Create the kafka topic where the log records produced:

```
docker exec -it kafka kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --create \
  --topic logs \
  --partitions 2 \
  --replication-factor 1
```

Attaching VS Code to the Spark Client container

Spark does **not** run on your host machine; it runs inside Docker containers. Attaching VS Code ensures:

- **Correct Spark version:** (4.0.0)
- **Correct Python environment**
- **Correct Kafka networking**
- **Identical setup for everyone**

> **Note:** VS Code becomes a remote UI for the `spark-client` container.

Prerequisite

Install this VS Code extension on your host:

- **Dev Containers** (Microsoft)

Attach to the running container

1. Open **VS Code**.
2. Open the **Command Palette**:
   - `Ctrl + Shift + P` (Linux/Windows)
   - `Cmd + Shift + P` (macOS)
3. Select: **Dev Containers: Attach to Running Container**.
4. Choose: **spark-client**.

VS Code will reload automatically.

Verify attachment

1. Look at the **bottom-left corner** of VS Code. It should display: `Dev Container: spark-client`
2. Open a terminal in VS Code and run:

```
spark-submit --version
```

3. open the folder `/opt/spark-apps/`

Understanding the Spark Structured Streaming code

Revise the Spark Structured Streaming application example: `spark_structured_streaming_logs_processing.py`

Running the Spark Structured Streaming application

In the spark-client terminal, example of how to run the Spark application:

```
spark-submit \
  --master spark://spark-master:7077 \
  --packages org.apache.spark:spark-sql-kafka-0-10_2.13:4.0.0 \
  --num-executors 1 \
  --executor-cores 1 \
  --executor-memory 1G \
  /opt/spark-apps/spark_structured_streaming_logs_processing.py
```

```
┌─────────────────────────┐
│    Spark Client         │
│  spark-submit           │
│  (user machine / pod)   │
└────────────┬────────────┘
             │
             │ 1) submit application
             │
             ▼
┌─────────────────────────┐
│     Spark Master        │
│  (Cluster Manager)     │
│                         │
│  - registers app        │
│  - allocates workers    │
└────────────┬────────────┘
             │
             │ 2) start executors
             │
    ┌────────┴─────────┬──────────┐
    ▼                  ▼          ▼
┌───────────┐   ┌───────────┐   ┌───────────┐
│ Worker 1  │   │ Worker 2  │   │ Worker N  │
│ Executor  │   │ Executor  │   │ Executor  │
└───────────┘   └───────────┘   └───────────┘
```

See the application submission in the Spark Master: http://localhost:8080 If there are no crashes, the Spark Driver should be reacheable: http://localhost:4040

Note that the python application stored locally is submitted to the spark master's URL. Also note number of executors, cores per executors, and memory management.

Running the logs producer (load generator). This should generate the data that the Spark application processes.

Inside the `load-generator` folder, revise the `docker-compose.yaml` file, especially the number of messages generated per second. To start the load generator:

```
docker compose up -d
```

Activity 1: Understanding the execution of Spark applications

Ilustration:

```
                        +-----------------------+
                        |     Spark Driver      |
                        |----------------------|
                        | - Job scheduling     |
                        | - DAG management     |
                        | - Resource tracking  |
                        +-----------------------+
                                    |
                                    v
                          +-----------------+
                          |   Worker 1      |
                          |-----------------|
                          |  +-----------+  |
                          |  | Executor 1|  |
                          |  |-----------|  |
                          |  | Job 1     |  |
                          |  | Stage 1   |  |
                          |  |  * Task0  |  |
                          |  |  * Task1  |  |
                          |  | Stage 2   |  |
                          |  |  * Task0  |  |
                          |  |  * Task1  |  |
                          |  +-----------+  |
```

```
                                  +-----------------+
                                  +-----------------+
                                  |   Worker 2      |
                                  |-----------------|
                                  |  +-----------+  |
                                  |  | Executor 2|  |
                                  |  |-----------|  |
                                  |  | Job 2     |  |
                                  |  | Stage 1   |  |
                                  |  |  * Task0  |  |
                                  |  |  * Task1  |  |
                                  |  | Stage 2   |  |
                                  |  |  * Task0  |  |
                                  |  |  * Task1  |  |
                                  |  +-----------+  |
                                  +-----------------+


Kafka Input Topic
+---------------+ +---------------+  +----------------+
|Partition0 (P0)| |Partition1 (P1)|  |Partition2 (P2) |
+---------------+ +---------------+  +----------------+
       |              |              |
       v              v              v
     +------------------------------------+
     |          Dataflow DAG               |
     |-----------------------------------  |
     |Stage 1: Map/Filter/Parse (3 tasks)|
     |  Task0 (P0) -> Shuffle -> Stage2   |
     |  Task1 (P1) -> Shuffle -> Stage2   |
     |  Task2 (P2) -> Shuffle -> Stage2   |
     |                                    |
     |Stage 2: Aggregation/Join (2 tasks)|
     |     * Task0 <- shuffled data       |
     |     * Task1 <- shuffled data       |
     +------------------------------------+
                    |
                    v
               +-----------+
               |  Sink     |
               |(Kafka,    |
               | HDFS, etc)|
               +-----------+
```

1. Accessing the Interface

Once your Spark application is running, the Web UI is hosted by the **Driver**: http://localhost:4040

