Databricks in Production

databricks

Module Objectives

1. Promote code from development to production with Databricks Repos
2. Leverage recommended best practices for managing Structured Streaming workloads on Databricks
3. Use the Databricks UI to configure and schedule multi-task jobs for task orchestration
4. Trigger and monitor Databricks jobs using the CLI & REST API
5. Troubleshoot error messages and configure logging

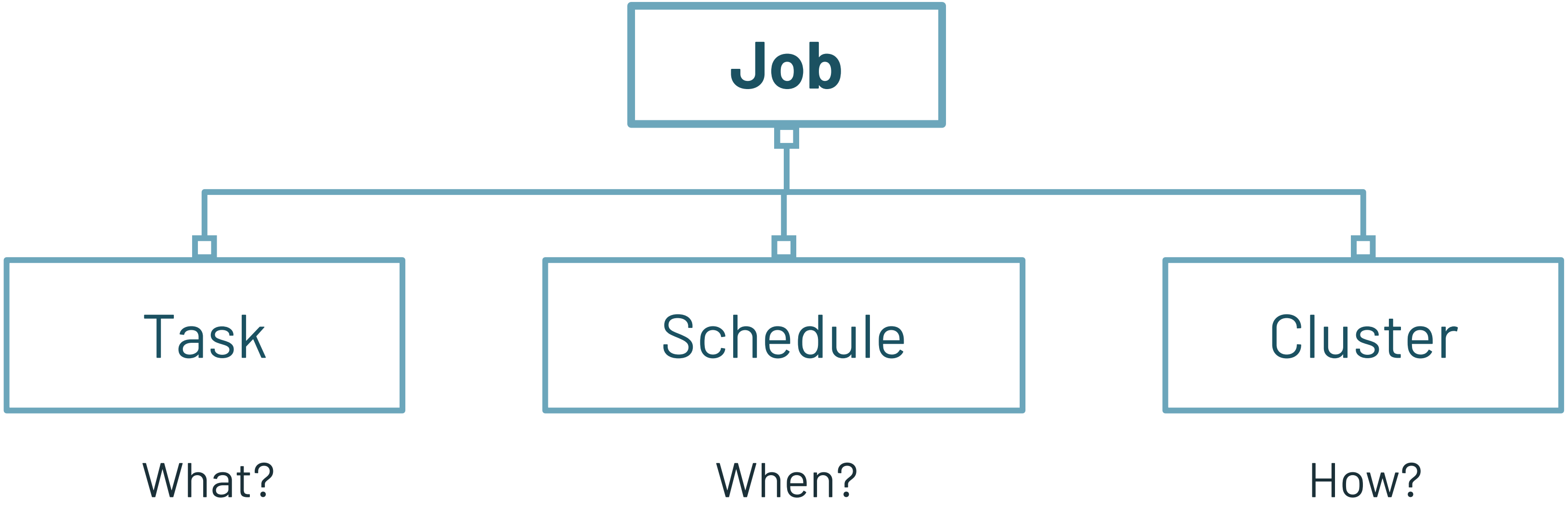
Agenda

- Orchestration and Scheduling with Multi-Task Jobs
- Monitoring, Logging, and Handling Errors
- Promoting Code with Databricks Repos
- Programmatic Platform Interactions
- Managing Costs and Latency with Incremental Workloads
- Deploying Streaming and Batch Workloads

Orchestration and Scheduling with Multi-Task Jobs

databricks

What is a Job?



Job

Task

Schedule

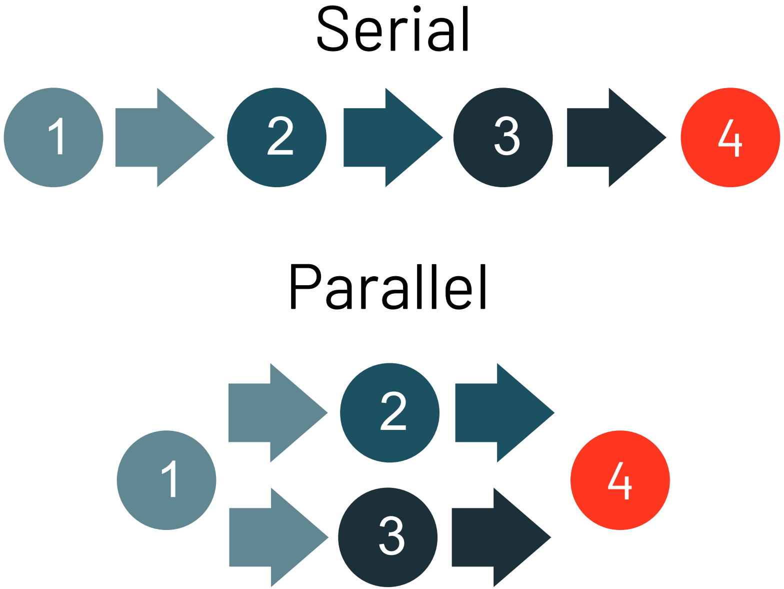
Cluster

What?

