1 Introduction

To monitor a Spark Structured Streaming pipeline in production, using the StreamingQueryListener is the most robust way to capture lifecycle events. This allows you to log specific metrics (like processing rate, input rows, and latencies) as JSON for your logging infrastructure.
Here is how you can implement a custom listener in PySpark.

1.1 Creating the Custom Listener

You need to subclass StreamingQueryListener and override three main methods: onQueryStarted, onQueryProgress, and onQueryTerminated.

```
from pyspark.sql.streaming import StreamingQueryListener
import logging
import json

# Setup your logger (using the manual config we discussed earlier)
logger = logging.getLogger("StreamingMetrics")

class MyPipelineListener(StreamingQueryListener):

    def onQueryStarted(self, event):
        """Triggered when a query is started."""
        log_data = {
            "event": "QUERY_STARTED",
            "id": str(event.id),
            "runId": str(event.runId),
            "name": event.name
        }
        logger.info("Streaming query started", extra=log_data)

    def onQueryProgress(self, event):
        """Triggered every time a micro-batch completes."""
        progress = event.progress
        log_data = {
            "event": "QUERY_PROGRESS",
            "query_name": progress.name,
            "batch_id": progress.batchId,
            "input_rows_per_sec": progress.inputRowsPerSecond,
            "processed_rows_per_sec": progress.processedRowsPerSecond,
            "num_input_rows": progress.numInputRows,
            # Extracting specific duration metrics
            "trigger_execution_ms": progress.durationMs.get("triggerExec
            "source_latency_ms": progress.durationMs.get("getBatch")
        }
        logger.info("Batch processed", extra=log_data)

    def onQueryTerminated(self, event):
        """Triggered when a query stops or fails."""
        log_data = {
            "event": "QUERY_TERMINATED",
            "id": str(event.id),
            "runId": str(event.runId),
            "exception": event.exception
        }
        if event.exception:
            logger.error("Streaming query failed", extra=log_data)
        else:
            logger.info("Streaming query stopped gracefully", extra=log_
```

1.2 Registering the Listener in your Spark Session

You must attach the listener to the SparkSession before starting the streaming query.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("ProductionDataPipeline") \
    .getOrCreate()

# Create an instance of your custom listener
my_listener = MyPipelineListener()

# Add the listener to the Spark session
spark.streams.addListener(my_listener)
```

```
# Example: A simple streaming query
query = spark.readStream \
    .format("rate") \
    .load() \
    .writeStream \
    .format("console") \
    .queryName("RateLimitTest") \
    .start()

query.awaitTermination()
```

2 Understanding the Lifecycle Events

The StreamingQueryListener tracks the flow of data through your pipeline by listening for these three specific state changes:

onQueryStarted: Useful for auditing when a job begins and identifying the runId. This ID is critical for correlating logs if a job restarts.

onQueryProgress: This is the most "chatty" event. It provides a StreamingQueryProgress object. In production, you typically monitor inputRowsPerSecond to detect data spikes and processedRowsPerSecond to see if your pipeline is falling behind (backpressure).

onQueryTerminated: Crucial for alerting. If event.exception is not None, it contains the stack trace of why the pipeline crashed (e.g., lost connection to Kafka, schema mismatch).

2.1 What happens during "getBatch"?

Before Spark can actually transform any data, it has to perform a "handshake" with your source (like Kafka, S3, or Kinesis).

- **Checking for new data:** Spark asks the source, "What is the latest offset/file available?"
- **Determining the range:** It calculates the difference between what it has already processed and what is currently available.

- **Metadata overhead:** It records these offsets in the checkpoint directory to ensure fault tolerance.

3 Important : Streaming Query Duration Metrics

In Structured Streaming, **durationMs** is a map of micro-batch execution phases to their execution time in milliseconds, useful for fine-grained performance analysis. It is a python dictionary.

The following table describes the internal components of the `durationMs` dictionary captured by the `StreamingQueryListener`.

Key	**Phase**	**What it means**
`getBatch`	Source Reading	Time spent querying metadata and retrieving offsets from the source (e.g., Kafka or S3).
`latestOffset`	Offset Fetching	Time taken to contact the source to find the most recent available data offsets.
`queryPlanning`	Optimization	Time spent by the Spark analyzer and optimizer to generate the execution plan for the micro-batch.
`walCommit`	Checkpointing	Time taken to write the current batch's offset range to the Write-Ahead Log in the checkpoint directory.
`addBatch`	Processing/Sink	The duration of the actual data processing (transformations) and writing the output to the sink.
`triggerExecution`	Total Cycle	The cumulative time for the entire micro-batch to complete, from start to finish.

Table 1: Breakdown of Spark Streaming Duration Metadata

How to use the table above for Optimization:

If getBatch is the highest: You have a connectivity or metadata issue with your source (Kafka, S3, etc.).

If addBatch is the highest: Your Spark transformations are computationally heavy, or your destination (Sink) is slow.

If walCommit is the highest: Your checkpoint storage (usually S3 or HDFS) is experiencing high latency or throttling.

4 Some Special Cases

When your latency_source_ms (the getBatch phase) is large and the total triggerExecution exceeds your defined processingTime window (30 seconds), it creates a situation called **Batch Overrun.** Here, what happens when Batch Overrun takes place.

1. **The Current Batch MUST Finish**
Spark Structured Streaming will never abandon or drop rows just because a batch is taking too long. Even if your trigger is set to 30 seconds, if the batch takes 50 seconds to complete (due to high source latency or slow processing), Spark will stay on that batch until every single row is processed and committed to the sink.
Result: The rows are safe; they are not lost.

2. **The "Immediate Next-Trigger Rule**

If a batch takes 50 seconds (exceeding your 30s trigger), spark realizes it is behind the schedule. The moment the current batch finishes, it will **immediately** start the next batch. It does not wait for the next 30-second clock tick.

4.1 What should I do?

If latency_source_ms is the culprit for the overrun:

Check Source Partitioning: In Kafka, ensure you have enough partitions. Spark reads in parallel based on partitions.

Increase maxRatePerPartition: If the source is slow because it's fetching too much at once, limit how much Spark grabs per batch so it stays under the 30s limit.

Network/Metadata: If getBatch is high but numInputRows is low, your Spark cluster is struggling to talk to the source metadata (e.g., S3 listing or Kafka offset lookups).