Databricks: The Complex Task Listed Below

**1. Ingestion Layer**

**Goal:** Efficiently bring structured/semi-structured/unstructured data from diverse sources into the platform.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Auto-ingestion with Auto Loader | Configure schema inference + checkpointing with cloudFiles API for millions of files. |
| CDC ingestion (Change Data Capture) | Use tools like Debezium or AWS DMS to stream changes incrementally into Delta Lake. |
| Multi-format ingestion (CSV, Avro, Parquet, JSON, XML) | Implement dynamic schema parsers with error handling, using Spark's DataFrameReader. |
| Schema evolution and enforcement | Handle column changes gracefully using mergeSchema, enforce schema validation. |
| IoT and streaming data handling | Build real-time ingestion from Kafka or Event Hubs using Structured Streaming. |
| Ingestion from legacy systems | Use JDBC connectors with predicates pushdown and partitioning for bulk extraction. |

**2. Raw/Bronze Layer**

**Goal:** Persist raw data in its native format with metadata and lineage tracking.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Add ingestion metadata | Automatically tag each row with source, ingestion time, batch ID. |
| Data lineage tracking | Integrate Unity Catalog or Delta Lake features to trace data origin. |
| Handling corrupt records | Use \_corrupt\_record column, redirect malformed data to quarantine tables. |
| Ingest log/event-heavy data | Design optimized schema with nested/struct columns and partition strategies. |
| Large-scale file consolidation | Use repartition and coalesce to balance file size (optimize shuffle). |

**3. Cleansed/Silver Layer**

**Goal:** Refine raw data by applying transformations, standardizations, and business logic.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Surrogate key generation | Create consistent keys across sources using hash functions or UUIDs. |
| De-duplication and versioning | Use window functions or Delta Lake MERGE logic. |
| Data standardization | Normalize units, currency, date formats across different source systems. |
| Slowly Changing Dimensions (SCD) Type 1/2 | Implement SCD logic using Delta MERGE INTO and UPDATE operations. |
| Business rule validations | Build a rules engine that dynamically applies validation logic per dataset. |
| Error segregation and logging | Build a reusable pipeline to route failed records with clear diagnostics. |

**4. Curated/Gold Layer**

**Goal:** Model data for analytics, reporting, and ML in an optimized and governed format.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Star/snowflake schema implementation | Design dimensional models with fact/dim tables optimized for BI tools. |
| Time-travel & audit queries | Leverage Delta Lake’s VERSION AS OF and TIMESTAMP AS OF. |
| Incremental aggregation | Maintain running totals, averages, and counts using Delta Lake updates. |
| KPI snapshotting | Persist periodic metric snapshots using orchestration. |
| Building semantic models | Create clean views and summary tables for Power BI/Tableau. |
| Partitioning and Z-Ordering | Apply multi-column partitioning and ZORDER by high-cardinality columns. |

**5. Orchestration Layer**

**Goal:** Ensure reliable, traceable, and performant execution of pipelines.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Dynamic pipeline generation | Use notebooks with parameterized widgets and orchestration with dbutils. |
| DAG dependency management | Build complex workflows using Databricks Workflows with custom triggers. |
| Retry logic and error recovery | Configure retries + logic to resume failed jobs from last checkpoint. |
| Event-driven workflows | Trigger jobs from cloud storage events, Git commits, or API calls. |
| Delta Live Tables (DLT) | Create declarative pipelines with built-in monitoring, quality rules. |

**6. Governance Layer**

**Goal:** Manage security, compliance, and data access at scale.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Unity Catalog implementation | Centralized access control and metadata across workspaces. |
| Row-level and column-level security | Use dynamic views + workspace identities for fine-grained control. |
| Tag-based access control | Implement data masking using tags (e.g., PII, confidential). |
| Audit logging and lineage | Enable Unity Catalog's audit logs + monitor usage with system tables. |
| Cross-region, cross-cloud security | Configure secure data sharing with Delta Sharing and token-based auth. |

