

How to Detect Auto-Loader Failure in Databricks

How to Detect Auto Loader Failures (Step-by-Step)

Objective:

Monitor Auto Loader jobs and **detect errors** like:

- Missing or corrupted files
- Schema mismatches
- Write failures
- Source path access issues

Step 1: Wrap Your Streaming Query in a Listener

Use a custom **StreamingQueryListener** to detect and log any job failures or terminations:

```
python
CopyEdit
from pyspark.sql.streaming import StreamingQueryListener

class AutoLoaderErrorListener(StreamingQueryListener):
    def onQueryProgress(self, event):
        print("Batch processed successfully:", event.progress.batchId)

    def onQueryTerminated(self, event):
        if event.exception:
            print("🚨 Auto Loader FAILED!")
            print("Reason:", event.exception)
        else:
            print("Stream terminated normally.")

# Register the listener
spark.streams.addListener(AutoLoaderErrorListener())
```

Step 2: Define the Auto Loader Stream with a Try/Except Block

```
python
CopyEdit
from pyspark.sql.functions import input_file_name, current_timestamp, upper, col

try:
    df = (
        spark.readStream
            .format("cloudFiles")
            .option("cloudFiles.format", "csv")
            .option("cloudFiles.inferColumnTypes", "true")
            .option("cloudFiles.schemaLocation", "/mnt/data/autoloader/schema/people/")
            .load("/mnt/data/autoloader/incoming/")
    )

    df_transformed = (
        df.withColumn("FirstName", upper(col("FirstName")))
        .withColumn("ingestion_timestamp", current_timestamp())
    )
```

```
        .withColumn("source_file", input_file_name())
    )

    query = (
        df_transformed.writeStream
            .format("delta")
            .option("checkpointLocation", "/mnt/data/autoloader/checkpoints/people/")
            .outputMode("append")
            .start("/mnt/data/bronze/people/")
    )

except Exception as e:
    print("Auto Loader failed to start:", str(e))
```

Step 3: (Optional) Log Errors into a Delta Table

You can write the error to an error log table for review:

```
python
CopyEdit
from datetime import datetime

def log_autoloader_error(error_msg):
    error_df = spark.createDataFrame(
        [(str(error_msg), datetime.now())],
        ["error_message", "logged_at"]
    )
    error_df.write.mode("append").saveAsTable("error_logs.autoloader_errors")
```

Use it in the exception block:

```
python
CopyEdit
except Exception as e:
    log_autoloader_error(e)
```

Step 4: Monitor via Databricks Workflows (Optional)

If you're running Auto Loader as part of a **Databricks Job**, you can:

- Enable **Job failure alerts** via email or webhooks
 - Automatically **retry failed tasks**
 - Log job run details in the **Workflows UI**
-

Summary

Step	Action
1	Add a <code>StreamingQueryListener</code> to monitor progress & termination
2	Wrap your Auto Loader logic in <code>try/except</code> to catch runtime errors
3	Log errors into a Delta table (<code>error_logs.autoloader_errors</code>)
4	(Optional) Use Databricks Workflows for alerting and retries

When to Reset Auto Loader in Databricks

When to Reset Auto Loader

You might want to reset Auto Loader if:

1. A file was missed due to a crash or misconfiguration.
2. You updated logic (e.g., transformations or schema).
3. Files were partially written or failed mid-batch.
4. You want to **reprocess everything** from scratch.

What Resetting Actually Means

Resetting Auto Loader means:

- **Deleting the checkpoint folder** — so Spark **forgets what it already processed**.
- (Optional) **Deleting the output Delta table or path** — if you want to **fully reload the data**.

Important Warning

❗ Resetting the checkpoint will cause **all files in the input directory** to be **reprocessed**, even those already written to the target table — unless deduplication logic is used.

Step-by-Step: Reset Auto Loader

Folder Structure:

```

📁 /mnt/data/autoloader/incoming/
    ♦ Source files (CSV, JSON, etc.)

📁 /mnt/data/autoloader/schema/people/
    ♦ Schema info used by Auto Loader

📁 /mnt/data/autoloader/checkpoints/people/
    ├── commits/      ✅ Batch success logs
    ├── offsets/     📄 File read progress
    ├── metadata/    ⚙️ Job config and Spark info
    ├── sources/     👁️ Files seen per batch
    └── state/       🧠 Aggregation state (only if needed)

📁 /mnt/data/bronze/people/
    ♦ Delta table output (cleaned/transformed data)
```