---

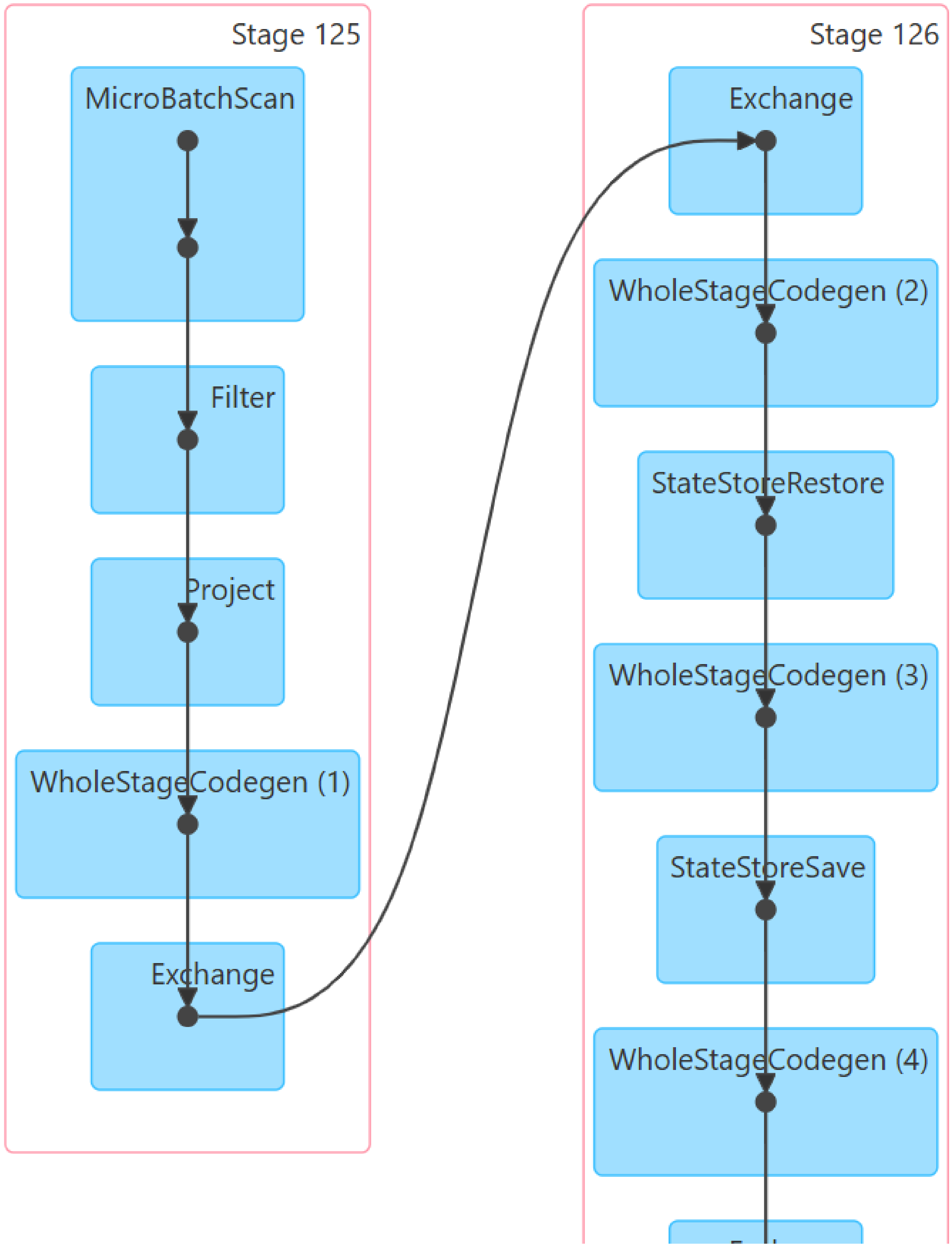
2. Key Concepts to Observe

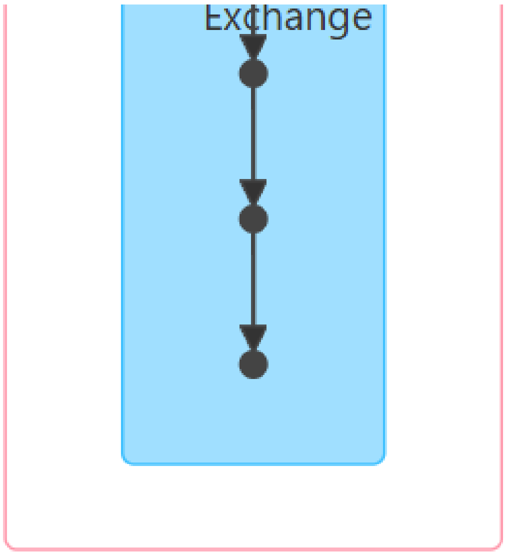
As you navigate the UI, find and analyze the following sections to see Spark theory in action:

A. The Jobs Tab & DAG Visualization

Every **Action** (like `.count()`, `.collect()`, or `.save()`) triggers a Spark Job.

- **Task:** Click on a Job ID to see the **DAG Visualization**.
- **Concept:** Observe how Spark groups operations. Transformations like `map` or `filter` stay in one stage, while `sort` or `groupBy` create new stages.





Stages:

Stage 125

- MicroBatchScan: Read new data from Kafka for the current micro-batch.
- Filter: Keep only rows matching the filter condition.
- Project: Select or compute the needed columns.
- WholeStageCodegen: Compile filter + project into optimized JVM code.
- Exchange: Shuffle data to repartition by key for stateful processing.

