Databricks in Production

databricks

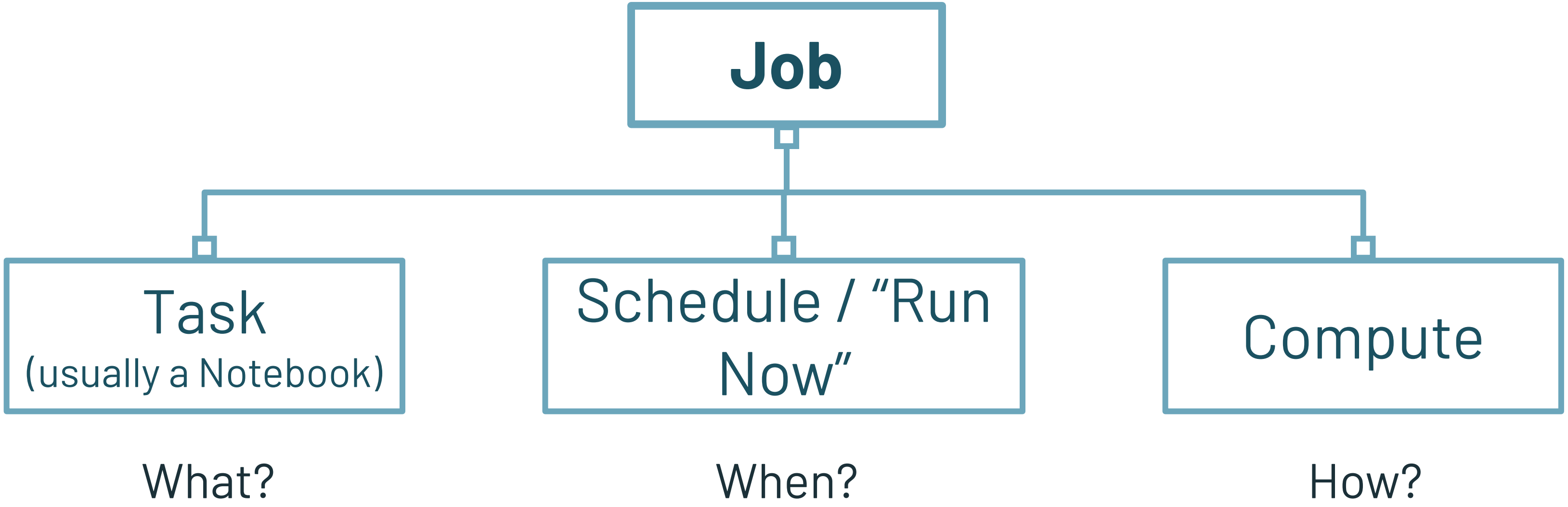
Course Objectives

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 logs

Orchestration and Scheduling with Multi-Task Jobs

databricks

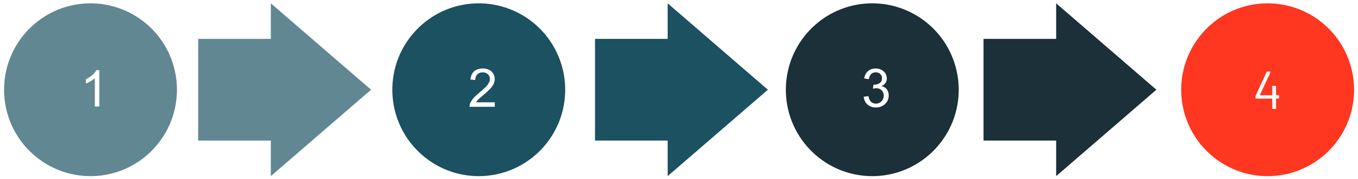
What is a Job?



