Building Fully Managed Service
for Beam Jobs with Flink on Kubernetes

Rishabh Kedia / Talat Uyarer
Streaming Team
Palo Alto Networks

BEAM SUMMIT
NYC 2023

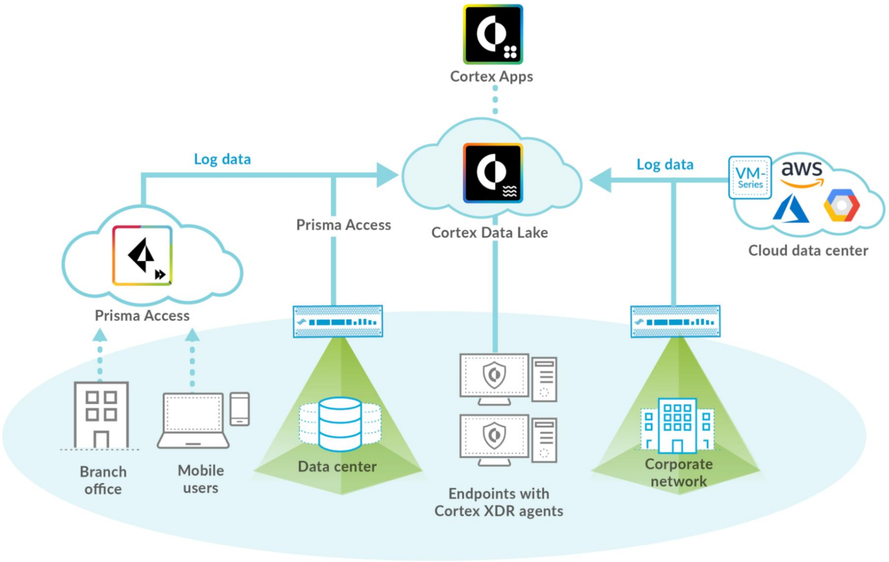
Agenda

- What is Cortex Data Lake
  - Streaming Service Overall Design
- Design of Kubernetes Streaming Infrastructure
  - How we run Beam Flink Jobs on Kubernetes
- What kind of Challenges that we faced
  - Checkpoint Challenges
  - Scale Challenges
- Autoscaling to use resources efficiently