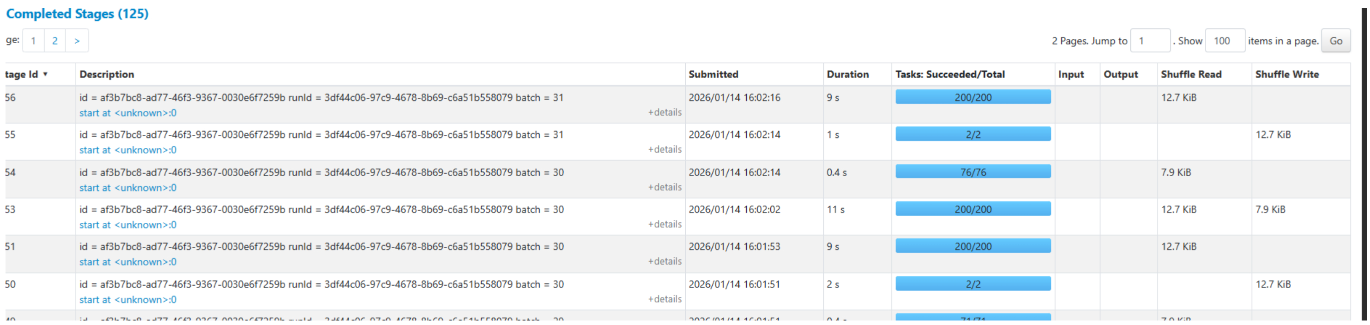
Stage 126

- Exchange: Receive shuffled data for processing.
- WholeStageCodegen: Optimize initial computation on the batch.
- StateStoreRestore: Load previous state for stateful operations.
- WholeStageCodegen: Perform the main computation on the batch.
- StateStoreSave: Save updated state for fault tolerance.
- WholeStageCodegen: Final computation before output.
- Exchange: Repartition for writing to sink if needed.

B. The Stages Tab

Stages represent a set of tasks that can be performed in parallel without moving data between nodes.

- **Concept:** Look for **Shuffle Read** and **Shuffle Write**. This represents data moving across the network—the most "expensive" part of distributed computing.



Completed Stages (125)

ge: 1 2 > 2 Pages. Jump to 1 . Show 100 items in a page. Go

tage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
56	id = af3b7bc8-ad77-46f3-9367-0030e6f7259b runId = 3df44c06-97c9-4678-8b69-c6a51b558079 batch = 31 start at <unknown>:0 +details	2026/01/14 16:02:16	9 s	200/200			12.7 KiB	
55	id = af3b7bc8-ad77-46f3-9367-0030e6f7259b runId = 3df44c06-97c9-4678-8b69-c6a51b558079 batch = 31 start at <unknown>:0 +details	2026/01/14 16:02:14	1 s	2/2				12.7 KiB
54	id = af3b7bc8-ad77-46f3-9367-0030e6f7259b runId = 3df44c06-97c9-4678-8b69-c6a51b558079 batch = 30 start at <unknown>:0 +details	2026/01/14 16:02:14	0.4 s	76/76			7.9 KiB	
53	id = af3b7bc8-ad77-46f3-9367-0030e6f7259b runId = 3df44c06-97c9-4678-8b69-c6a51b558079 batch = 30 start at <unknown>:0 +details	2026/01/14 16:02:02	11 s	200/200			12.7 KiB	7.9 KiB
51	id = af3b7bc8-ad77-46f3-9367-0030e6f7259b runId = 3df44c06-97c9-4678-8b69-c6a51b558079 batch = 30 start at <unknown>:0 +details	2026/01/14 16:01:53	9 s	200/200			12.7 KiB	
50	id = af3b7bc8-ad77-46f3-9367-0030e6f7259b runId = 3df44c06-97c9-4678-8b69-c6a51b558079 batch = 30 start at <unknown>:0 +details	2026/01/14 16:01:51	2 s	2/2				12.7 KiB

C. The Executors Tab

This shows the "Workers" doing the actual computation.

- **Concept:** Check for **Data Skew**. If one executor has 10GB of Shuffle Read while others have 10MB, your data is not partitioned evenly.

Executors

▸ Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(2)	0	23.9 MiB / 848.3 MiB	0.0 B	1	2	0	15763	15765	30 min (14 s)	0.0 B	1 MiB	671.1 KiB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(2)	0	23.9 MiB / 848.3 MiB	0.0 B	1	2	0	15763	15765	30 min (14 s)	0.0 B	1 MiB	671.1 KiB	0

Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump	Heap Histogram	Add Time	Remove Time
driver	22337c3ac65b:41105	Active	0	11.9 MiB / 434.4 MiB	0.0 B	0	0	0	0	0	17 min (1 s)	0.0 B	0.0 B	0.0 B		Thread Dump	Heap Histogram	2026-01-14 16:46:53	-
0	172.22.0.5:46657	Active	0	11.9 MiB / 413.9 MiB	0.0 B	1	2	0	15763	15765	14 min (13 s)	0.0 B	1 MiB	671.1 KiB	stdout stderr	Thread Dump	Heap Histogram	2026-01-14 16:47:00	-

Showing 1 to 2 of 2 entries

Previous 1 Next

Only two executer one driver and one actual worker.