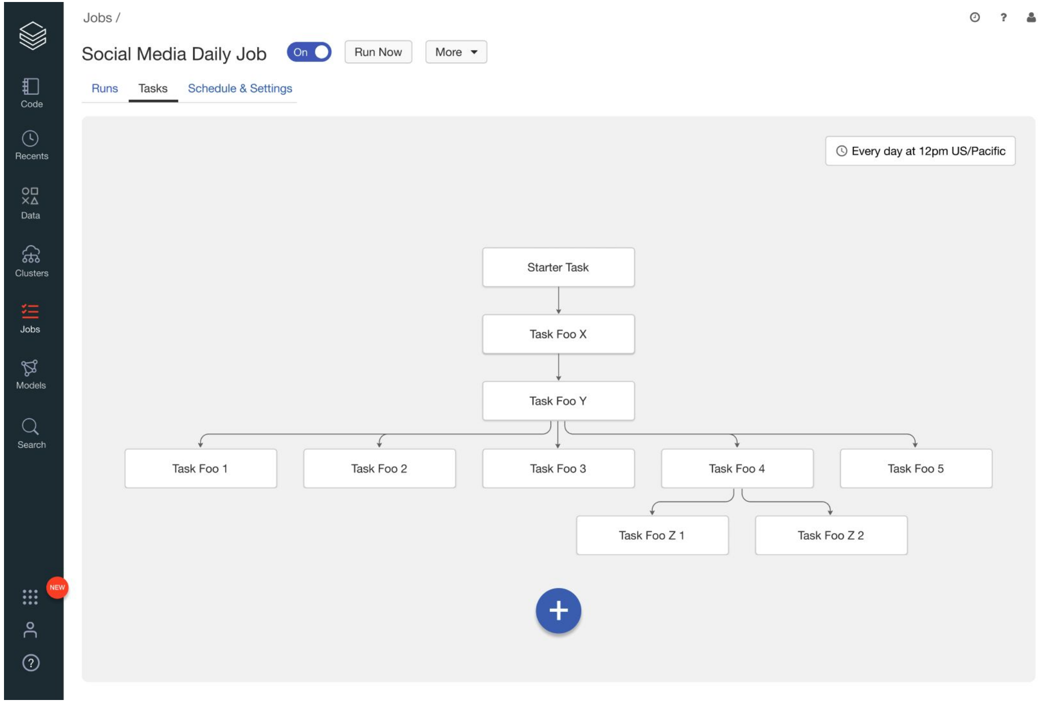
Job
Task
(usually a Notebook)
Schedule / "Run Now"
Compute
What?
When?
How?

An Introduction to Multi-Task Jobs

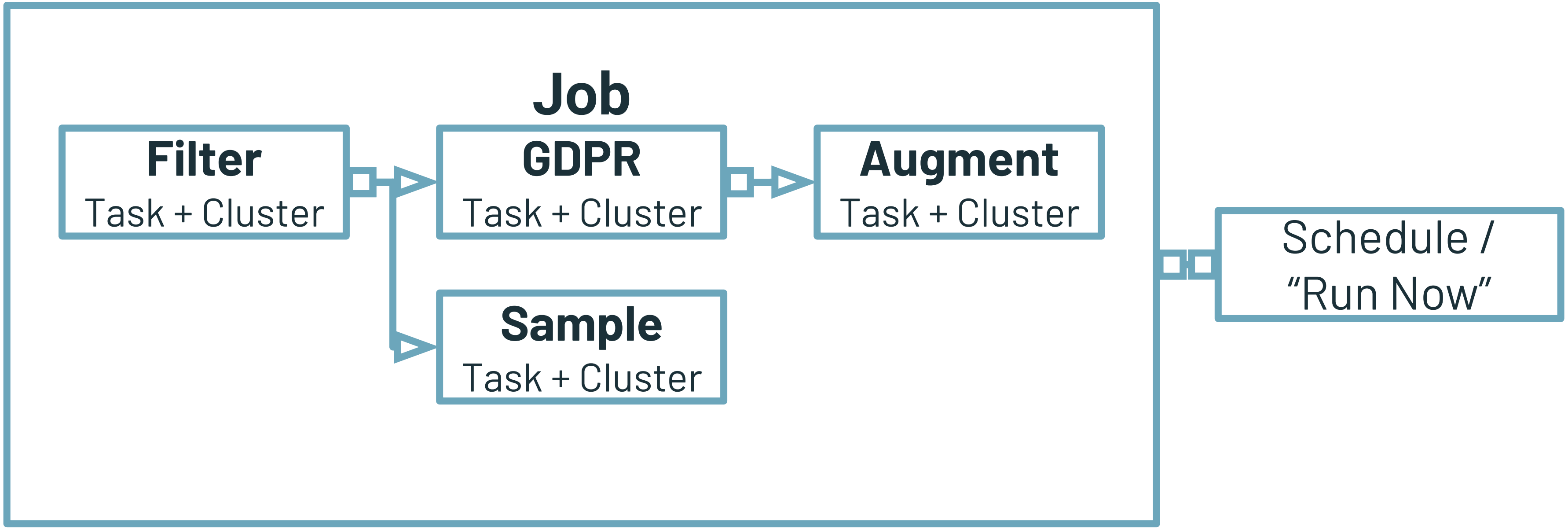
- Directed Acyclic Graphs (DAGs)