**7. Serving Layer**

**Goal:** Expose data to end users, applications, and dashboards.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Low-latency APIs with MLflow or Flask | Serve real-time models/data via Databricks-hosted endpoints. |
| Power BI DirectQuery performance tuning | Push down queries, cache semantic models, use Composite Models. |
| Dashboard publishing with DBSQL | Build native dashboards using Databricks SQL Warehouses. |
| Multi-tenant data delivery | Implement data isolation via RLS and workspace separation. |
| Data sharing via Delta Sharing | Expose live Delta tables to external consumers securely. |

**8. Monitoring & Optimization Layer**

**Goal:** Ensure reliability, performance, and cost-efficiency.

|  |  |
| --- | --- |
| **Task** | **Description** |
| Cluster tuning for performance | Optimize autoscaling, worker types, and spot instance use. |
| Job monitoring + alerting | Implement alert policies, log aggregation with ML-based anomaly detection. |
| Cost tracking per team/project | Tag compute jobs, track spend with tools like FinOps or Azure Cost Mgmt. |
| Data quality observability | Integrate Great Expectations or expectations in DLT pipelines. |
| File compaction and optimization | Automate small file cleanup using Optimize + Vacuum. |

|  |
| --- |
| Checkpoint Directory: /mnt/data/autoloader/checkpoints/people/ After you start your streaming job with .option("checkpointLocation", "/mnt/data/autoloader/checkpoints/people/"),  Spark will create a structure like this:  CopyEdit  /mnt/data/autoloader/checkpoints/people/  ├── commits/ # Records successful micro-batch completions  ├── offsets/ # Tracks which data (files/rows) has been processed  ├── metadata/ # Stores job-level and query configuration info  ├── sources/ # Logs discovered files and input source metadata  └── state/ # Maintains internal state for aggregations or windowing 1. /commits/ – Stores info about completed micro-batches **Filename:** 0  **Contents:**  json  CopyEdit  {  "id": 0,  "timestamp": "2025-07-07T04:20:00.123Z",  "message": "Batch committed successfully"  }  This tells Spark: "Batch 0 completed successfully."  Note: When Spark Structured Streaming runs (including Auto Loader), it **divides work into micro-batches**. Each batch processes a group of files (or even one file, depending on trigger and availability). 2. /offsets/ – Tracks which files have already been processed **Filename:** 0  **Contents:**  json  CopyEdit  {  "0": {  "latestOffset": {  "sourceVersion": 2,  "inferredSchema": {  "type": "struct",  "fields": [  { "name": "id", "type": "integer" },  { "name": "first\_name", "type": "string" },  { "name": "last\_name", "type": "string" },  { "name": "email", "type": "string" }  ]  },  "seenFiles": [  {  "path": "dbfs:/mnt/data/autoloader/incoming/people\_1.csv",  "timestamp": 1720357200000  }  ]  }  }  }  This tells Spark which files it has already seen and processed. 3. /metadata/ - Job and engine config **Filename:** 0  **Contents:**  json  CopyEdit  {  "id": 0,  "sparkVersion": "13.3.x-scala2.12",  "timestamp": "2025-07-07T04:19:55.000Z",  "trigger": {  "type": "ProcessingTime",  "intervalMs": 10000  }  }  This logs what version of Spark ran the stream and how often it was triggered. 4. /sources/ – Source-specific metadata (like for cloudFiles) **Filename:** 0  **Contents:**  json  CopyEdit  {  "description": "FileStreamSource[file:/mnt/data/autoloader/incoming]",  "startOffset": null,  "endOffset": {  "seenFiles": [  "people\_1.csv",  "people\_2.csv"  ]  }  }  This tracks what files were seen between offsets. 5. /state/ – Internal processing state (used in aggregations/window ops) This folder contains internal binary state files Spark uses to resume complex stateful operations. These are:   * Not human-readable (binary encoded) * Only needed for advanced streaming like joins, aggregations, or watermarks  Optional Clean-Up Tip If you want to restart the stream **fresh**, you can safely delete the whole checkpoint folder:  bash  CopyEdit  dbutils.fs.rm("/mnt/data/autoloader/checkpoints/people/", recurse=True)  But only do this if you want to **reprocess all files** from scratch |

.