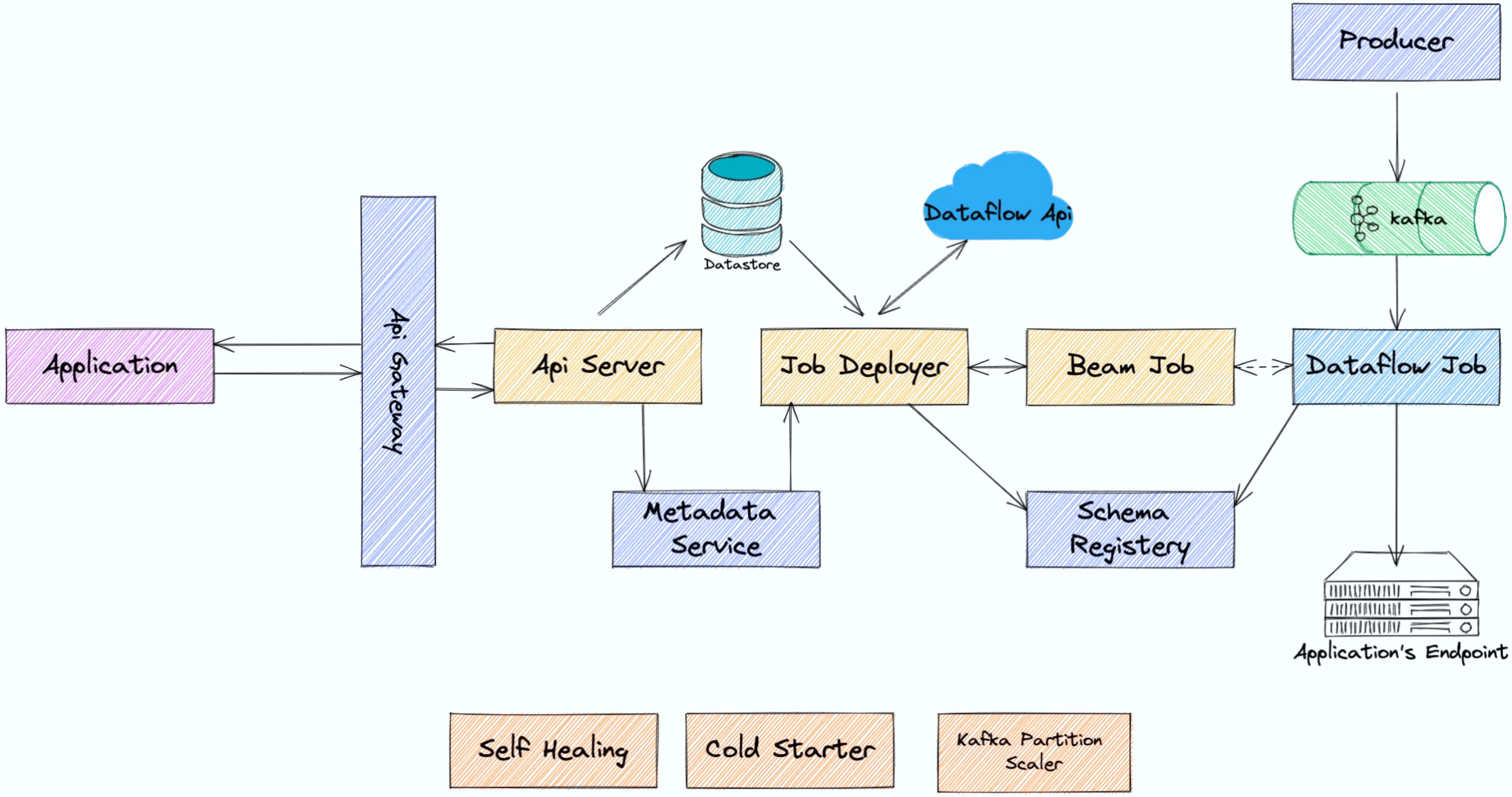
What is Cortex Data Lake

What is Cortex Data Lake

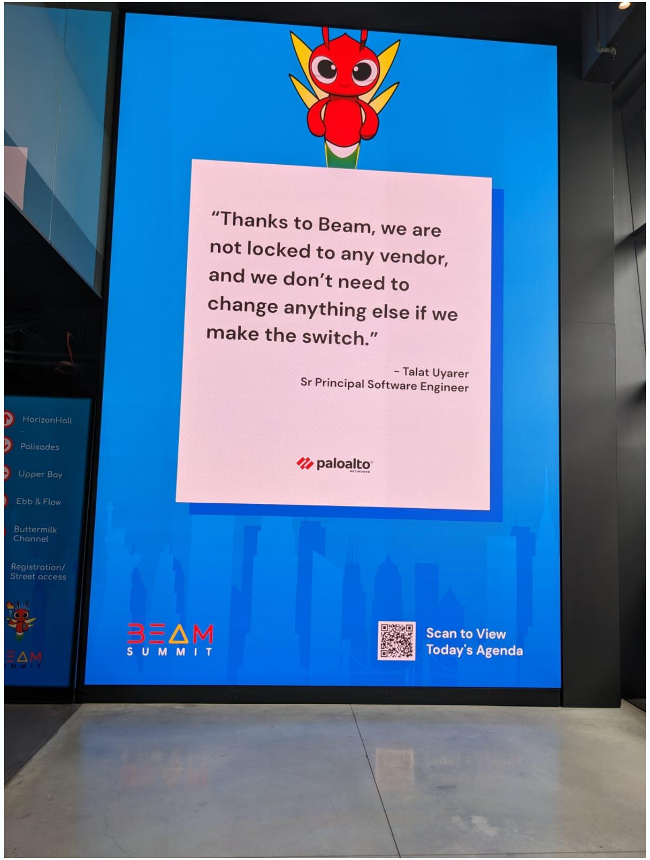
- PANW provides cloud-based, centralized log storage and aggregation for any kind of firewalls
- One of our locations receives more than **20+ million records per second** and can be scaled to receive more than 100 million records per second.
- We serve more than 10 different applications with **20 thousands streaming jobs** in 10+ geographical regions