Multitask Jobs let you create a DAG of tasks



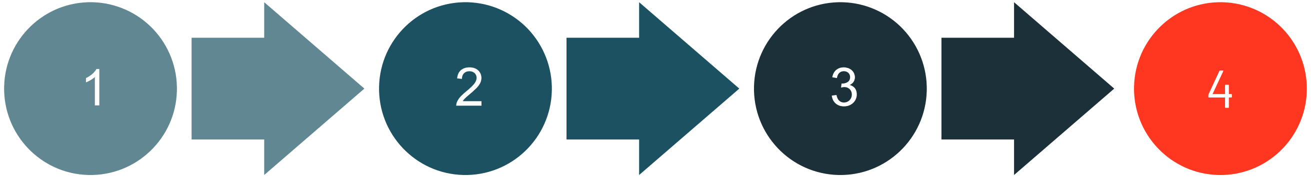
Jobs revisited



databricks

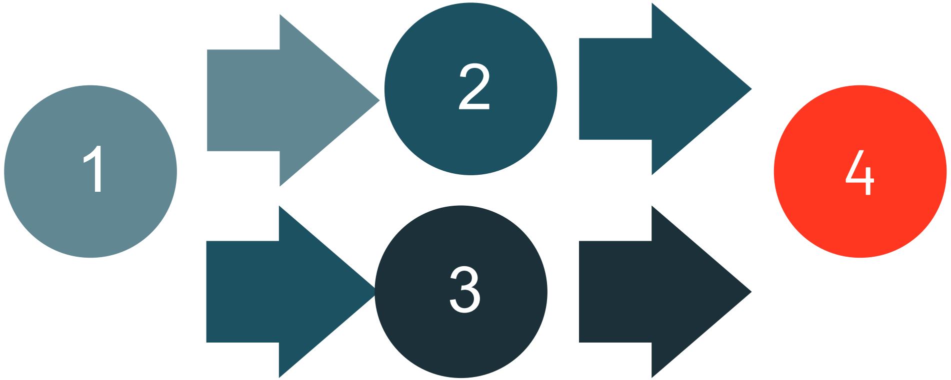
An Introduction to Multi-Task Jobs

- DAGs
  - Linear
  - Non-Linear



An Introduction to Multi-Task Jobs

- DAGs
  - Linear
  - Non-Linear



An Introduction to Multi-Task Jobs

- Jobs is a service
  - Control plane
  - One logical deployment

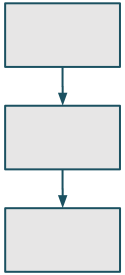
- Provides programmatic interface to manage execution

