscaling mode. Disable autoscaling if you suspect that it's not beneficial, for short-running pipelines, or for the pipelines for which the timing is not critical.
- Specify the number of workers for the pipeline
  - *numWorkers* determines the initial number of workers. "Guestimate" the number of workers needed in the initial stage of the pipeline. It will save cost by avoiding the work redistribution required during scaling up.
  - *maxNumberOfWorkers* determines the maximum number of workers a pipeline can create. Use this parameter to cap the potential spend. Start with a small number when you first test the pipeline and increase it to the maximum number required to process a production load.

BEAM SUMMIT

Batch Pipeline Run-Mode Optimization (cont.)

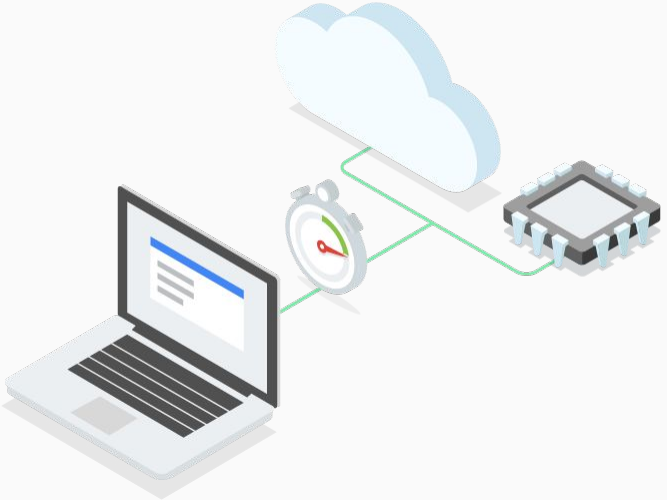
- Avoid running pipelines which process very small amount of data quickly. If possible, run fewer pipelines on larger datasets. There is cost of starting and shutting down worker VMs during which no data processing occurs. Streaming pipelines can be cost effective in these cases.
- Consider GPUs for the workload which can benefit from them.
- Consider specialized machine types for memory or compute intensive workloads.
    - Use CoreMark scores to compare CPU performance
- Some pipeline will benefit from changes to the default number of threads per worker. For streaming pipeline it's currently in 300-500 range per VM in Java, 12 in Python, for batch pipelines - 1 per vCPU.
    - Enable Dynamic Thread Scaling to let the pipeline tune the thread count per VM based on CPU utilization.
- Automatically stop pipelines exceeding max expected duration - use "max_workflow_runtime_walltime_seconds" service option.

BEAM SUMMIT

Flexible Resource Scheduling

Reduce cost of your batch processing jobs using advanced scheduling techniques, Dataflow Shuffle, and preemptible VMs

- Pools multiple VM types for stockout protection
- Stores state in Shuffle to handle preemptions
- Run your weekly & daily jobs or onboard your historical data at 40% discount for workers

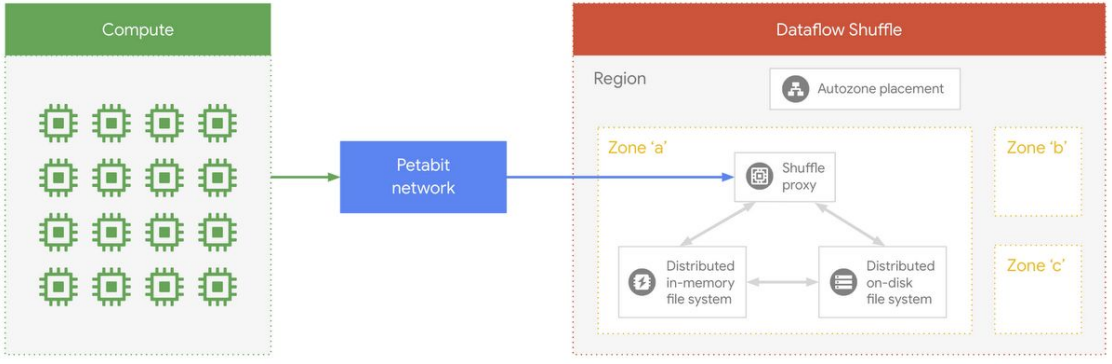


Key Takeaway: Identify latency-tolerant jobs and enable FlexRS by passing in flag

Dataflow Shuffle

Distributed, in-memory Shuffle service used for shuffling data (similar to BigQuery)

- Faster execution time for the majority of pipeline types
- Reduction in consumed CPU, memory, and PD
- Better autoscaling (VMs do not hold any more shuffle data)



Key Takeaway: Test jobs with large amounts of data (>300 GB) to shuffle and pick the right combination of resources that will complete your job (lesser PD).