Streaming Architecture at PANW



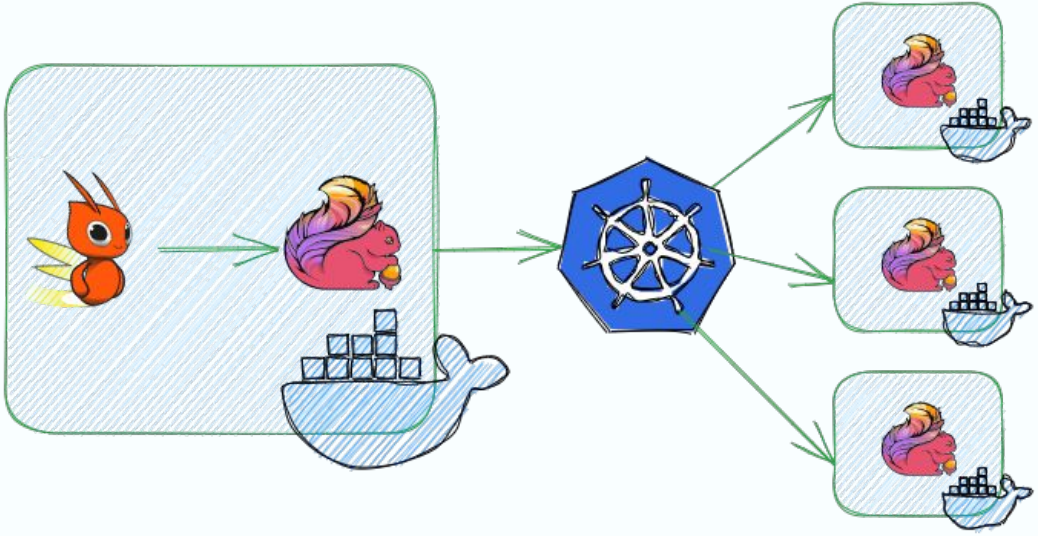
Streaming Architecture at PANW



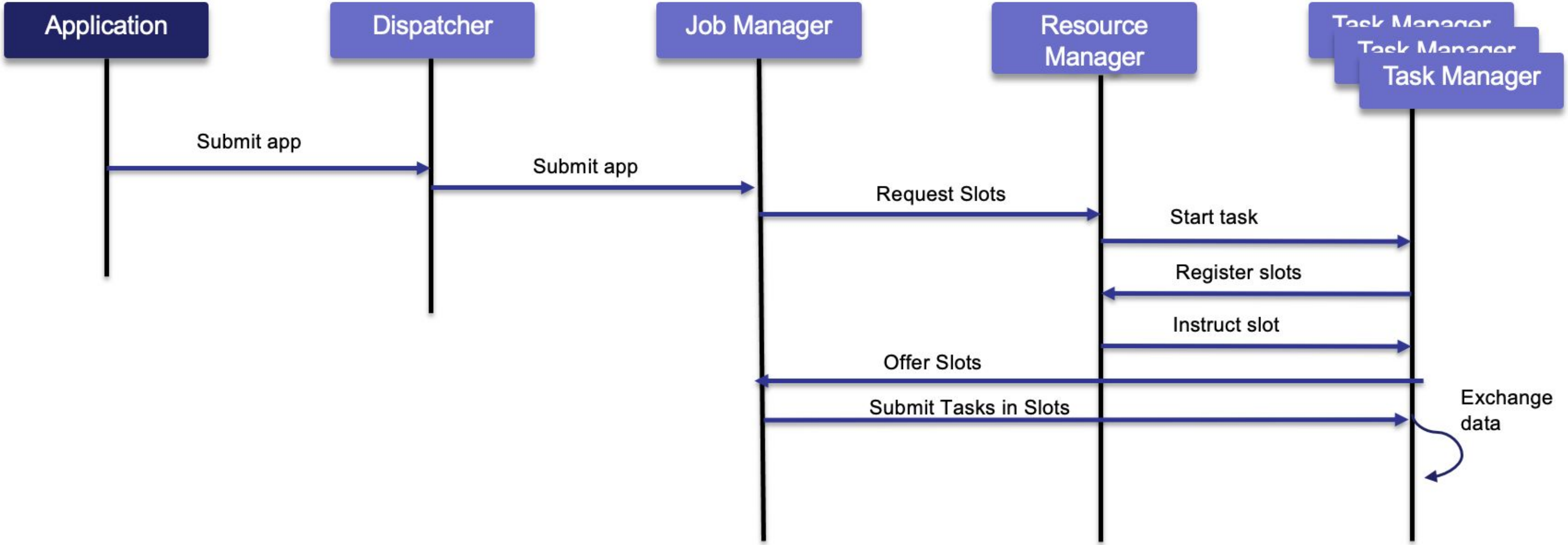
Kubernetes

Deploying a Beam Job on Kubernetes

1. Create a BEAM pipeline and compile into a Uber JAR.
2. Create a Docker image containing the JAR and Flink Configuration files
3. Create a Yaml to Deploy Kubernetes make entry point is flink-run command