- ETL Pipelines

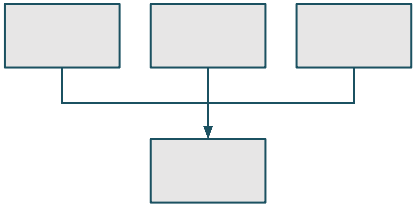
Codealong: The Jobs UI in Databricks

databricks

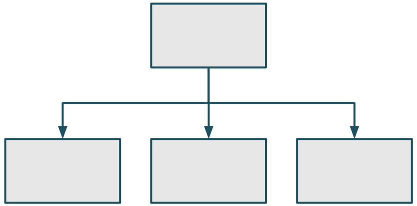
Common Jobs Patterns



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



Funnel
- Multiple data sources
- Data collection



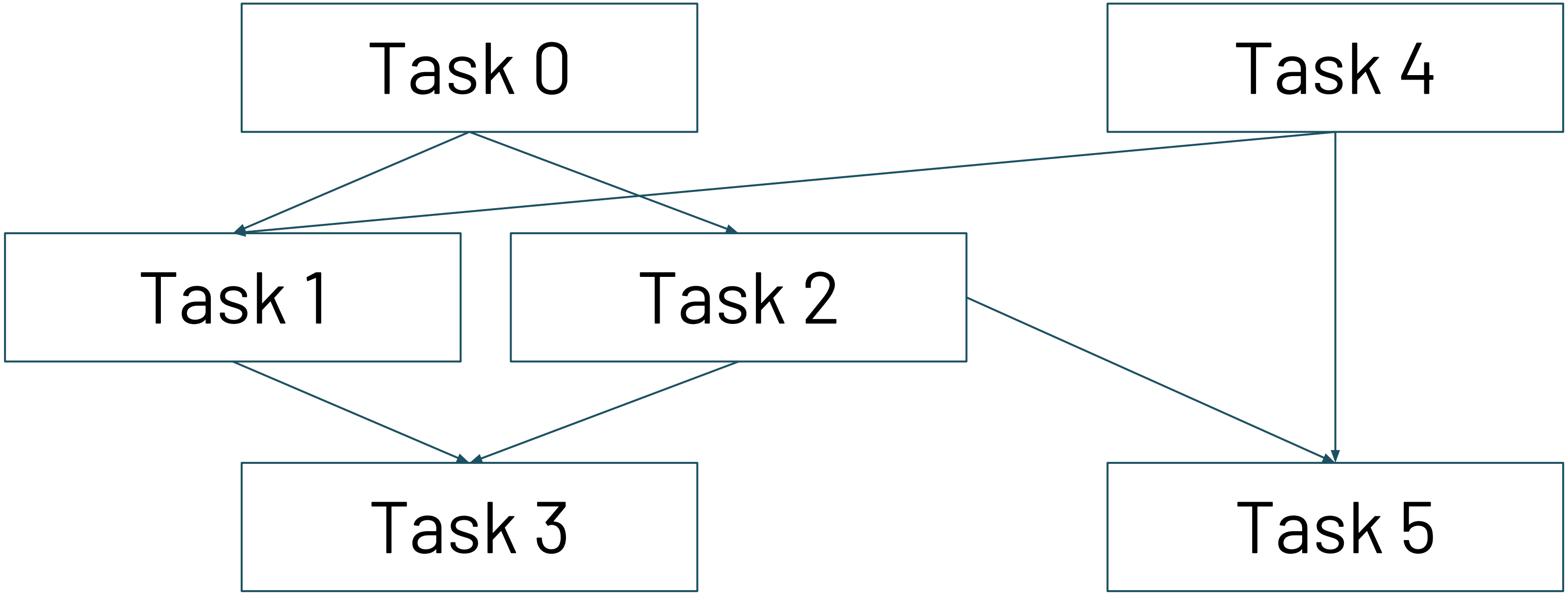
Fan-out, star pattern
- Single data source
- Data ingestion and distribution

Without multiple tasks in a Jobs:
- Each task would be a series of notebooks triggered at a specific time (hoping that the previous one has already completed)
- Notebooks triggering other notebooks with limited visibility on execution state

Lab: Creating a Multi-Task Job

databricks

Jobs UI Lab



Final Product

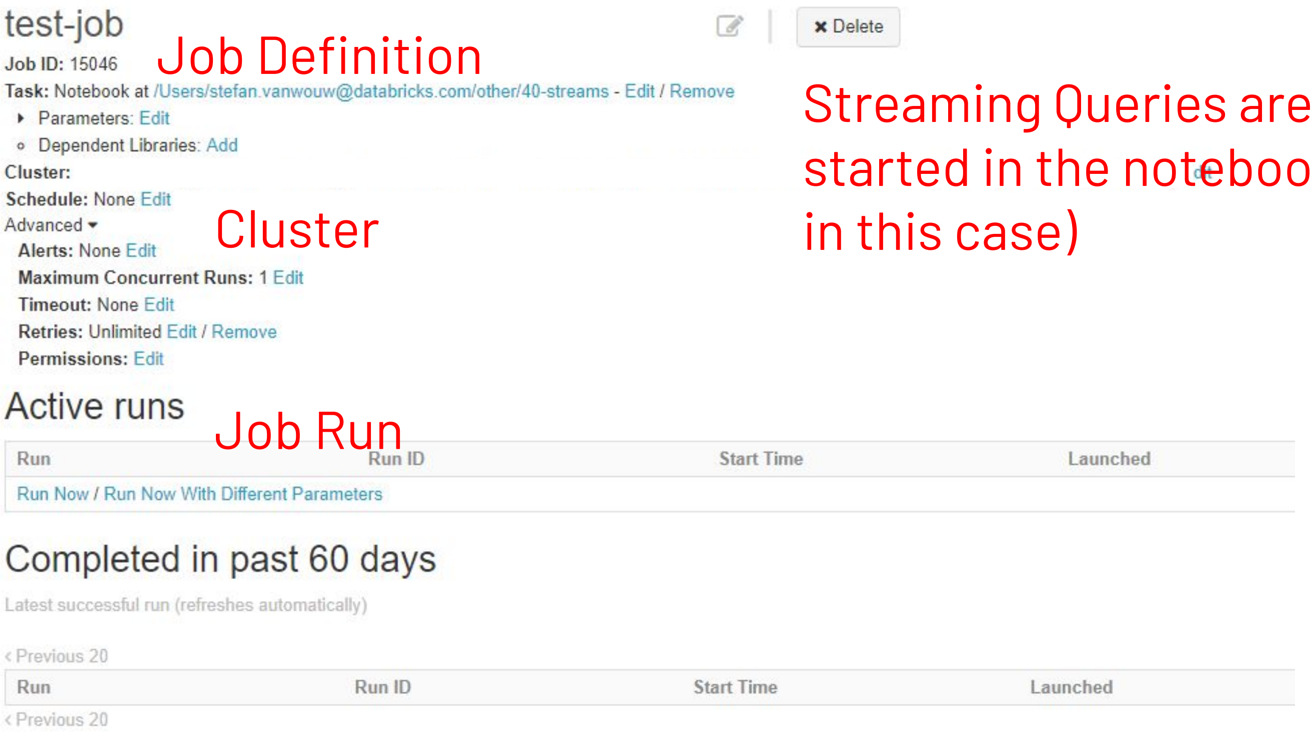
databricks

Managing Costs and Latency with Incremental Workloads

databricks

Concepts that we need to deal with

How do they translate in Databricks?