Streaming Pipeline Resource Optimization - General

- Make sure Streaming Engine is enabled and reduce the size of PDs (30Gb) to save costs
  - Another option is to use Streaming Appliance, but it's not recommended
- Consider using Dataflow CUDs
- Enable resource-based billing for streaming engine to optimize costs
  - Cost savings will depend on the pipeline's structure, expected to be more cost effective for the majority of the pipelines.
- Try running with the default/smallest machine types before using larger machine shapes. If needed, tune the machine memory as described in this section.

BEAM SUMMIT

Streaming Pipeline Resource Optimization - Runners

- Consider whether the pipeline will benefit from using the default runner or Runner v2
  - Runner v2 is the only runner available for Python and Go SDKs
  - Java SDK:
    - Some pipelines require Runner V2 (multi-language pipelines, Spanner & Bigtable change stream IOs).
    - Some pipelines require the default runner, esp. If they use certain Beam States.
    - The default runner is typically faster than Runner v2
- Consider whether the pipeline will benefit from running in the "at-least-once-mode". New feature (February 2024)
  - Some sinks have "at-least-once" semantics - BigQueryIO's STORAGE_API_AT_LEAST_ONCE method, JdbcIO

BEAM SUMMIT

Streaming Pipeline Resource Optimization - Autoscaling

- Specify the number of workers for the pipeline
  - *numWorkers* option determines the initial number of workers. If you are processing a large backlog of the messages start with high number if you need to process the backlog quickly. It will save cost by avoiding the work redistribution required during scaling up.
  - *maxNumberOfWorkers* option determines the maximum number of workers a pipeline can create. Use this parameter to cap the potential spend. Current default limit is 100 workers.
  - *min_num_workers* experiment (not documented) forces the lower boundary
- "Overwrite" the default autoscaling (allows changing min/max number of workers without pipeline update) by using the lightweight update API (blog)
- Use autoscaling hints (target CPU utilization) to further tune autoscaling
- Disable autoscaling if you suspect that it's not beneficial, or for the pipelines for which the timing is not critical.

BEAM SUMMIT

Streaming Pipeline Resource Optimization - JVM Configuration

- Pipelines with memory-intensive transforms can override the default garbage collection
  - Use “--experiment=java_gc=UseZGC” option
  - See more details at https://docs.oracle.com/en/java/javase/11/gctuning/z-garbage-collector1.html
- CPU-intensive pipelines can benefit from a worker machine type different than the default
  - Some customers reported that “t2d” machine types reduced their costs by over 30%
- Reducing the number of threads per VM can be useful if the smaller machines have very high CPU utilization and show very high memory usage.
  - Use “numberOfWorkerHarnessThreads” pipeline option.