Deploying a Beam Job on Kubernetes

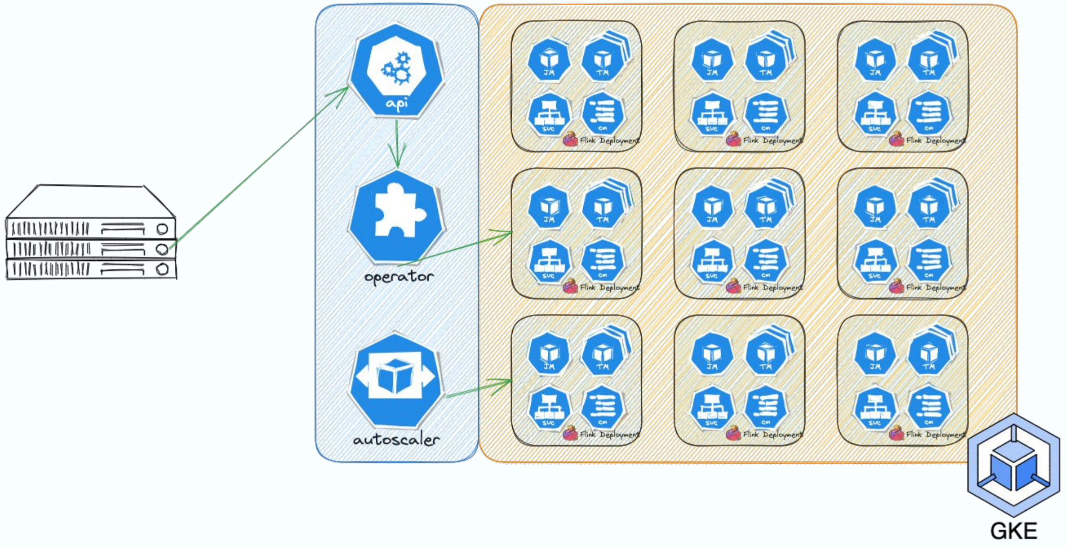


Source: https://jbcodeforce.github.io/flink-studies/architecture/

How About Deploying Thousand Jobs ?

- Creating deployments via a client is not feasible and makes integration difficult

- Flink introduced a **Kubernetes Operator** to handle all deployment needs via extending Kubernetes Api Server with Custom Resource Definition.



How to integrate in current production?

```yaml
apiVersion: flink.apache.org/v1beta1
kind: FlinkDeployment
metadata:
  name: basic-reactive-example
spec:
  image: flink:1.16
  flinkVersion: v1_16
  flinkConfiguration:
    scheduler-mode: REACTIVE
    taskmanager.numberOfTaskSlots: "2"
    state.savepoints.dir: file:///flink-data/savepoints
    state.checkpoints.dir: file:///flink-data/checkpoints
    high-availability: org.apache.flink.kubernetes.highavailability.KubernetesHaServicesFactory
    high-availability.storageDir: file:///flink-data/ha
  serviceAccount: flink
  jobManager:
    resource:
      memory: "2048m"
      cpu: 1
  taskManager:
    resource:
      memory: "2048m"
      cpu: 1
  podTemplate:
    spec:
      containers:
        - name: flink-main-container
          volumeMounts:
          - mountPath: /flink-data
            name: flink-volume
      volumes:
      - name: flink-volume
        hostPath:
          # directory location on host
          path: /tmp/flink
          # this field is optional
          type: Directory
  job:
    jarURI: local:///opt/flink/examples/streaming/StateMachineExample.jar
    parallelism: 2
```