Streaming Queries are started in the notebook (40 in this case)

databricks

Concepts that we need to deal with

$$\text{Job Definition} \rightarrow \text{Job Run} \rightarrow \text{Cluster} \rightarrow \text{Structured Streaming Query}$$

Conceptually:

- Every job definition can run **m** concurrent runs at any point in time
- Every job run can only run on 1 cluster, but every cluster can support **n** concurrent runs
- Each cluster can support **p** concurrent streaming queries (e.g. to optimize for cost)

databricks

Controlling Latency in Structured Streaming

databricks

Driver resource contention

This happens on the driver (does not horizontally scale):

- Query planning and scheduling
- S3-SQS/ABS-AQS queue processing
- Kafka source administration
- Delta transaction log administration
- Broadcasting
- Keeping track of metrics
- Chauffeur/Driver connections with jobs API (another suspect)

Efficient
Structured Streaming

databricks

How many job runs per cluster?

Every job run can only run on 1 cluster, but every cluster can support **n** concurrent runs

Best practice for streaming:

- Each job run should have its own fresh "New Cluster", hence **n**=1
- This prevents ending up with "ghost" streams or otherwise polluted cluster state created by failed runs

How many job runs per job definition?

Every job definition can run **m** concurrent runs at any point in time

Best practice for streaming:

- Even though you might programmatically spin up different runs from 1 job definition (different parameters), it is recommended to use **m**=1 to prevent auto retries from spinning up multiple clusters with the same streaming queries (causing conflicts)
- Every concurrent run counts towards the shard-wide maximum (150 currently). You really want to keep the concurrent runs to a minimum, and remove any risk that anyone spins up multiple runs from the same job definition.

How many streams per cluster?

Each cluster can support **p** concurrent streaming queries (e.g. to optimize for cost)

Best practice trade-off for streaming:

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 (by default this causes the entire job run to fail)

Requires monitoring and planning to determine ideal p

Summary

For every set of **p** structured streaming queries there needs to be **m=n=1** concurrent job runs active at any point in time on an isolated cluster. Using retry **unlimited** to restart on failure using a **new cluster** every time.

Capacity planning/rollout strategies

Benchmark using representative streams to get **p** for the workload.
Autoscaling does not work well for streaming

1. Simply binpack in case of similar streams
2. Isolate streams that require their own cluster (large hitters)
3. Isolate streams based on their domain / pipeline, and update frequency
4. Isolate streams based on failure isolation requirements

In case of large number of streams a separate shard might be necessary due to the global job run limit

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)
Scheduled Trigger Once Stream	- (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)**

Other concerns and optimizations

Achieve

1. Reduction in driver GC
2. Higher cost efficiency
3. Lower latency for small streams
4. More reliable recovery

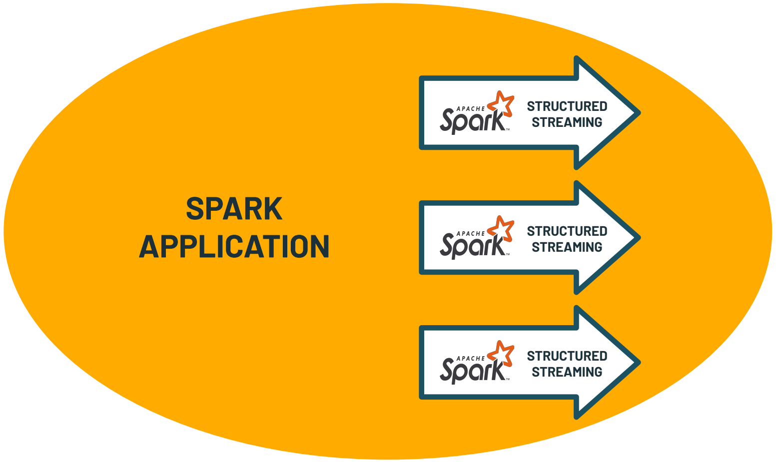
By

1. Enable G1GC
2. Capacity planning (autoscaling does not work well with Streaming) and cluster sizing
3. Add each stream to a separate FAIR scheduler pool
4. Specify a maxOffsetsPerTrigger or maxFilesPerTrigger you know the cluster can handle