BEAM SUMMIT

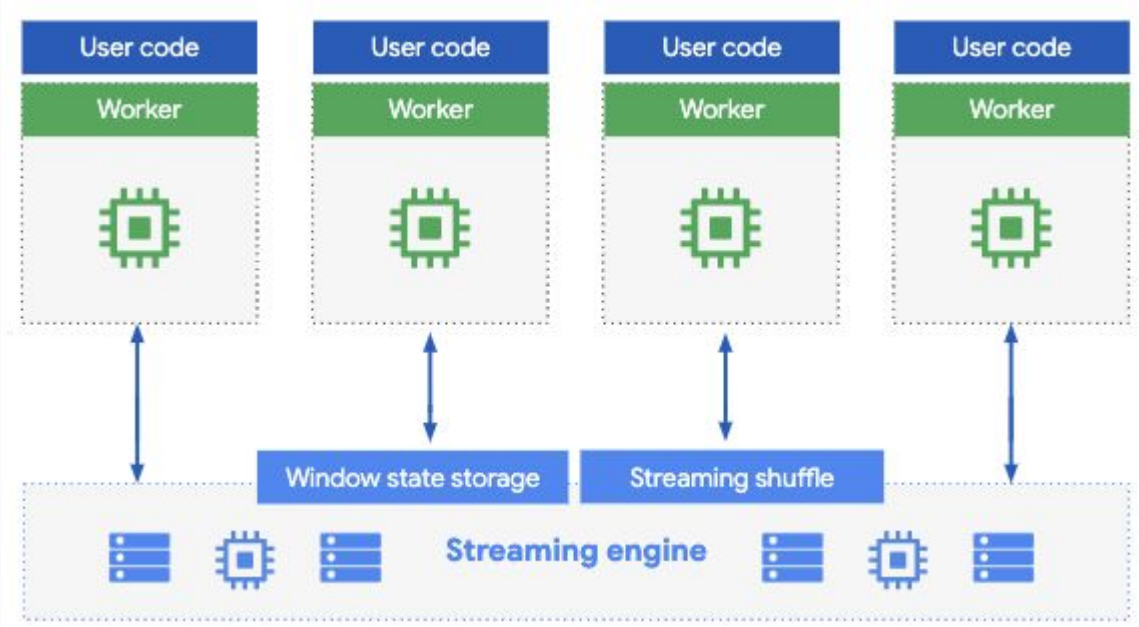
Streaming Engine

Offloads window state storage from PDs attached to workers to a backend service

- Smoother autoscaling
- Better supportability
- Less worker resources

Works best with n1-standard-2 machine types (instead of n1-standard-4 for regular streaming)

Detailed blog is highly recommended to understand details



Key Takeaway: Test streaming jobs with Streaming Engine and find the minimum resource footprint to execute

Pipeline Performance

Pipeline Performance Optimization

Faster pipelines usually means cheaper pipelines. But it's not always the case because it can mean that by using faster but costlier external APIs the cost has shifted from the direct Dataflow costs to indirect costs. See IO Specific Recommendations for some examples.

Make sure you understand Beam and Dataflow processing model. Critical parts:

- PCollections, PTransforms, DoFns
- If using streaming pipelines - Windows, Watermarks and different triggering strategies
- Execution model - multiple worker VMs, each worker executes multiple worker threads, bundles of elements from a PCollection get distributed to each worker thread.
- Bundle processing lifecycle. Understand that exactly-once processing doesn't mean that the bundle will be processed only once!
- Dataflow fusion optimization, error handling, autoscaling

BEAM SUMMIT

Pipeline Performance Optimization (cont.)

Address the low hanging fruit first

- Does the pipeline run in the region where the resources are located?
- Does it process regional resources?
- Does it use the temporary Cloud Storage bucket which is regional?
- Does it run into any quota issues - both Dataflow specific and sources and sink specific?
- Does it have enough network bandwidth to access the data?
- Does the performance test reflect realistic production environment?
- Have you accounted the non-processing time (start/shutdown/autoscaling) in your calculations?
- Have you reviewed if your case matches one of the documented common errors?

Tips:

- Run pipelines in dedicated projects
- Create a dedicated Dataflow worker service account per pipeline
- Test streaming pipelines in two modes - steady state (normal) and large-backlog-processing ("crash recovery")

BEAM SUMMIT

Pipeline Performance Optimization (cont.)

Invest time in creating tests for your pipelines

Good thing to have in general. Helps with SDK upgrades, pipeline refactoring, code reviews, etc.

Unit tests:

- Testing data pipelines can be hard, but Beam has built-in support for unit testing. It also supports testing complex streaming pipelines by simulating time changes.
- Many optimizations, especially reworking custom CPU intensive transforms, can be done directly on developers' workstations without the need to run a single Dataflow pipeline.

Integration tests:

- There is no substitute for end-to-end integration tests; some issues (usually IO related) can only be detected when running using Dataflow runner
- Can use these tests for early cost estimation

Performance tests:

- For large scale pipelines invest in the test generator that gives you a realistic data "shape" - number of events (batch), number of events per second (streaming), event size, number of keys, etc.

GitHub repo with a good example of different types of tests.

BEAM SUMMIT

Pipeline Performance Optimization (cont.)