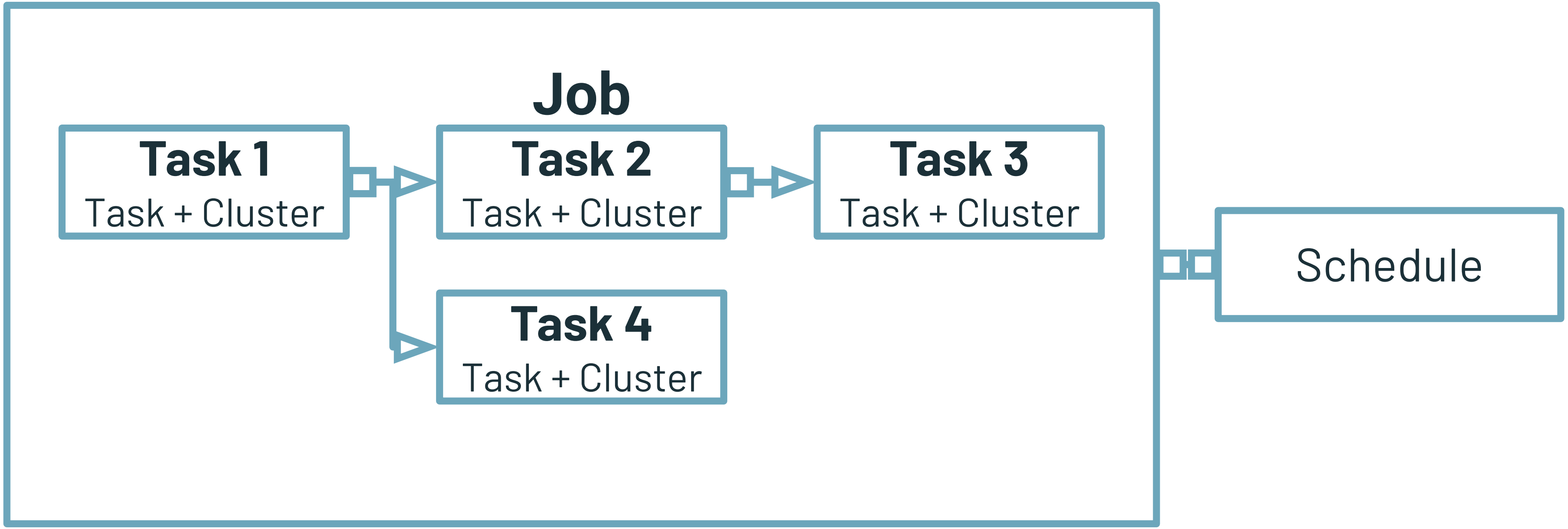
When?

How?