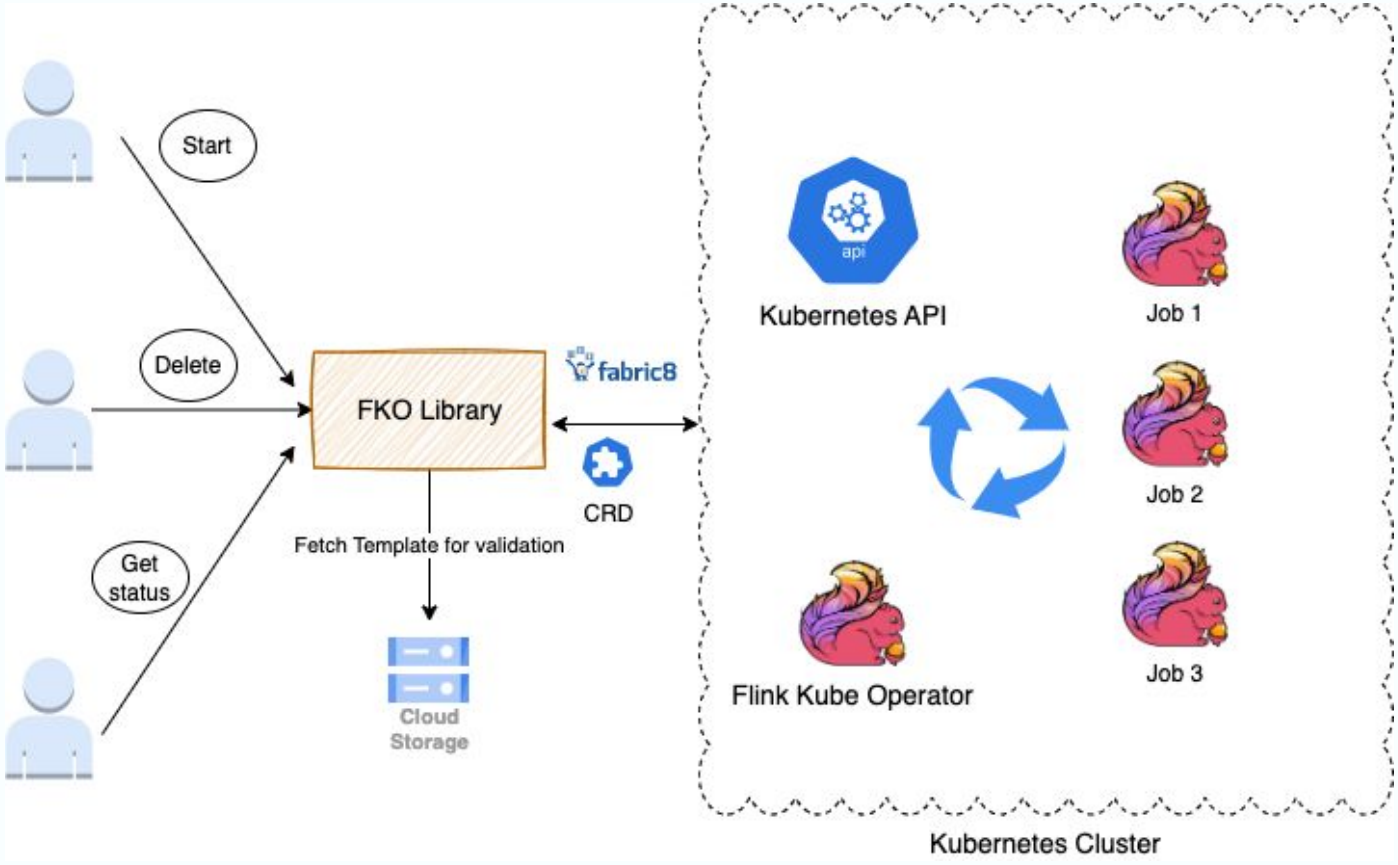
```
$  kubectl apply -f flink-job1.yaml
```

```
$  kubectl scale flinkdeployment flink-job1 --replicas=16
```

- Complex Kubernetes Interactions
- Inconsistent Flink Job Management
- Deployment Errors and Inefficiencies
- Barriers to Innovation

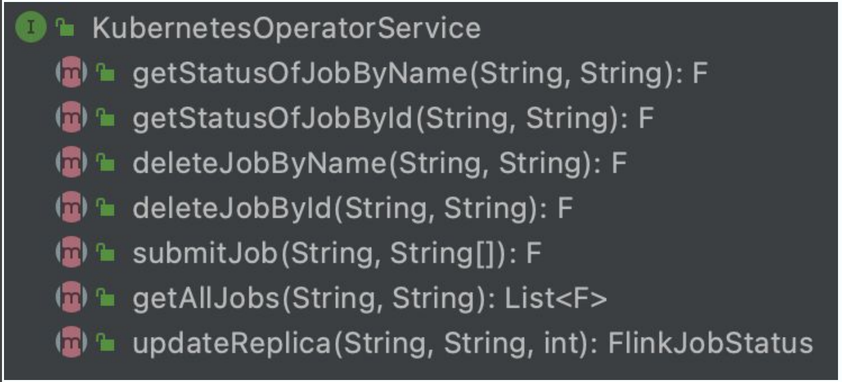
In addition to above items, our existing production service used Dataflow API. So how we can integrate Operator without an API support?

Solution: FKO Library



Benefits

- Authentication and Authorization
- Standardized Job Management
- Abstracted Kubernetes Complexity
- Easy Upgrades
- Effortless Deployment
- Empowering Data teams



Challenges

Checkpointing

Checkpointing is important to have healthy job. Let's calculate possible cost for our scale

- 20 K+ jobs
- 10 Seconds targeted checkpointing time
- 200 write calls (20000 / 10) per seconds per job
- ~ 20 Operator per Job
- (20 x 200) ~ 4000 write calls per second ($2 per second)

API or Feature	Class A Operations	Class B Operations	Free Operations
JSON API	storage.*.insert[1] storage.*.patch storage.*.update storage.*.setIamPolicy storage.buckets.list storage.buckets.lockRetentionPolicy storage.notifications.delete storage.objects.compose storage.objects.copy storage.objects.list storage.objects.rewrite storage.objects.watchAll storage.projects.hmacKeys.create storage.projects.hmacKeys.list storage.*AccessControls.delete	storage.*.get storage.*.getIamPolicy storage.*.testIamPermissions storage.*AccessControls.list storage.notifications.list Each object change notification[2]	storage.channels.stop storage.buckets.delete storage.objects.delete storage.projects.hmacKeys.delete