Whenever possible, start with upgrading Beam SDK

- Performance improvements are constant; it's likely the issue you are facing is already resolved
- Read release notes - they can give you an idea of what changed and what new features might be available.

Identify where to focus the optimization efforts

- Use Dataflow UI to identify the problem areas
- Use Dataflow metrics to get details not covered by the UI. Consider new DoFn specific ones added in May 2024.
- For CPU intensive pipelines, profile the pipeline to identify the hot spots
- Use custom metrics, especially Distribution, to capture performance metrics
- Use data sampling in pre-production to identify data related issues
- Do *not* attempt to use logging as a way to capture per-element processing metrics for high volume pipelines - logging is subject to limits and excessive logging will cause performance degradation and will skew the performance results.
    - It is sometimes useful to turn on detailed logging for particular IO transforms to understand when and how many API clients are created.

BEAM SUMMIT

Pipeline Performance - Excessive Data Shuffles

Shuffle happens when GroupByKey, CoGroupByKey, or Stateful DoFns are used.

Symptoms:

- Too much data is shuffled (“Total streaming data processed” resource metrics)

Causes:

- Unnecessary data gets into the shuffle phase
- Inefficient coders are used

Solutions:

- **Filter out as much data/remove unused attributes as possible as early as possible.**
- Use efficient coders
- Avoid shuffle if possible. But in many cases it’s a required step to get exactly-once-processing semantics
- Can you use Combiners instead of GroupByKey? They are typically much more efficient.

BEAM SUMMIT

Pipeline Performance - Limited Parallelism

Symptoms:

- Slow processing of GroupByKey transform, warnings of the presence of the hot keys in the UI console

Cause:

- Very few keys or very skewed keys in one of the KV PCollections

Solutions:

- Redesign the key partitioning if possible
- Use Combiners and several variations of ‘withHotKeyFanout’ transforms. See this blog for details

BEAM SUMMIT

Pipeline Performance - Overly Aggressive Dataflow Fusion Optimization

Symptoms:

- Execution graph shows much larger number of elements in the output collection than in the input one.

Causes:

- Several transforms fused into a single stage where a transform in the beginning or middle of the stage causes a high fan-out of the elements

Solutions:

- Prevent fusion by reshuffling the data. Caution - it involves additional shuffle, which comes with its own cost.
- Consider implementing a Splittable DoFn in place of the regular transform which causes the high fanout.

BEAM SUMMIT

Pipeline Performance - Expensive Per Element Calculations

Symptoms:

- Excessive CPU consumption by pipeline, very slow execution of non-IO transforms
- Execution tab show large “wall time” for a particular transform/ParDo
- Profiling identifies the @ProcessElement method that consumes excessive CPU

Cause:

- Expensive initialization of libraries/API clients/etc., for each element in PCollection
  - Example: Json library usage described in this blog.
- Inefficient calculations.
  - Example: parsing source payload every time a data element needs to be extracted from the payload.

Solution:

- Use the DoFn lifecycle methods to initialize and clean up the objects which can be safely used for all the elements in the bundle
- Optimize the code, e.g., parse JSon once and transform payload into a strongly typed class

BEAM SUMMIT

Pipeline Performance - Inefficient Coders

Symptoms:

- High volume of shuffled bytes, slow performance

Cause:

- Coders used for PCollections *that are shuffled* are sub-optimal

Solution:

- Use some of the high performance coders - Beam Schema-based coder, ProtoCoder, AvroCoder
- Avoid SerializableCoder
- Create custom coder for large PCollections that would benefit from unique optimizations.
- Try using compression-enabled coders, e.g. ZstdCoder

BEAM SUMMIT

Pipeline Performance - Batch External API Calls

Symptoms:

- External API quotas are quickly reached
- Transforms which include these calls are slow
- External API charges are higher than they can be. API supports micro-batch processing which is cheaper per element than processing individual requests, but the pipeline incurs per-element cost.

Causes:

- Calling an API involves expensive initialization and is done per element
- Calls are done per element and are not batched

Solution:

- Use DoFn's lifecycle methods (annotated with @Setup, @StartBundle) to initialize the API client
- Collect the API requests in the @ProcessElement method and call the API in the @FinishBundle method.
- Or use GroupIntoBatches transform to create batches of certain sizes
- Make sure that the calls are idempotent - bundles can be replayed!