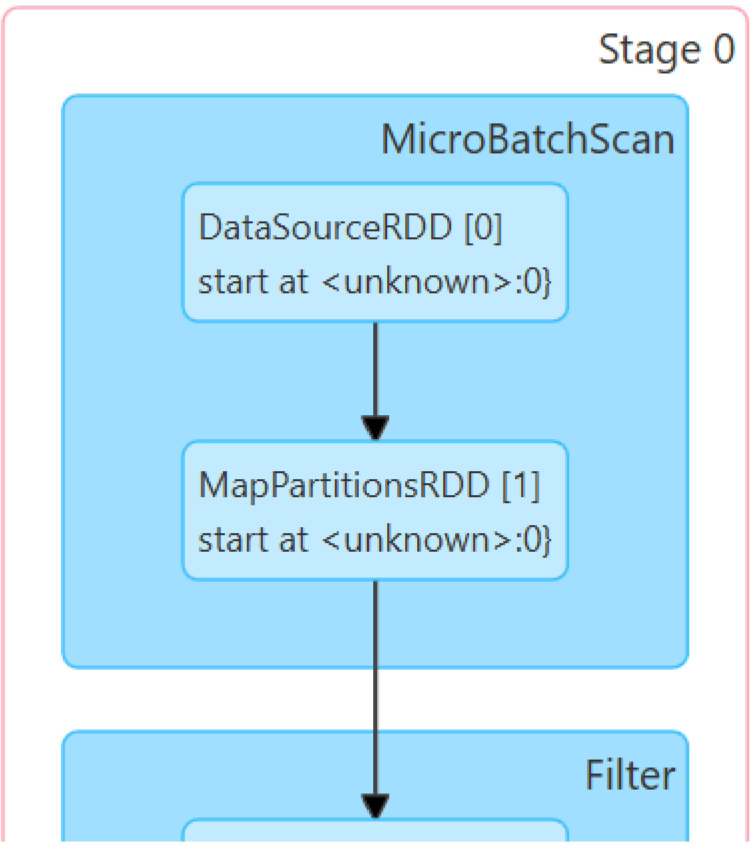
---

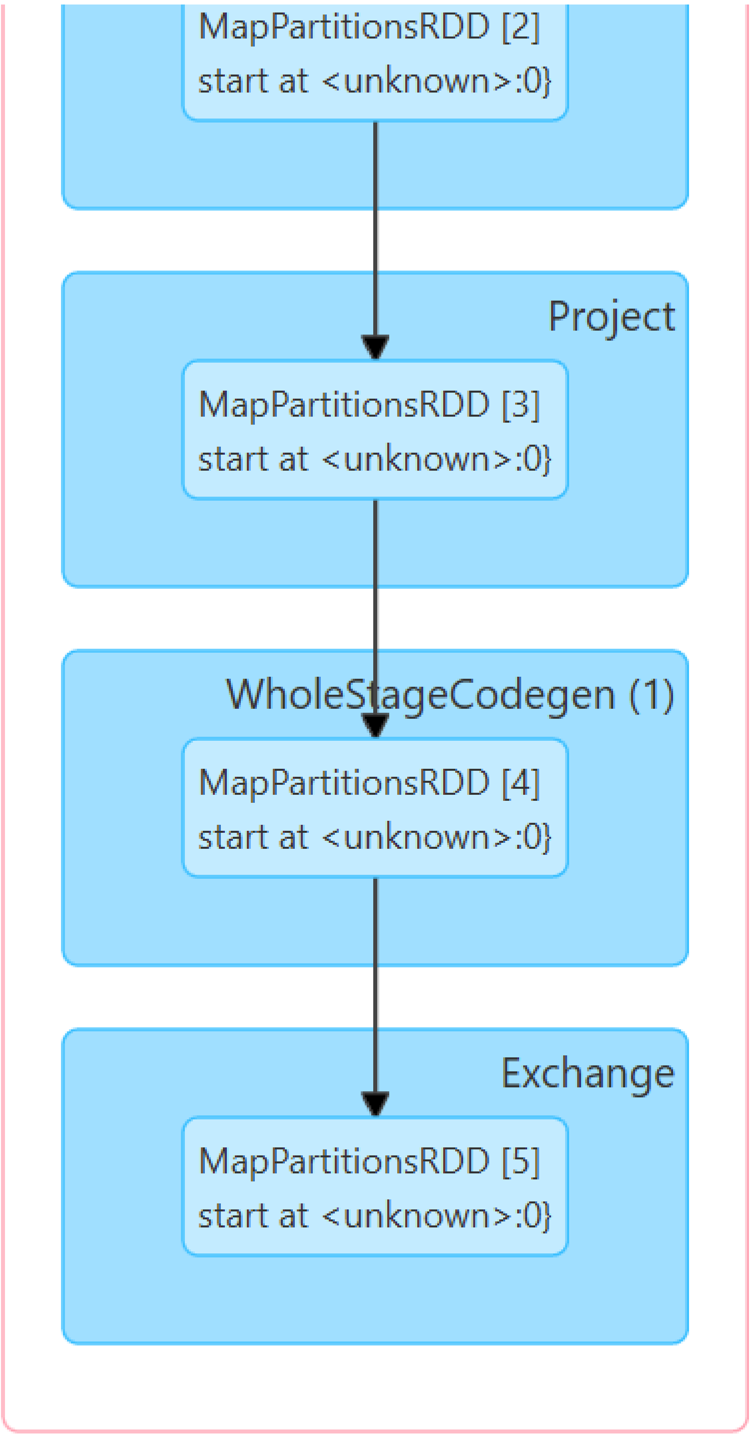
3. Practical Exploration Questions

While your application is running, try to answer these questions:

1. **The Bottleneck:** Which Stage has the longest "Duration"? What are the technical reasons for it?