Orchestration with Multi-Task Jobs



Jobs revisited

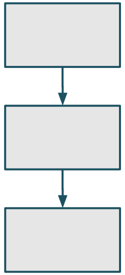


Codealong: The Jobs UI in Databricks

databricks

Common Jobs Patterns

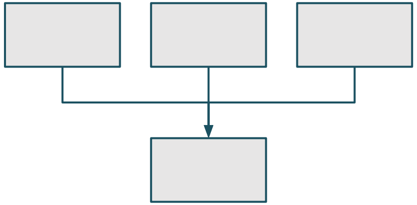
Sequence



Sequence

- Data transformation/ processing/cleaning
- Bronze/silver/gold tables

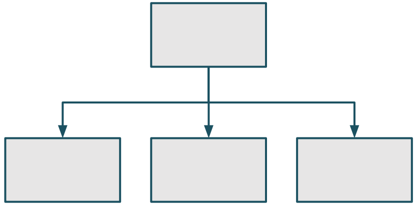
Funnel



Funnel

- Multiple data sources
- Data collection

Fan-out

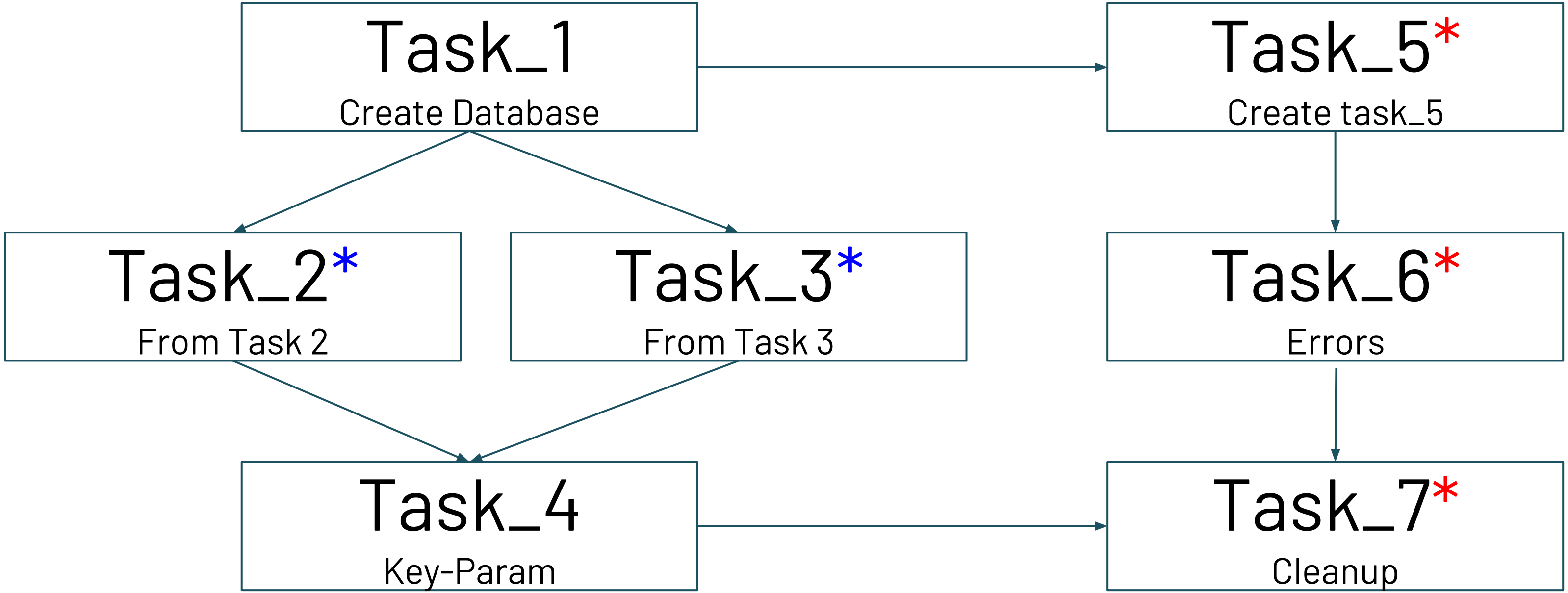


Fan-out, star pattern

- Single data source
- Data ingestion and distribution

Lab: Creating a Multi-Task Job

Jobs UI Lab



* Make sure to use a **different** cluster for Tasks #2 & #3
* Make sure to use the **same** cluster for Tasks #5, #6 & #7
* After running once with the error, update Task_6 to pass

databricks

Monitoring, Logging, and Handling Errors

databricks

Monitoring vs Observability

Monitoring	Observability
Tells you whether the system works	Lets you ask why it's not working
Is "the how" / Something you do	Is "the goal" / Something you have
An Operational Concern	Embedded at the time of system design
I *monitor* you	You *make yourself* observable

How does Monitoring apply to Databricks?

Reduce Mean Time to Detect (MTTD) outages	Something is broken, and somebody needs to fix it right now! Or, something might break soon, so somebody should look soon.
Ad-hoc retrospective analysis	The job latency just shot up; what else happened around the same time?
Build system health dashboards	Answer basic questions about the health of your jobs and track core/golden signals
Inspect and predict resource usage or cost	Create and track metrics that allow you to correlate or predict growth.
Compare / experiment configurations	Are my jobs running slower than it was last week? Can I add more machines and reduce the processing time?

Metrics To Track

System Metrics

Tracks resource-level metrics, such as CPU, memory, disk & network.

Spark Metrics

Spark has a configurable metrics system based on the Dropwizard Metrics Library. This allows users to report Spark metrics to a variety of sinks including HTTP, JMX, and CSV files.

Custom Metrics

Custom metrics ties to your service level objectives (SLOs) and indicators (SLIs).

e.g QueryExecutionListener, StreamingQueryListener

StreamingQueryListener

- This is what powers the streaming statistics in notebooks
- Listens for Query Start, Progress, and Termination events
- StreamingQueryProgress holds basic metrics
  - batchId
  - batchDuration
  - numInputRows (aggreggate number of records processed in a trigger)
  - inputRowsPerSecond (rate of data arriving)
  - processedRowsPerSecond (rate that Spark is processing data)

StreamingQueryListener

- Scala API only
- For Python, use py4j to invoke StreamingQueryListener written in Scala
- Implement by overriding onQueryStarted, onQueryProgress, and onQueryTerminated events (see package org.apache.spark.sql.streaming)
- spark.streams.addListener(new StreamingQueryListener() {...})

Logs in Databricks

Event logs

Tracks important cluster lifecycle events like cluster start, stop, resize etc.

Audit logs

Provide end-to-end logs of activities performed by Databricks users, allowing your enterprise to monitor detailed Databricks usage patterns.