BEAM SUMMIT

IO Specific Recommendations

- Whenever possible, make sure that the GCS and other data source are regional and co-located with the region where the pipeline runs.
  - BigQueryIO can be an exception when Storage Read and Write APIs are used at scale due to much larger quotas in multi-region locations.
- BigQueryIO writes
  - Enable autosharding for the methods that support it.
  - Use the FILE_LOADS method for the pipelines where ingest latency is not critical. This method can be also used in streaming pipelines.
    - By default writes TableRows as Json, make sure to use Avro for faster writes and better load performance.
  - Otherwise - use one of the Storage Write API methods
    - Exactly-once semantics is more expensive than at-least-once
    - Consider if CDC-writes using at-least-once method is an option
- BigQueryIO reads
  - Use the EXPORT method for the pipelines where processing latency is not critical.

BEAM SUMMIT

IO Specific Recommendations (cont.)

- FileIO reads, including Cloud Storage
  - Reading compressed data can be very slow, depending on the compression type. Zip-based compression doesn't allow parallel reads and forces the sequential processing of files. Avro compressed files are typically the most efficient way to read data from object stores.
  - Reading a large number of many small files can affect overall performance.
- FileIO writes, including Cloud Storage
  - Number of shards used to write the data directly affect the parallelism and speed of the writes

BEAM SUMMIT

Language-dependent Optimization

Some Python pipelines can benefit from using cross-language pipeline transforms implemented Java:

- High performance IOs (Kafka, BigQueryIO using Storage Write API, JDBC)
- BeamSQL
- Custom written transforms

Python pipelines can benefit from using custom containers with pre-packaged dependencies. It decreases the worker start up time.

Consider using DataFrame API to take advantage of the high performance optimizations in the underlying pandas implementation.

BEAM SUMMIT

Advanced Optimizations

- For complex multi-transform pipelines consider replacing several transforms with a single BeamSQL transform. The auto-generated graph can be more efficient than the hand created one.
- Assess how complex joins are done and consider creating custom transform to handle them. See how Spotify improved their processing (video)
- All the Apache Beam is open source - customers can learn a lot from seeing how many core transforms are implemented

BEAM SUMMIT

Don’t Forget the Big Picture

BEAM SUMMIT

Periodically Reassess Overall Architecture

- Cheapest pipeline is no pipeline at all
  - Can I avoid the “T” in ETL altogether by changing the input and doing direct ingest into the sink using native load tools (e.g., BigQuery load)?
  - Can I do the “T” during the load by native tools (“insert into bigquery_native_table from (select transformed_columns from external_table)”)?
  - Are there new native ways to ingest (e.g., Pub/Sub BigQuery subscriptions)?
- Do I truly need to have data processes in streaming mode or can it be done in batches?
  - Streaming compute is more expensive than batch
  - But don’t rush to change before getting the streaming pipeline to properly autoscale, the change can be negligible
- Am I using the right SDK for the job? Consider multi-language pipelines to use unique features offered by different SDKs (high performance IOs in Java and RunInference in Python)
- Can I use the turnkey transforms to simplify my pipelines?

BEAM SUMMIT

Key Takeaways

- Invest in monitoring, alerting, cost analysis
- Learn how to estimate the cost of new pipelines
- Invest in understanding Beam model and learn the best practices
- Learn how to spot performance problems
- Create unit and integration tests, use code reviews
- Don’t lose the big picture and reassess your assumptions and architecture periodically
- Think “iterations” rather than “big bang”

BEAM SUMMIT

Further reading

Useful content besides the official documentation

- Writing Dataflow pipelines with scalability in mind blog
- A number of very good sessions on various topics at Beam Summits ('22 sessions)
  - Dataflow cost optimization at Orange
  - Avro Serialization and Deserialization
  - Common Customer Issues of Running Beam on Google Cloud
  - Relational Beam
- Dataflow cost optimization blog

BEAM SUMMIT

Thank you!

Reach out if you have questions:

slilichenko@google.com

https://www.linkedin.com/in/sergei-lilichenko

BEAM SUMMIT