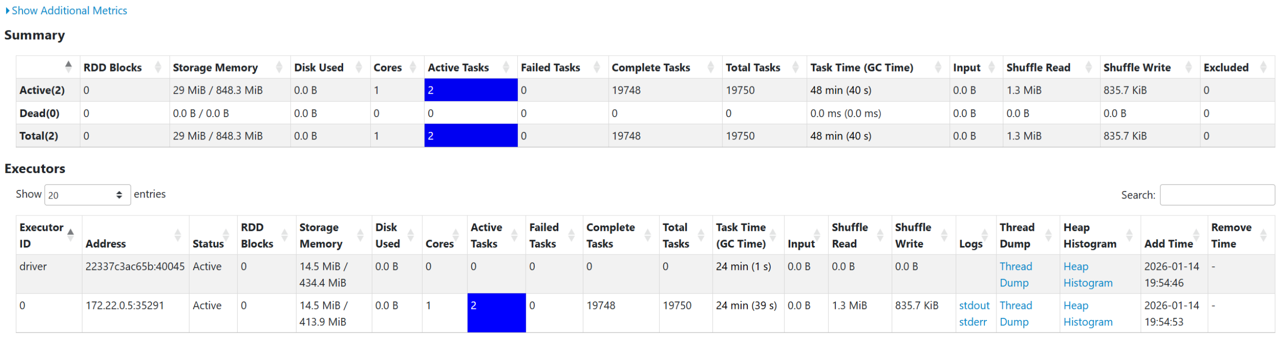
Stage Id	Description	Submitted	Duration ▾	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	id = e5aa2f5c-73c5-4ddf-9baa-86855cdfc36c runId = 0abab63f-7c69-4957-b011-2519fa342f2b batch = 0 start at <unknown>:0 +details	2026/01/14 18:54:56	6.9 min	2/2				12.8 KiB
5	id = e5aa2f5c-73c5-4ddf-9baa-86855cdfc36c runId = 0abab63f-7c69-4957-b011-2519fa342f2b batch = 1 start at <unknown>:0 +details	2026/01/14 19:02:33	33 s	2/2				12.7 KiB





- Stage id 0 took 7 minutes:
  - MicroBatchScan (Kafka read) is the bottleneck
  - Reads all available Kafka offsets since last batch
  - The generator produces 10,000 msgs/sec
  - Backlog accumulates --> each batch grows

2. **Resource Usage:** In the Executors tab, how much memory is currently being used versus the total capacity?



Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(2)	0	29 MiB / 848.3 MiB	0.0 B	1	2	0	19748	19750	48 min (40 s)	0.0 B	1.3 MiB	835.7 KiB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(2)	0	29 MiB / 848.3 MiB	0.0 B	1	2	0	19748	19750	48 min (40 s)	0.0 B	1.3 MiB	835.7 KiB	0

Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump	Heap Histogram	Add Time	Remove Time
driver	22337c3ac65b:40045	Active	0	14.5 MiB / 434.4 MiB	0.0 B	0	0	0	0	0	24 min (1 s)	0.0 B	0.0 B	0.0 B		Thread Dump	Heap Histogram	2026-01-14 19:54:46	-
0	172.22.0.5:35291	Active	0	14.5 MiB / 413.9 MiB	0.0 B	1	2	0	19748	19750	24 min (39 s)	0.0 B	1.3 MiB	835.7 KiB	stdout stderr	Thread Dump	Heap Histogram	2026-01-14 19:54:53	-

- 29.0 MiB of 848.3 MiB are used at the moment

3. Explain with your own words the main concepts related to performance and scalability in the scenario of Spark Structured Streaming.

- Micro-batching for efficient execution:
    - Instead of processing each event individually, Spark groups records into micro-batches.
    - This improves performance because:
        - Task scheduling overhead is amortized over many records
        - JVM and code-generation optimizations apply to batches
        - Network and I/O operations are more efficient in bulk
- Parallel execution through partitioning:
    - Spark improves performance by splitting the data into partitions and processing them in parallel.
        - Kafka topics are divided into partitions
        - Each partition can be read by a separate Spark task
        - Tasks run concurrently on different executor cores
- Shuffle optimization and controlled data movement
    - Writes intermediate data in a compressed, serialized format
    - Uses efficient network transfer mechanisms
    - Allows tuning of shuffle partitions
- State mangement
    - For stateful streaming queries, Spark maintains intermediate results across micro-batches.
    - State enables advanced analytics such as windowed aggregations
    - Spark stores state in a fault-tolerant StateStore backed by disk
    - Memory is used aggressively to cache hot state and reduce I/O
- Scalability Mechanisms
    - Spark Structured Streaming is designed to scale horizontally.
    - Adding executors across machines increases throughput
    - Kafka partitions enable parallel ingestion

Summary:

Spark Structured Streaming improves performance through micro-batch execution, optimized query plans, whole-stage code generation, and parallel processing across executors and partitions. Scalability is achieved by increasing cores, executors, memory, and Kafka partitions while tuning shuffle and state management parameters to match available hardware.