Cloud provider logs

Storage logging, network logging

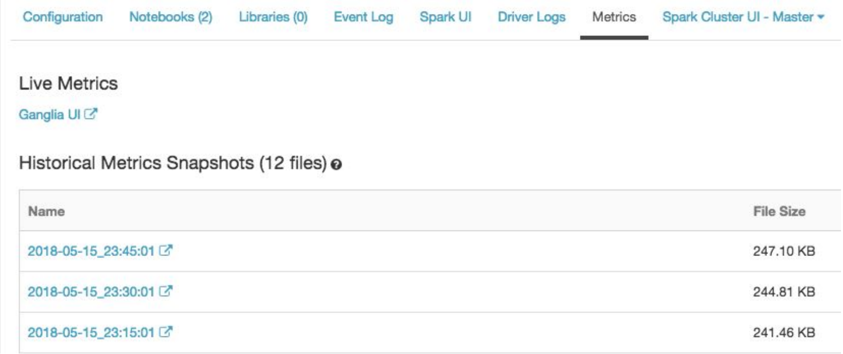
Cluster - Driver & Worker logs

log4j / stdout / stderr from Driver/Executor

Init script output

Native Solutions

Ganglia UI



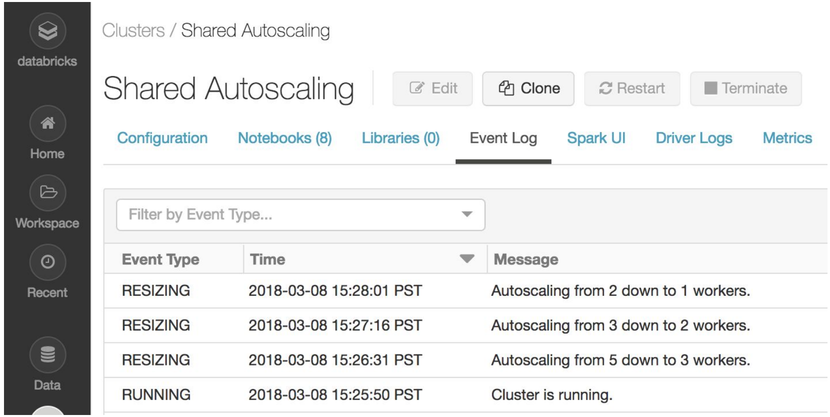
Live Metrics

Ganglia UI

Historical Metrics Snapshots (12 files)

Name	File Size
2018-05-15_23:45:01	247.10 KB
2018-05-15_23:30:01	244.81 KB
2018-05-15_23:15:01	241.46 KB

Event Logs

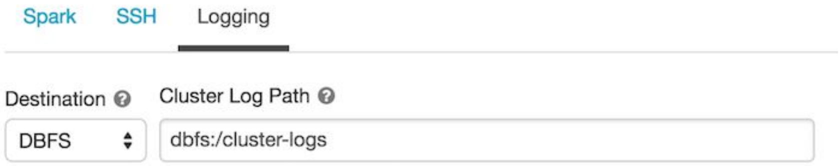


Clusters / Shared Autoscaling

Shared Autoscaling

Event Type	Time	Message
RESIZING	2018-03-08 15:28:01 PST	Autoscaling from 2 down to 1 workers.
RESIZING	2018-03-08 15:27:16 PST	Autoscaling from 3 down to 2 workers.
RESIZING	2018-03-08 15:26:31 PST	Autoscaling from 5 down to 3 workers.
RUNNING	2018-03-08 15:25:50 PST	Cluster is running.