The Databricks: **Auto-Loader** **Data Processed** vs. **Yet to Be Processed**

Config Path	Internally Controls or Maps To
source_path	sources/, offsets/ (files discovered + read)
schema_path	Maintains schema inference snapshots
checkpoint_path	Creates commits/, offsets/, sources/, etc.
target_path	Stores the final Delta data output

Assume your setup:

```
python
CopyEdit
source_path = "/mnt/data/autoloader/incoming/"
schema_path = "/mnt/data/autoloader/schema/people/"
checkpoint_path = "/mnt/data/autoloader/checkpoints/people/"
target_path = "/mnt/data/bronze/people/"
```

Once Auto Loader starts writing, this folder will contain:
Target_path is the **location on DBFS or cloud storage** where **processed data is written** .

```
/mnt/data/bronze/people/
├─ _delta_log/           # 📄 Transaction Logs for Delta table (JSON + checkpoint files)
├─ part-00000-...snappy.parquet # 📦 Actual data files in Parquet format
├─ part-00001-...snappy.parquet
└─ ...
```

Folder / File	Description
_delta_log/	Stores metadata about the table (transaction history, schema changes, commit logs)
*.parquet	Your actual Ingested and transformed data
(No checkpoint/ here) Checkpoint is stored separately under checkpoint_path, not here	

Register it as Delta Table , below is the run query:

```
CREATE TABLE bronze_people
USING DELTA
LOCATION '/mnt/data/bronze/people/'

SELECT * FROM bronze_people;
```

Summary

Term	Path	Stores What?
target_path	/mnt/data/bronze/people/	Delta table (data + metadata)
Contents	_delta_log/ + *.parquet	All processed data from Auto Loader pipeline

Note:

If the table is not registered, you can still read it using the **Delta table path**, but you **can't query it by name** in SQL or Unity Catalog.

You can't see data lineage in Unity Catalog if the table is **not registered** – lineage tracking only works for **registered tables** in a **Unity Catalog-enabled metastore**.

1. Stop the Streaming Job

You can do this from the notebook UI, Jobs tab, or by calling `.stop()` if you've assigned it to a variable:

```
python
CopyEdit
query = df_transformed.writeStream...start()
query.stop()
```

2. Delete the Checkpoint Directory

```
python
CopyEdit
dbutils.fs.rm(checkpoint_path, recurse=True)
```

This erases:

- commits/
- offsets/
- sources/
- state/

→ Spark no longer knows which files were already processed.

(Optional) 3. Delete the Output Delta Table

If you want to **rebuild from scratch**:

```
python
CopyEdit
dbutils.fs.rm(target_path, recurse=True)
```

Or if registered as a table:

```
sql
CopyEdit
DROP TABLE IF EXISTS bronze.people;
```

4. Re-run Your Auto Loader Job

Use the same logic from the document:

```
python
CopyEdit
from pyspark.sql.functions import current_timestamp, input_file_name, upper, col

df = (
    spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "csv")
        .option("cloudFiles.inferColumnTypes", "true")
        .option("cloudFiles.schemaLocation", schema_path)
        .load(source_path)
```

```
)

df_transformed = (
    df.withColumn("FristName", upper(col("FirstName")))
    .withColumn("ingestion_timestamp", current_timestamp())
    .withColumn("source_file", input_file_name())
)

df_transformed.writeStream \
    .format("delta") \
    .option("checkpointLocation", checkpoint_path) \
    .outputMode("append") \
    .start(target_path)
```

Optional: Add Deduplication to Avoid Double Processing

If you can't delete the output but still want to reprocess without duplicating:

```
python
CopyEdit
from pyspark.sql.functions import expr

df_deduped = df_transformed.dropDuplicates(["id", "source_file"])

# Or add a hash-based deduplication
```

Summary: Reset Guide

Step	Action
Stop the stream	.stop() or notebook control
Delete checkpoint	dbutils.fs.rm(checkpoint_path, recurse=True)
Delete output (optional)	dbutils.fs.rm(target_path, recurse=True)
Restart the job	Run your Auto Loader pipeline again

Pros of Unregistered (Path-Based) Tables

Advantage	Description
Simple setup	No need to create a table in metastore
Flexible for ad-hoc use	Useful for temporary or dev pipelines
Easier testing	You can write and delete freely without affecting production tables
Still fully readable	Use .read.format("delta").load("/path/") in PySpark
Portable	Can be copied or moved without breaking SQL references

Cons of Not Registering

Limitation	Description
No SQL name	Can't use SELECT * FROM table_name — must use path
No Unity Catalog support	No lineage, governance, or RBAC features
Harder collaboration	Other users must know the exact path to access
No lineage tracking	Data lineage graph in Unity Catalog won't include it
No data discovery	Won't show up in catalogs, schemas, or SHOW TABLES
Prone to accidental overwrite	No metastore protection for concurrent writes

Trainee Q&A: How Auto Loader Handles Files Across Batches

- Mentor (IT Architect)
- Trainee (Junior Data Engineer)

Auto Loader is a high-performance, scalable file ingestion tool provided by Databricks for ingesting new data files from cloud storage (like AWS S3, Azure Data Lake, or Google Cloud Storage) **automatically and incrementally** using **Databricks Structured Streaming**.