Activity 2: Tuning for High Throughput

The Challenge

Your goal is to scale your application to process **several hundred thousand events per second are processed with batch sizes under 20 seconds to maintain reasonable event latency and data freshness**. On a standard laptop (8 cores / 16 threads), it is possible to process **1 million records per second** with micro-batch latencies staying below 12 seconds.

Please note that the `TARGET_RPS=10000` configuration in the docker compose file of the load generator. This value represents how many records per second each instance of the load generator should produce. The load generator can also run in parallel with multiple docker instances to increase the generation speed.

The Baseline Configuration

Review the starting configuration below. Identify which parameters are limiting the application's ability to use your hardware's full potential:

From the previous example of how to run the Spark application:

```
spark-submit \
  --master spark://spark-master:7077 \
  --packages org.apache.spark:spark-sql-kafka-0-10_2.13:4.0.0 \
  --num-executors 1 \
  --executor-cores 1 \
  --executor-memory 1G \
  /opt/spark-apps/spark_structured_streaming_logs_processing.py
```

- Limits:
  - --num-executors 1
    - Limits the application to a single executor
    - Prevents horizontal parallelism across multiple CPUs or machines
    - Even if the host has many cores, Spark can only run on one executor JVM
  - --executor-cores 1
    - Allows only one task to run at a time within the executor
    - Spark cannot process multiple partitions concurrently
    - Kafka partitions, shuffle partitions, and tasks are processed serially
  - --executor-memory 1G
    - Restricts how much data can be kept in memory
    - Forces shuffle and state data to spill to disk
    - Increases I/O overhead during aggregations and joins
- Result:

  - Even on a multi-core machine, most CPU and memory resources remain unused.

Tuning Configurations (The "Knobs")

You must decide how to adjust the configurations to increase the performance. Consider the relationship between your **CPU threads**, **RAM availability**, and **Parallelism**. Examples of configurations

Parameter	Impact on Performance
`--num-executors`	Defines how many parallel instances (executors) run.
`--executor-cores`	Defines how many tasks can run in parallel on a single executor.
`--executor-memory`	Affects the ability to handle large micro-batches and shuffles in RAM.
`--conf "spark.sql.shuffle.partitions=2"`	Controls how many partitions are created during shuffles.

See full configuration: https://spark.apache.org/docs/latest/submitting-applications.html and general configurations: https://spark.apache.org/docs/latest/configuration.html. Also check possible configurations with:

```
spark-submit --help
```

Updated the Spark resources to improve streaming job performance and handle larger workloads

- Increased `--executor-cores` from 1 to 16 which allows for parallelized processing
- Increased `--executor-memory` from 1G to 4G which allows for more caching
- Kept `--num-executors` at 1 since it is only running on one pysical machine

Update docker compose

```
services:
  generator:
    image: adrianovogel/hgb-load-gen:latest
    extra_hosts:
      - "host.docker.internal:host-gateway"
    environment:
      - KAFKA_BROKER=host.docker.internal:9095
      - KAFKA_TOPIC=logs
      - TARGET_RPS=60000
      - ADDITIONAL_TERM=crash
      - ADDITIONAL_TERM_RATE=100
    deploy:
      replicas: 4  # This enables to run more instances (containers)
      resources:
        limits:
```

```yaml
          memory: 1024M
          cpus: '1'
    networks:
      - streaming-net

networks:
  streaming-net:
    external: true
    name: streaming-net
```

```yaml
  spark-worker:
    image: bitnamilegacy/spark:4.0.0
    depends_on:
      - spark-master
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
      - SPARK_WORKER_CORES=16
      - SPARK_WORKER_MEMORY=4G
    networks:
      - streaming-net
    deploy:
      replicas: 1
      resources:
        limits:
          memory: 4096M
          cpus: '16'
```

Updated parameters to use more compute power:

```
spark-submit \
  --master spark://spark-master:7077 \
  --packages org.apache.spark:spark-sql-kafka-0-10_2.13:4.0.0 \
  --num-executors 1 \
  --executor-cores 16 \
  --executor-memory 4G \
  /opt/spark-apps/spark_structured_streaming_logs_processing.py
```

Monitoring

Navigate to the **Structured Streaming Tab** in the UI to monitor the performance:

*** Input Rate vs. Process Rate:**

If your input rate is consistently higher than your process rate, your application is failing to keep up with the data stream.

1 Pages. Jump to 1 . Show 100 items in a page. Go

Run ID	Start Time ▾	Duration	Avg Input /sec	Avg Process /sec	Latest Batch
2ecdbbcf-7542-48e0-a4ff-8cb1637e15df	2026/01/14 22:15:03	11 minutes 5 seconds	193523.20	163582.66	7

1 Pages. Jump to 1 . Show 100 items in a page. Go

- Average input/sec: 193523.20
- Average process/sec: 163582.66

Unfortunatelly I was unable to push it any higher as windows started to lagg and working got really difficult.

The Executors Tab

In the The Executors Tab, check the **"Thread Dump"** and **"Task"** columns to verify resource utilization.