Cluster Log Delivery



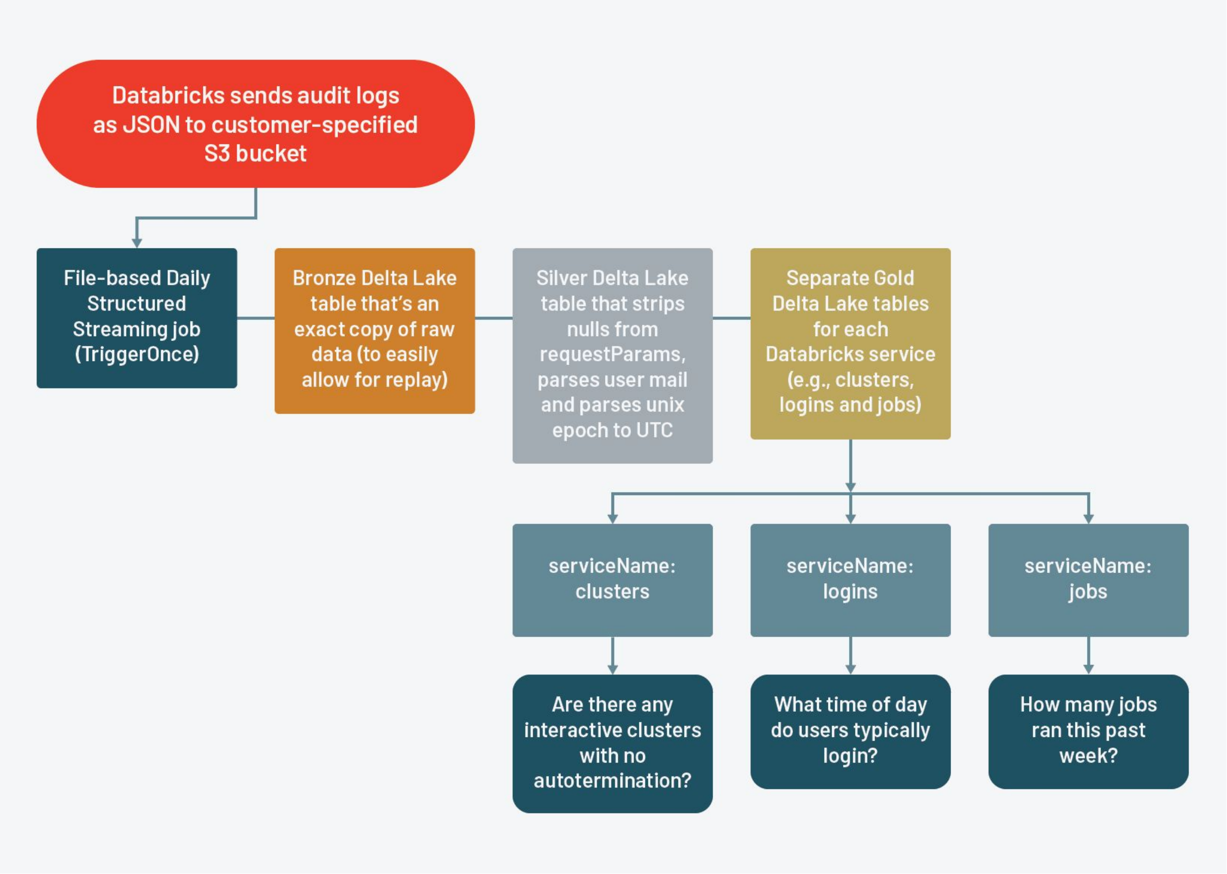
Destination: DBFS

Cluster Log Path: dbfs:/cluster-logs

databricks

Delivered Logs

- accounts
- clusters
- dbfs
- genie
- globalInitScripts
- groups
- iamRole
- instancePools
- jobs
- mlflowExperiment
- notebook
- secrets
- sqlPermissions
- ssh
- workspace



Databricks sends audit logs as JSON to customer-specified S3 bucket

File-based Daily Structured Streaming job (TriggerOnce)

Bronze Delta Lake table that's an exact copy of raw data (to easily allow for replay)

Silver Delta Lake table that strips nulls from requestParams, parses user mail and parses unix epoch to UTC

Separate Gold Delta Lake tables for each Databricks service (e.g., clusters, logins and jobs)

serviceName: clusters

serviceName: logins

serviceName: jobs

Are there any interactive clusters with no autotermination?

What time of day do users typically login?

How many jobs ran this past week?