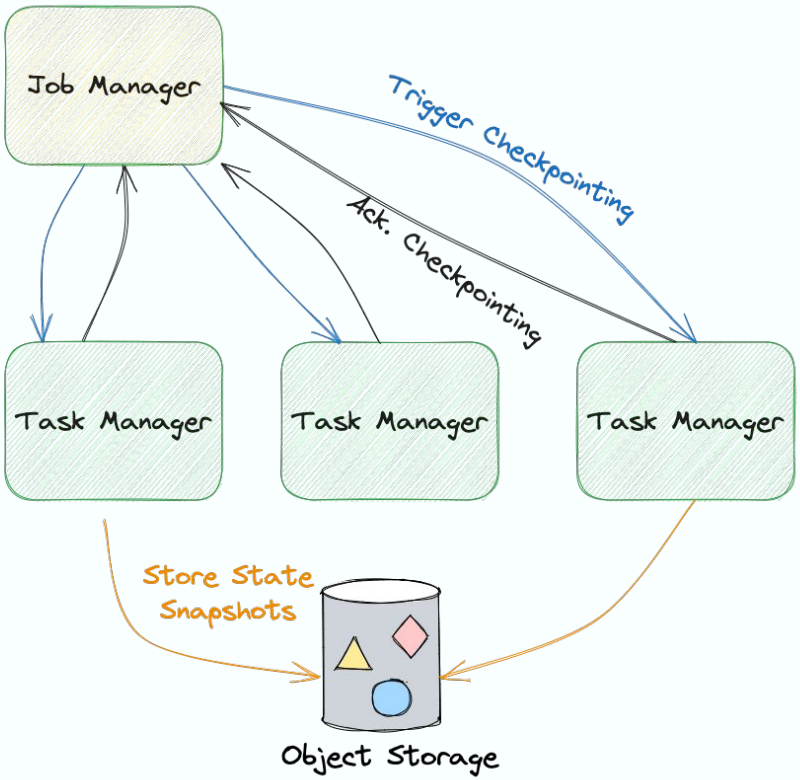
For buckets located in a single region:

Storage Class[1]	Class A operations (per 1,000 operations)	Class B operations (per 1,000 operations)	Free operations
Standard storage	$0.005	$0.0004	Free

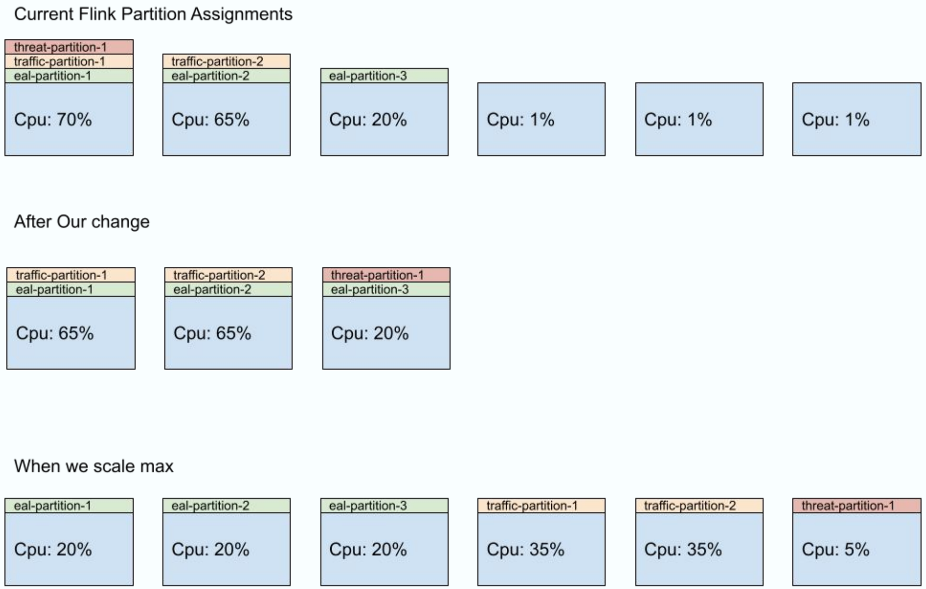
Source: https://cloud.google.com/storage/pricing

What we did for Checkpointing

- Reduce Checkpoint size by removing PipelineOptions
- Beam assign default parallelism for all independent pipelines without checking Kafka Partition Count.
- Define Memory Threshold to prevent creating so many small files (state.storage.fs.memory-threshold)
- Enable Unaligned Checkpointing