Executors

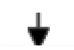
Show 20 entries Search:

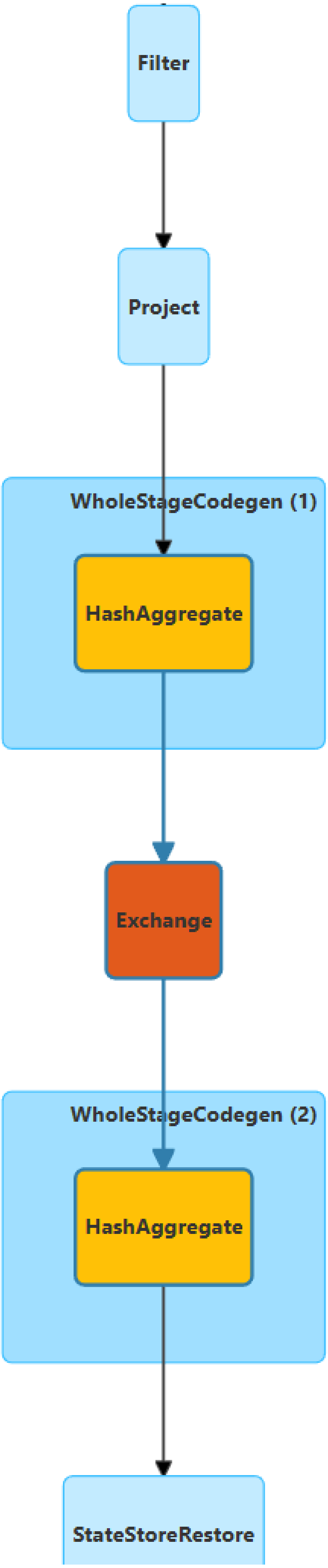
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump	Heap Histogram	Add Time
driver	b2af8b4106a3:35299	Active	0	2.7 MiB / 434.4 MiB	0.0 B	0	0	0	0	0	9.7 min (0.7 s)	0.0 B	0.0 B	0.0 B		Thread Dump	Heap Histogram	2026-01-14 23:14:56
0	172.22.0.5:44563	Active	0	2.6 MiB / 2.2 GiB	0.0 B	16	2	0	3812	3814	32 min (31 s)	0.0 B	233.1 KiB	144.1 KiB	stdout stderr	Thread Dump	Heap Histogram	2026-01-14 23:15:00

The SQL/Queries Tab

Click on the active query to see the **DAG (Directed Acyclic Graph)**.

- **Identify "Shuffle" Boundaries:** Look for the exchange points where data is redistributed across the cluster.

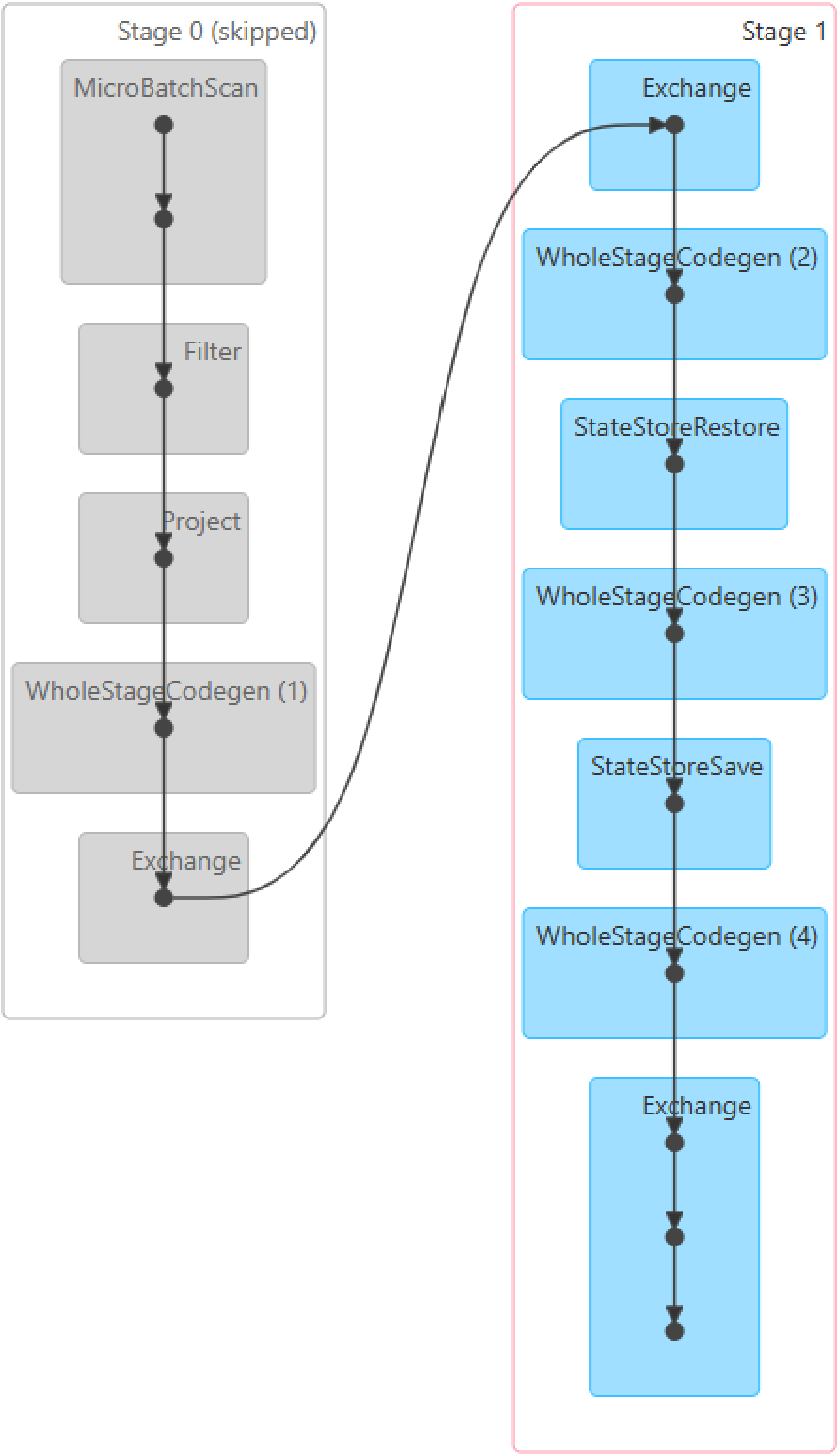



Filter
Project
WholeStageCodegen (1)
HashAggregate
Exchange
WholeStageCodegen (2)
HashAggregate
StateStoreRestore