databricks

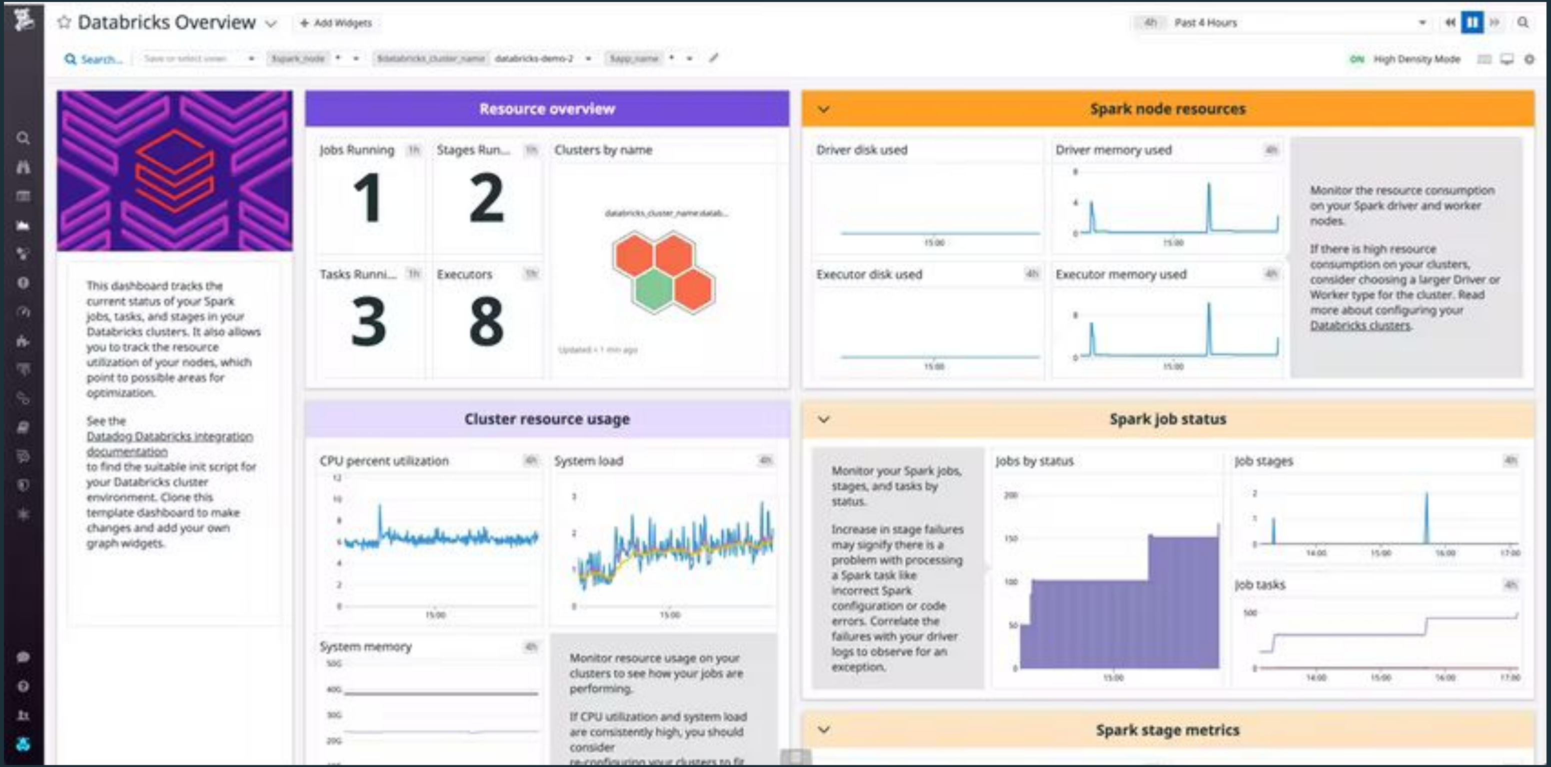
Custom Metrics in Practice

Examples of pipeline SLOs - Metrics With A Purpose

Data Freshness	● X% of data processed in Y [seconds, days, minutes] ● The oldest data is no older than Y [seconds, days, minutes] ● The pipeline job has completed successfully within Y [seconds, days, minutes]
Data correctness	● Validation error threshold ● Data Quality Score

Third Party Integrations

Datadog



This dashboard tracks the current status of your Spark jobs, tasks, and stages in your Databricks clusters. It also allows you to track the resource utilization of your nodes, which point to possible areas for optimization.

See the Datadog Databricks integration documentation to find the suitable init script for your Databricks cluster environment. Clone this template dashboard to make changes and add your own graph widgets.

Monitor the resource consumption on your Spark driver and worker nodes.

If there is high resource consumption on your clusters, consider choosing a larger Driver or Worker type for the cluster. Read more about configuring your Databricks clusters.

Monitor resource usage on your clusters to see how your jobs are performing.

If CPU utilization and system load are consistently high, you should consider re-configuring your clusters to fit

Monitor your Spark jobs, stages, and tasks by status.

Increase in stage failures may signify there is a problem with processing a Spark task like incorrect Spark configuration or code errors. Correlate the failures with your driver logs to observe for an exception.

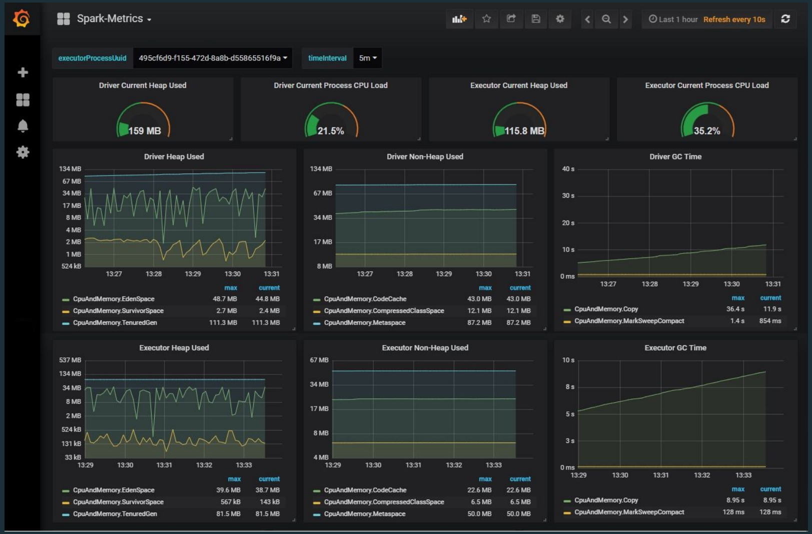
databricks

Image source: https://www.datadoghq.com/blog/databricks-monitoring-datadog/

Prometheus & Grafana

Prometheus uses a pull based model to scrape metrics from applications over http.