Purpose:

- Watches a directory for **new files only** — no need to reprocess old data.
- Supports **schema inference** and **evolution**.
- Handles **millions of files** efficiently using file notification services.
- Works with formats like **CSV, JSON, Parquet, Avro**, and more.
- Ideal for implementing **Bronze layer ingestion** in Delta Lake.

Trainee

Hey, I've been exploring Auto Loader in Databricks, and I've got a question. Let's say we have 5 files to ingest — how exactly are they represented in the checkpoint directory?

Specifically in commits/ and sources/ — does sources track all 5 files or only the ones from the current micro-batch?

Mentor

That's a great question! Auto Loader organizes metadata in a very structured way — and understanding it will help you debug and optimize streaming jobs efficiently.

Let me break it down for you. There are **two possible ways** those 5 files could be processed, depending on when they arrive and your stream's trigger setting.

Mentor

Option A: All 5 files arrive together (one micro-batch)

If all 5 files are discovered **before the next trigger fires**, Auto Loader groups them into a **single batch**.

You'll see:

- commits/0 → indicating **batch 0 was committed**
- sources/0 → listing all files seen in this batch:

```
json
CopyEdit
"seenFiles": [
  "people_1.csv",
  "people_2.csv",
  "people_3.csv",
  "people_4.csv",
  "people_5.csv"
]
```

So in this case, sources/0 alone shows all 5 files — and they were all committed in batch 0.

Trainee:

Got it — so if all files arrive close together, one batch can handle all of them. What if the files arrive at different times?

Mentor

Option B: Files arrive at different times (multiple batches)

Let's say:

- 3 files arrive first,
- and 2 more come in **10 seconds later**.

With a typical trigger interval (like every 10 seconds), Spark will process them in **two separate batches**:

- commits/0 → batch 0
 - sources/0 → "seenFiles": ["people_1.csv", "people_2.csv", "people_3.csv"]
- commits/1 → batch 1
 - sources/1 → "seenFiles": ["people_4.csv", "people_5.csv"]

So the files are **split across multiple sources/N files**, but all 5 are still tracked.

Trainee:

Oh, I see. So even though they're split across batches, the total seen files still add up to 5. Makes sense!

Final Takeaway

Folder	What It Reflects
commits/	Number of micro-batches (e.g., 0, 1, 2...)
sources/	Tracks the files seen in each batch
Total seenFiles	Equals all files that were actually processed

So, even if you process 5 files over 1, 2, or 10 batches, the total number of seenFiles across sources/ will still be 5.

Trainee:

That clears it up perfectly! So it's the **batches** that control the file grouping, not the number of files per se.

Mentor:

Exactly! It's **time-based batching**, not file-count-based. And once you understand how commits/ and sources/ align, it's much easier to track and debug your Auto Loader pipelines.

Appendix:

Sample Auto Loader Code

```
python
CopyEdit
from pyspark.sql.functions import current_timestamp, input_file_name, upper, col

# Read data using Auto Loader
df = (
    spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "csv") # File format
        .option("cloudFiles.inferColumnTypes", "true") # Infer schema
        .option("cloudFiles.schemaLocation", "/mnt/data/autoloader/schema/employee/") # Where schema is stored
        .load("/mnt/data/autoloader/incoming/") # Input folder
)

# Apply transformations
df_transformed = (
    df.withColumn("department", upper(col("department")))
        .withColumn("ingestion_timestamp", current_timestamp())
        .withColumn("source_file", input_file_name())
)

# Write to Delta Lake with checkpointing
(
    df_transformed.writeStream
        .format("delta")
        .option("checkpointLocation", "/mnt/data/autoloader/checkpoints/employee/") # Required
        .outputMode("append")
        .start("/mnt/data/bronze/employee") # Bronze table path
)
```