Flink Task Assignment Issue

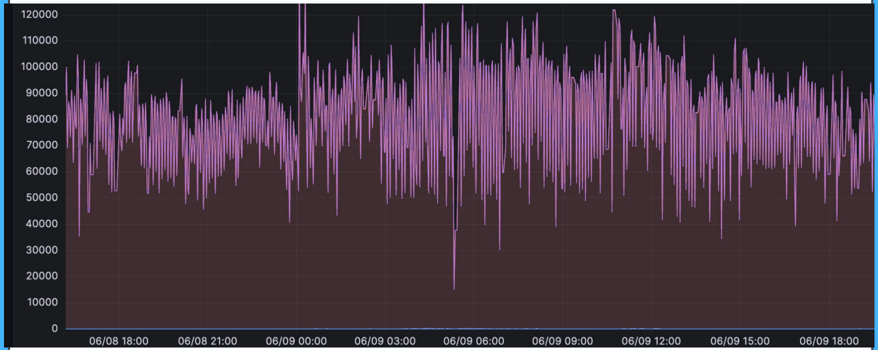


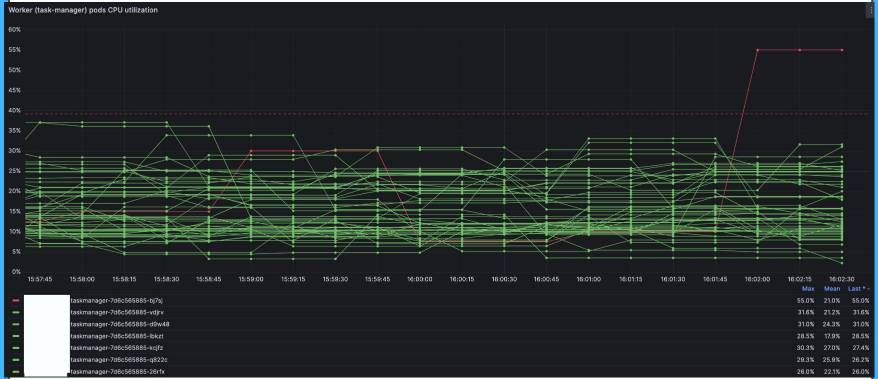
- Flink supports evenly distribution if you have one source.
- If you have multiple independent pipeline like us, Flink starts scheduling each partitions of source from zeroth pod/machine
- This makes first couple machines' load heavier than tail nodes.