How to keep your streams performant after deployment

Multiple streams per Spark cluster

- Some small streams do not warrant their own cluster
- Packing them together in one Spark application might be a good option, but then they share driver process which has performance impact



Temporary changes to load (elasticity)

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

Permanent changes to load (capacity planning)

- Permanent load increase warrants capacity planning
- Requires checkpoint wipe-out since shuffle partitions is fixed per checkpoint location!
- Think of strategy to recover state (if necessary)

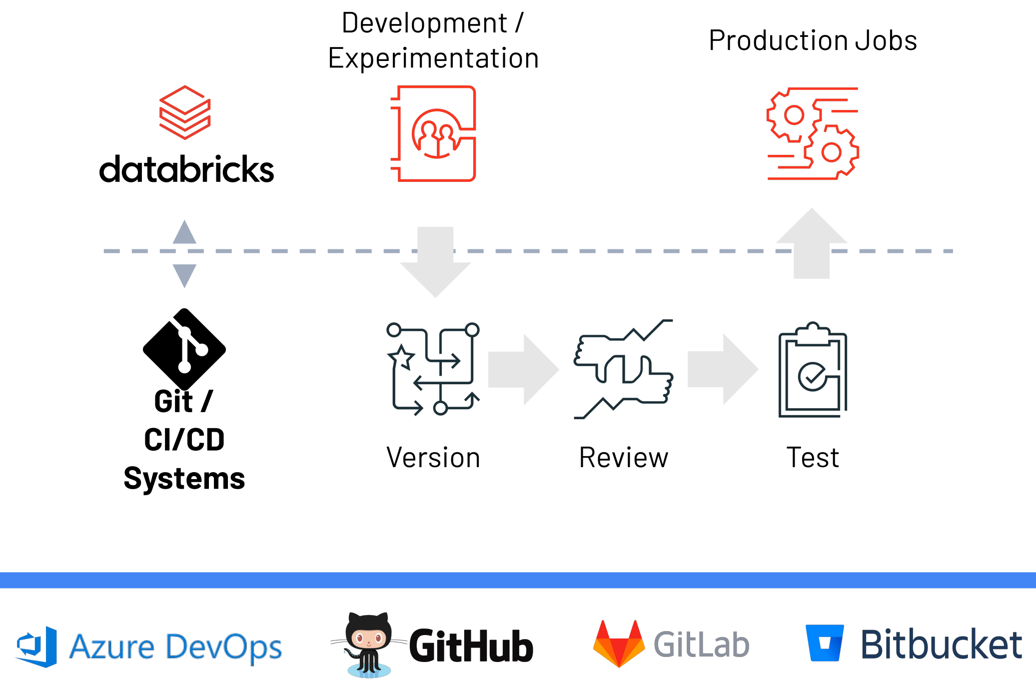
Codealong: Deploying Streaming and Batch Workloads

databricks

Promoting Code with Databricks Repos

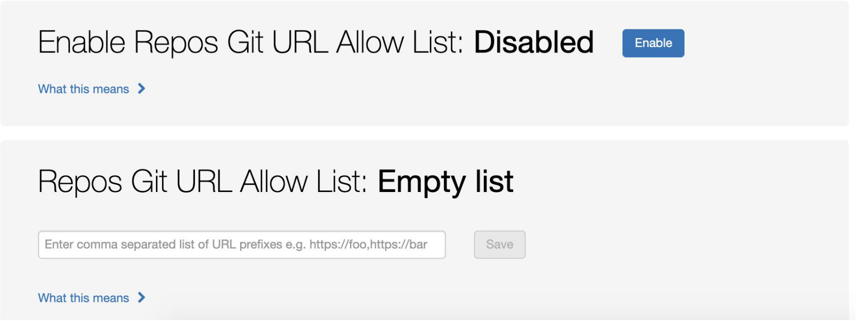
databricks

CI/CD Integration



Supported Git Providers

Enterprise Readiness



Codealong: Import a Git Repo

databricks

Codealong: Refactor %run

databricks

Codealong: Relative Imports with Python Wheel

databricks

Demo: Commit, Merge, Pull

databricks

Programmatic Platform Interactions

databricks

CLI

- Basic CLI installation and usage
  - Install: pip3 install databricks-cli
  - databricks configure -h
  - Configure with token with databricks configure –token
  - Downloading/uploading code with workspace
  - Uploading libraries
  - Stopping interactive clusters
- Controlling Jobs via the CLI
  - List clusters available databricks clusters list
  - Create a job: databricks jobs create -h
  - databricks jobs run-now --job-id <job_id> --notebook-params '{"param1":"CLI"}'