AutoLoader

Config : Without Trigger	Config : With Trigger																		
<pre>df.writeStream \ .format("delta") \ .option("checkpointLocation", "/mnt/checkpoints/people") \ .start("/mnt/data/delta/bronze_people")</pre>	<pre>df.writeStream \ .format("delta") \ .option("checkpointLocation", "/mnt/checkpoints/people") \ .trigger(processingTime="5 seconds") \ .start("/mnt/data/delta/bronze_people")</pre>																		
Behavior: <table border="1"> <thead> <tr> <th>Time</th><th>Action</th></tr> </thead> <tbody> <tr> <td>00:00</td><td>Spark sees people_1.csv and processes it in batch 0 immediately</td></tr> <tr> <td>00:03</td><td>Spark detects people_2.csv and runs batch 1</td></tr> <tr> <td>00:06</td><td>Spark sees people_3.csv and starts batch 2</td></tr> <tr> <td>...</td><td>Spark polls continuously with no delay</td></tr> </tbody> </table>	Time	Action	00:00	Spark sees people_1.csv and processes it in batch 0 immediately	00:03	Spark detects people_2.csv and runs batch 1	00:06	Spark sees people_3.csv and starts batch 2	...	Spark polls continuously with no delay	Behavior: <table border="1"> <thead> <tr> <th>Time</th><th>Action</th></tr> </thead> <tbody> <tr> <td>00:00</td><td>Spark sees people_1.csv and starts batch 0</td></tr> <tr> <td>00:05</td><td>No new files → batch runs, nothing processed</td></tr> <tr> <td>00:10</td><td>Sees people_2.csv and people_3.csv (if they arrived) → batch 1 processes them together</td></tr> </tbody> </table>	Time	Action	00:00	Spark sees people_1.csv and starts batch 0	00:05	No new files → batch runs, nothing processed	00:10	Sees people_2.csv and people_3.csv (if they arrived) → batch 1 processes them together
Time	Action																		
00:00	Spark sees people_1.csv and processes it in batch 0 immediately																		
00:03	Spark detects people_2.csv and runs batch 1																		
00:06	Spark sees people_3.csv and starts batch 2																		
...	Spark polls continuously with no delay																		
Time	Action																		
00:00	Spark sees people_1.csv and starts batch 0																		
00:05	No new files → batch runs, nothing processed																		
00:10	Sees people_2.csv and people_3.csv (if they arrived) → batch 1 processes them together																		
<p>⇒ 3 batches for 3 files</p> <p>⇒ Latency = as fast as Spark can respond</p>	<p>⇒ Only 2 batches for 3 files</p> <p>⇒ Latency = bounded by 5-second interval</p>																		

Comparison Table

Feature	Without Trigger	With .trigger(processingTime="5s")
Trigger	Default (as fast as possible)	Fixed 5-second interval
Batch Count	3 batches (1 per file)	2 batches (grouped by time)
Latency	Lower (real-time)	Medium (5s delay max)
Resource Efficiency	High CPU usage per file	More efficient grouping
Control over behavior	No	Yes
Use in Production	Can be noisy / expensive	More predictable

Summary

- **Without .trigger()** = lower latency, high responsiveness, but may create too many small batches.
- **With .trigger()** = more control, better performance, lower cost at scale.
- Choose based on:
 - **Latency sensitivity** (alerts? dashboards?)
 - **Cost and throughput**
 - **File arrival pattern**

Final Note:

Auto Loader = smart, incremental ingestion from cloud storage with schema management and fault tolerance built-in. It's the recommended way to build the **Bronze layer** in modern data lakehouses using Delta Lake.

Data Processed	Yet to Be Processed
Files that Auto Loader has discovered, read, and successfully written to the target table (e.g., Delta Lake). These are tracked in the checkpoint under commits/, offsets/, and sources/.	Files that are newly arrived in the input directory but haven't been picked up by Auto Loader in any completed micro-batch yet.
<ul style="list-style-type: none"> • Already part of a committed batch • Logged in checkpoint metadata • Will not be reprocessed unless: • Checkpoint is deleted/reset • Source is modified manually 	<ul style="list-style-type: none"> • Discovered only if they appear in seenFiles in a future batch • Not yet included in commits/ or offsets/ • Will be automatically picked up in the next streaming batch
<i>Example: people_1.csv appears in sources/0 and commits/0 → processed.</i>	<i>Example: people_4.csv was added to the folder after batch 0 completed → will be picked up in batch 1.</i>

Final Analogy

Term	Think of it as...
Data Processed	Checked in at airport & on the plane
Yet to Be Processed	Still waiting in line at security

Ref: [checkpoint-in-databricks](#)