There are different integration options available for prometheus



executorProcessUuid 495cf6d9-f155-472d-8a8b-d55865516f9a timeInterval 5m

Driver Heap Used

	max	current
CpuAndMemory.EdenSpace	48.7 MB	44.8 MB
CpuAndMemory.SurvivorSpace	2.7 MB	2.4 MB
CpuAndMemory.TenuredGen	111.3 MB	111.3 MB

Driver Non-Heap Used

	max	current
CpuAndMemory.CodeCache	43.0 MB	43.0 MB
CpuAndMemory.CompressedClassSpace	12.1 MB	12.1 MB
CpuAndMemory.Metaspace	87.2 MB	87.2 MB

Driver GC Time

	max	current
CpuAndMemory.Copy	36.4 s	11.9 s
CpuAndMemory.MarkSweepCompact	1.4 s	854 ms

Executor Heap Used

	max	current
CpuAndMemory.EdenSpace	39.6 MB	38.7 MB
CpuAndMemory.SurvivorSpace	567 kB	143 kB
CpuAndMemory.TenuredGen	81.5 MB	81.5 MB

Executor Non-Heap Used

	max	current
CpuAndMemory.CodeCache	22.6 MB	22.6 MB
CpuAndMemory.CompressedClassSpace	6.5 MB	6.5 MB
CpuAndMemory.Metaspace	50.0 MB	50.0 MB

Executor GC Time

	max	current
CpuAndMemory.Copy	8.95 s	8.95 s
CpuAndMemory.MarkSweepCompact	128 ms	128 ms

1.)JmxSink & jmx_exporter

Databricks clusters could be configured to use JMXSink via editing the file /databricks/spark/conf/metrics.properties . Prometheus has a JMX to Prometheus exporter which is a collector that can scrape and expose mBeans of a JMX target. https://github.com/prometheus/jmx_exporter

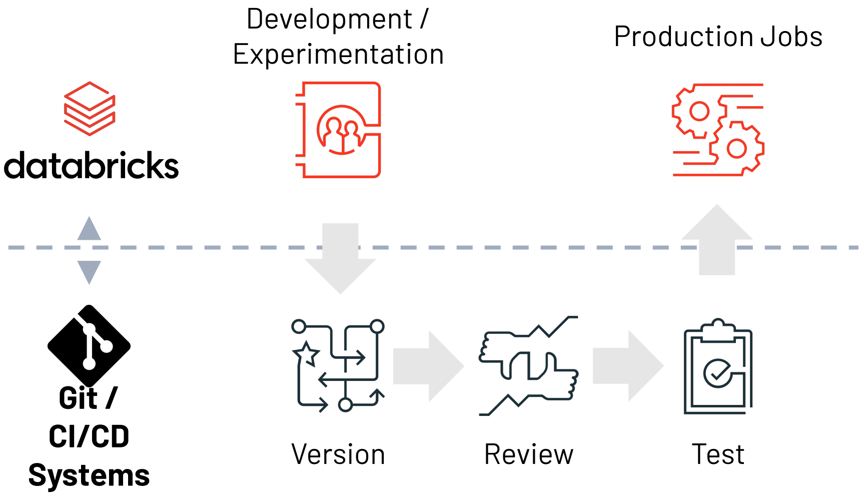
2.) banzai cloud/spark-metrics

For ephemeral or batch jobs, prometheus has a push gateway - https://github.com/prometheus/pushgateway . Since these kinds of jobs may not exist long enough to be scraped, they can instead push their metrics to a Pushgateway. The Pushgateway then exposes these metrics to Prometheus.

Lab: Troubleshooting Errors

Promoting Code with Databricks Repos

CI/CD Integration



databricks

Development / Experimentation

Production Jobs

Git / CI/CD Systems

Version

Review

Test

Supported Git Providers

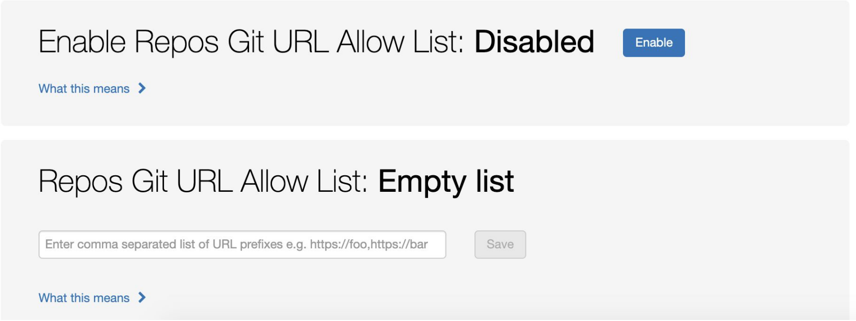
Azure DevOps

GitHub

GitLab

Bitbucket

Enterprise Readiness



Enable Repos Git URL Allow List: **Disabled**

Repos Git URL Allow List: **Empty list**

Enter comma separated list of URL prefixes e.g. https://foo,https://bar

Codealong: Import a Git Repo

https://github.com/databricks-academy/cli-demo

Codealong: Refactor %run

databricks

Codealong: Relative Imports with Python Wheel

databricks

Demo: Commit, Merge, Pull

databricks

Programmatic Platform Interactions

https://github.com/databricks-academy/cli-demo

Follow markdown lab instructions:

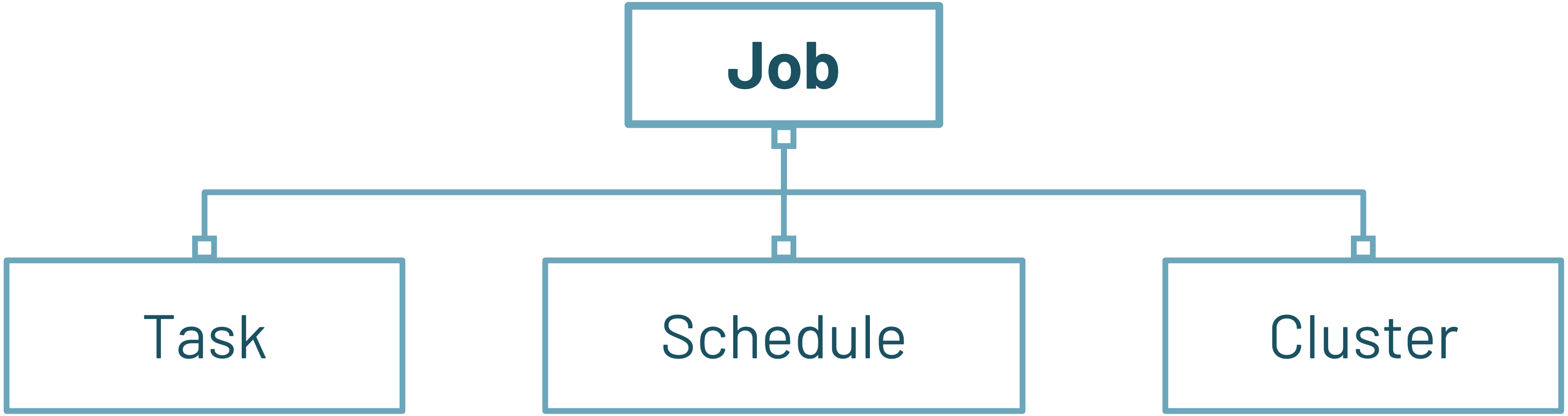
1. cli-orch.md
2. api-demo.md

Managing Costs and Latency with Incremental Workloads

Cost trade-offs

Option / Reqs	**Low latency**	**Cost effective**	**Future proof (stricter latency)**
Scheduled Batch	- (startup time)	**+ (not always on)**	- (code changes / no state concept)
Triggered Incremental Batchs	- (startup time)	**+ (not always on)**	**+ (can easily convert to always on)**
Always-on Stream	**+ (no startup time)**	- (idle cpu every x minutes)	**+ (out of the box)**

Streaming Job Recommendations



Job

Task

Up to 20-40 streaming queries

Schedule

1 concurrent run, unlimited retries

Cluster

New job cluster for each run

Driver Resource Contention

- Query planning and scheduling
- Cloud queue service processing
- Kafka source administration
- Delta transaction log administration
- Broadcasting
- Keeping track of metrics

Capacity Planning Trade-Offs

Requires monitoring and planning to determine ideal number of streaming queries per task

Extreme cluster utilization:

- Super cost efficient
- Less complicated management overhead (no load balancing)
- Fewest concurrent job runs required per shard

Extreme Isolation:

- Little to no resource contention
- Fault isolation: no other queries affected when one fails

Capacity Planning Strategies

1. Simply binpack in case of similar streams
2. Isolate streams based on their domain / pipeline, and update frequency
3. Isolate streams that require their own cluster (large hitters)
4. Isolate streams based on failure isolation requirements
5. Benchmark using representative streams to understand cluster requirements and the appropriate number of streaming queries for the workload

Optimizations and Planning

Goal	**Solution**
Reduce driver garbage collection	Enable RocksDB state store
Higher cost efficiency	Capacity planning and cluster sizing
Lower latency for small streams	Add each stream to a separate scheduler pool
More reliable recovery	Specify records processed per trigger to right-size microbatches to the cluster

Load Scaling

Elastic

- Temporary scaling up a streaming cluster to handle backlog
- Can only scale out until #cores <= #shuffle partitions

Permanent

- Requires checkpoint wipe-out since shuffle partitions is fixed per checkpoint location
- Plan ahead to recover state (leverage filters, file partitions)

Codealong: Deploying Streaming and Batch Workloads

databricks

Module Recap

1. Promote code from development to production with Databricks Repos
2. Leverage recommended best practices for managing Structured Streaming workloads on Databricks
3. Use the Databricks UI to configure and schedule multi-task jobs for task orchestration
4. Trigger and monitor Databricks jobs using the CLI & REST API
5. Troubleshoot error messages and configure logging

Course Recap

databricks

Learning Objectives

1. Design databases and pipelines optimized for the Databricks Lakehouse Platform.
2. Implement efficient incremental data processing to validate and enrich data driving business decisions and applications.
3. Leverage Databricks-native features for managing access to sensitive data and fulfilling right-to-be-forgotten requests.
4. Manage error troubleshooting, code promotion, task orchestration, and production job monitoring using Databricks tools.

databricks