Autoscaling

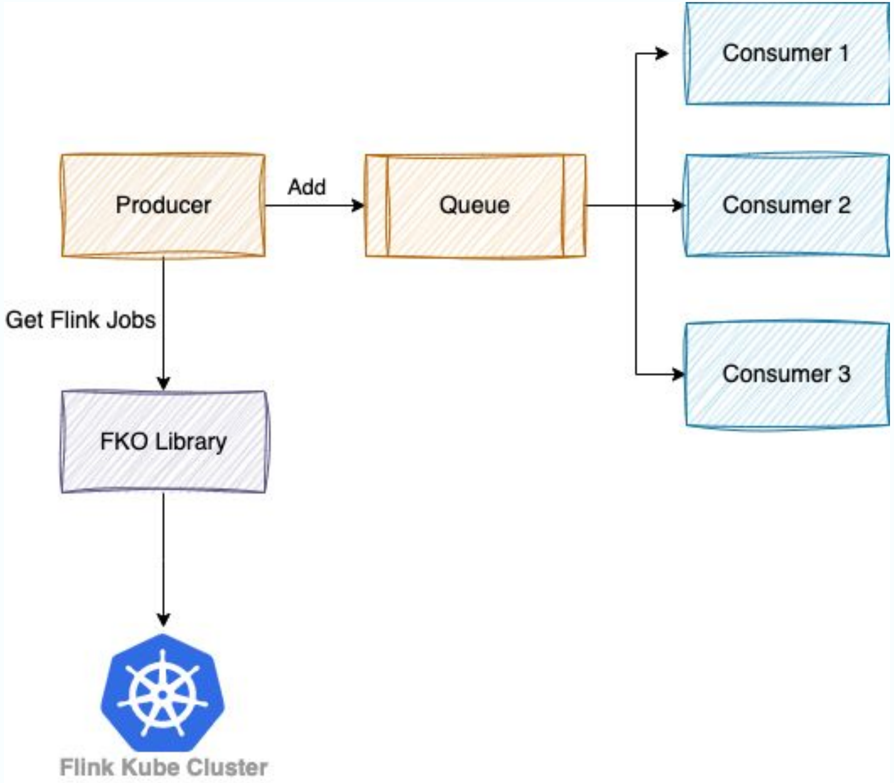
Problem: Scaling

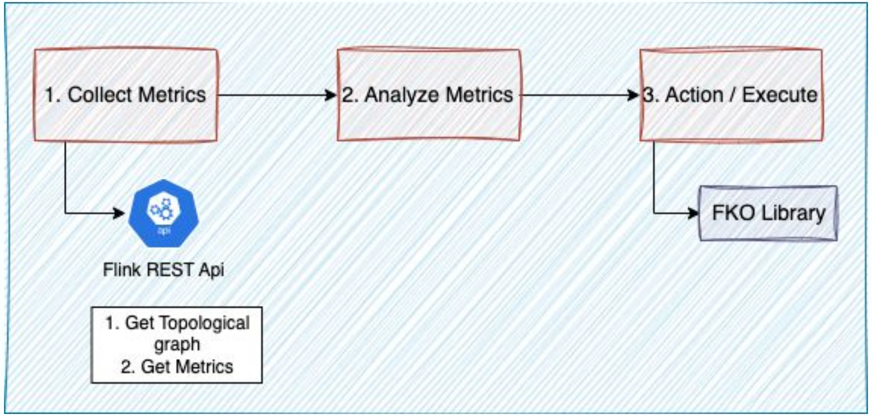
- Variable Traffic Pattern
- Performance Bottleneck
- Resource Underutilization (HPA is not enough)





Solution: Custom Autoscaler





Metrics

	Meter	Definition
1.	Throughput	How fast we process data on a given pipeline.
2.	Backlog	The current lag of the Kafka topic
3.	Backlog Growth	Backlog is increasing/decreasing or constant $Backlog\ Growth = \frac{dBacklog}{dt}$
4.	Input Rate	Input Rate = BacklogGrowth + Throughput
5.	Backlog Time	We should have some backlog for healthy processing. (~10 sec) $Backlog\ Time = \frac{Backlog}{throughput}$
6.	CPU Utilization	What % of CPU is busy

Note: https://www.infoq.com/presentations/google-cloud-dataflow/

Scale Up

Pre-condition: cpu > 75% and backlogTime > 10s

1. Increasing Backlog aka Backlog Growth > 0 :

$$Worker_{require} = Worker_{current} \frac{Input\ Rate}{Throughput}$$

2. Consistent Backlog aka Backlog Growth = 0:

$$Worker_{extra} = Worker_{current} \frac{Backlog\ Time}{Time\ to\ Reduce\ Backlog}$$

To sum up:

$$Worker_{scaleup} = min\left(Worker_{require} + Worker_{extra}, Worker_{max}\right)$$

Scale Down

Pre-condition: cpu < 75% and backlogGrowth < 0 and backlogTime < 10s

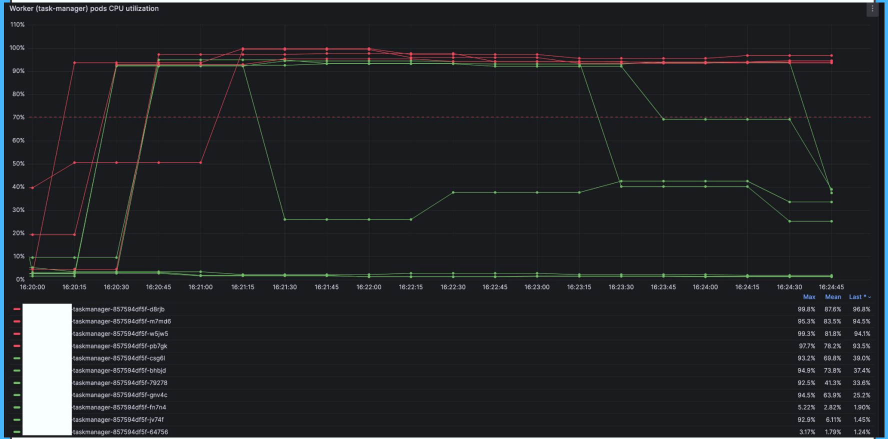
1. So the only driving factor to calculate the required resources after a scale down is CPU.

$$Cpu\ Rate_{desired} = \frac{Worker_{current}}{Worker_{new}} Cpu\ Rate_{current}$$

Note: https://www.infoq.com/presentations/google-cloud-dataflow/

Benefits

- Efficient Resource Utilization
- Customizable Scaling Parameters
- Lower Costs (reduced cost by 50%)



Before

Available Task Slots

42

Total Task Slots **122** | Task Managers **122**

Running Job List

Job Name

After

Available Task Slots

0

Total Task Slots **122** | Task Managers **122**

Running Job List

Job Name

Scaling Lessons

- Active Resource Manager vs Standalone
  - Reducing provisioning time for scale actions
- Even though we use Standalone Mode, Internally Flink Submits jobs when we do scale up and scale down.
  - We hit Metascape Out of Memory issue.
  - Solution add your jar in Flink lib directory to prevent creating multiple classloader
- Beam did not expose backlog metrics
- Flink cache metrics when tasks are rescheduled

Talat Uyarer
Rishabh Kedia

QUESTIONS?

linkedin.com/in/talatuyarer/
linkedin.com/in/rishabhkedia/