- **Identify Data Skew:** Is data being distributed evenly across all your cores, or are a few tasks doing all the work? Use the DAG to pinpoint which specific transformation is causing a bottleneck.

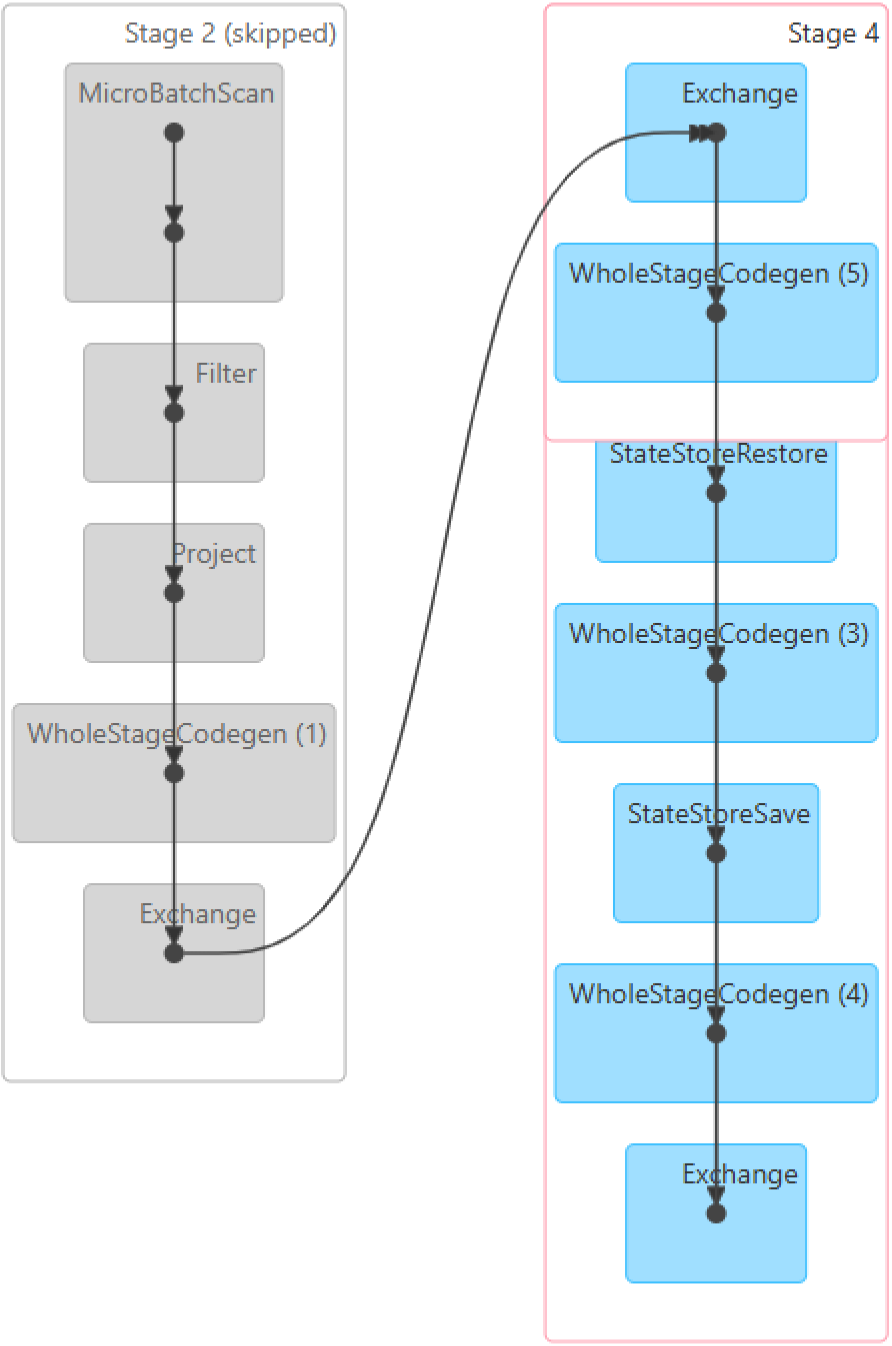
Job 0:

	Submitted	Duration	Tasks: Succeeded/Total	Inp
ails	2026/01/14 20:41:03	27 s	200/200	



Job 1:

	Submitted	Duration	Tasks: Succeeded/Total	Input
+details	2026/01/14 20:41:48	0.2 s	1/1	
+details	2026/01/14 20:41:30	18 s	200/200	



The load is somewhat equally distributed. The slower job 0 DAG features more exchange steps at the end compared to job 1.

- **Submit activities 1 and 2 (answers and evidences) via Moodle until 20.01.2026**