CLI Job Lab

- Get a Job ID
- Run with new parameters
- Check on run status

REST API

- Provides full feature set of Databricks product with programmatic access
- Configured with Personal Access Tokens
- Leveraged by both 1st and 3rd party integrations
- Can be used to build/configure custom applications

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

Streaming Listener

databricks

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() {...})

Logging

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

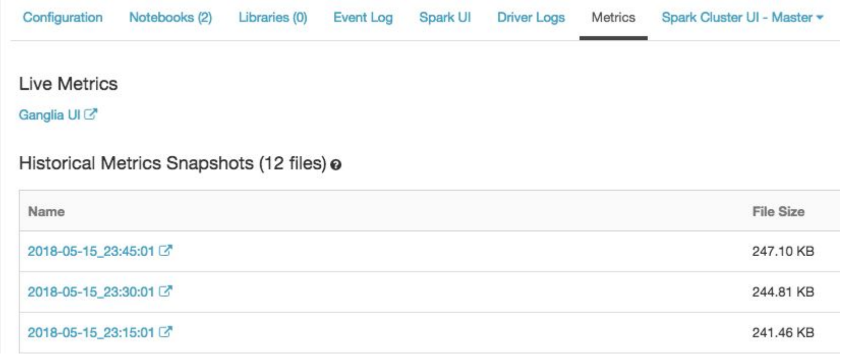
Cluster - Driver & Worker logs

log4j / stdout / stderr from Driver/Executor

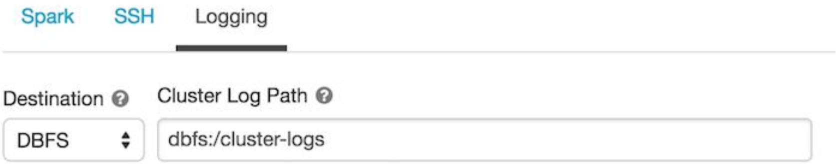
Init script output

Native Solutions

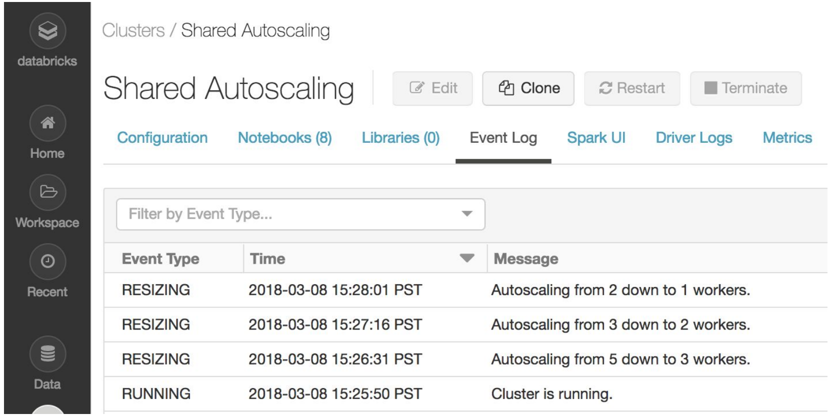
Ganglia UI



Cluster Log Delivery



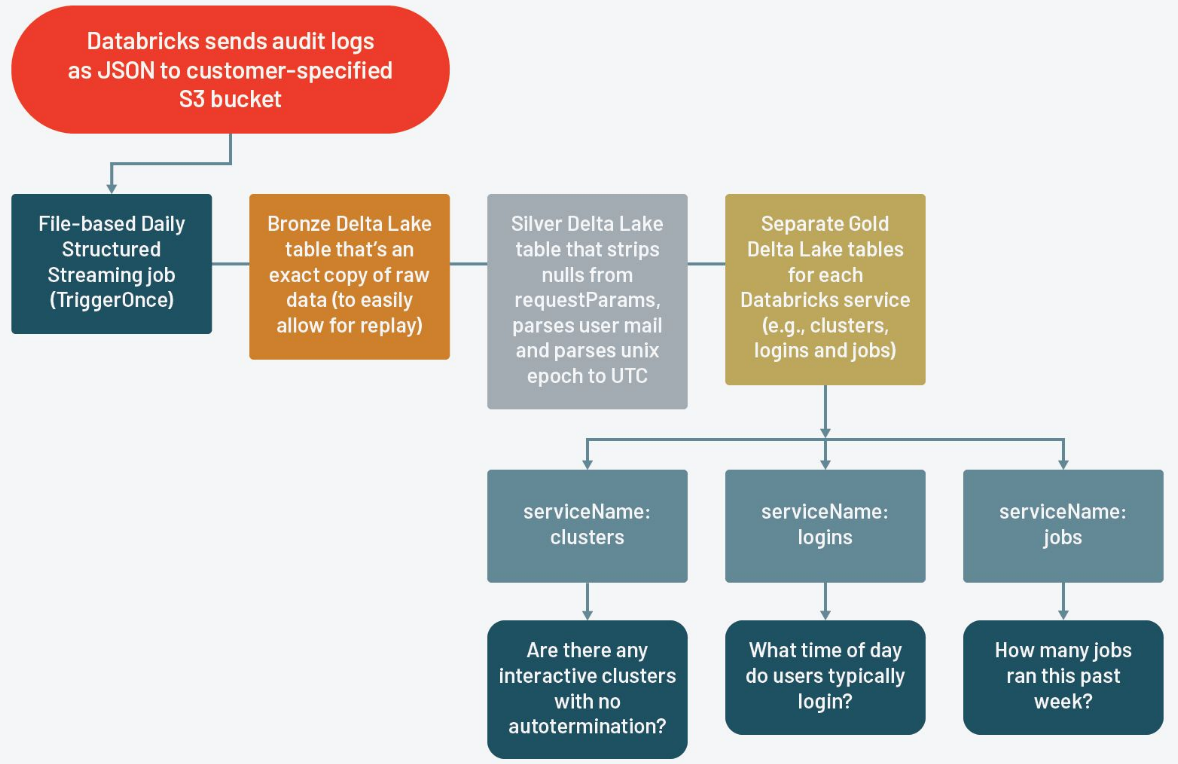
Event Logs



databricks

Delivered Logs

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



databricks

Custom Metrics in Practice

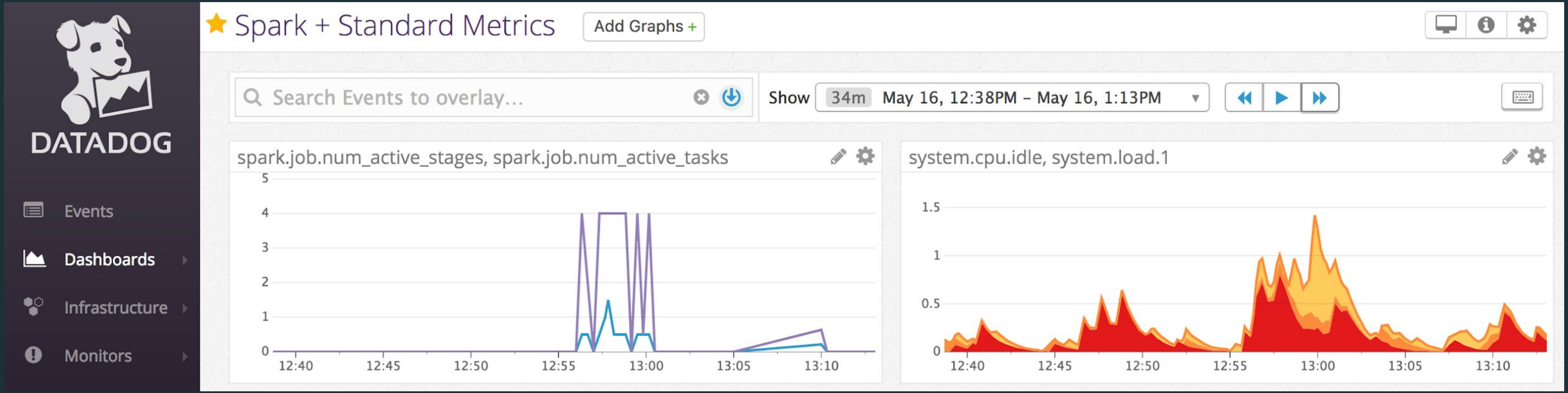
Examples of pipeline SLOs - Metrics With A Purpose

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

Datadog collects spark metrics via it's spark integration plugin



Prometheus & Grafana

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

There are different integration options available for prometheus

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.

Troubleshooting Errors

databricks

Troubleshooting Errors Lab

- Run the notebook
- Parse the run output

Course Recap

Learning Objectives

1. Build relational tables and ELT pipelines designed for the Lakehouse
2. Write Databricks-native code to incrementally process ever-expanding (streaming) data with ease
3. Design pipelines that store and delete personal identifiable information (PII) securely for data governance and compliance
4. Use best practices for developing, troubleshooting, and promoting code on Databricks
5. Implement best practices for balancing costs and latency in data pipelines
6. Schedule, orchestrate, and monitor production Databricks code

databricks