**Scenario**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assuming there are 5 files to ingest, how are they represented across the commits and sources directories in the checkpoint? Specifically, does the sources folder track all 5 files or only those seen in the current micro-batch? How Auto Loader Processes ThemOption A: All 5 files processed in ****one micro-batch****  * If all 5 files are discovered before the next trigger interval, they are bundled into a **single batch**.   **Result:**   * commits/0 → batch 0 committed * sources/0 → contains:   json  CopyEdit  "seenFiles": [  "people\_1.csv",  "people\_2.csv",  "people\_3.csv",  "people\_4.csv",  "people\_5.csv"  ]  ➡ seenFiles = 5 files Option B: Files processed across ****multiple batches**** Let’s say 3 files came first, then 2 more came 10 seconds later.  **Result:**   * 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"]   ➡ Total across batches = seenFiles = 3 + 2 = 5 files Final Answer:  |  |  | | --- | --- | | **Folder** | **File Count Reflects** | | commits/ | Number of **micro-batches** (not files) | | sources/ | Each file = **files seen in that batch** | | Total seenFiles across sources | = All files that were processed |   So, **if 5 files are processed**, the seenFiles across all sources/N files will total **5 files**, regardless of how many batches. |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| At any given point in time, irrespective of how many files arrive, Auto Loader can process them together as a single micro-batch. As far as I understand, there is no hard limit on the number of files per batch — it's determined by the trigger interval and system capacity, not by file count. Key Clarifications:  * **Correct**: Spark Auto Loader groups **all discovered files** within the current trigger interval into a **single batch**. * There is **no fixed limit** like “only 3 files per batch”. * However, practical constraints like:   + **cluster resources**   + **file discovery latency**   + **cloud storage throughput**   may affect how many files can actually be read in time. Optional Rewording (More Concise): "Auto Loader batches are time-based, not file-count limited — any number of files discovered within a trigger interval are processed together as one batch."  We need to define as  .writeStream  .trigger(processingTime="10 seconds")   * Every **10 seconds**, Spark will **look for new files**. * Any files discovered in that 10-second window are processed together as **one batch**.  Important Notes:  |  |  | | --- | --- | | **Concept** | **Explanation** | | Time-based batch | Spark uses the **trigger interval** (e.g., 10 seconds) to decide when to run a batch | | No file count limit | Spark doesn’t care how many files — 1 or 1000 — if it finds them during that window, it processes them | | You define the time trigger | Using .trigger(processingTime="xx seconds") in the writeStream | | Default (if not defined) | If you don’t set .trigger(), Spark uses the **default continuous** behavior and schedules batches as fast as it can |  Example: Let’s say:   * You set .trigger(processingTime="30 seconds") * And 10 files arrive within that time   ➡Spark creates **one batch** that processes all 10 files. Summary  * Yes, Auto Loader **batches are time-based**, not file-count-based * You can (and should) define the trigger interval using .trigger(...) * This gives you **control over latency vs throughput** |

What if we not mentioned / defined trigger for Auto Loader

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| What happens if you ****don't define a trigger****? If you **don’t explicitly set .trigger(...)**, Spark uses the **default trigger**, which is called:  **"micro-batch mode with default interval"**  This means: Spark will:  * Run **as fast as possible** * Automatically schedule the **next batch** as soon as the previous batch completes   This behavior is often called **“run continuously”** or **“process as fast as engine allows”**.  df.writeStream \  .format("delta") \  .option("checkpointLocation", "...") \  .start("...") # Stream Starts   * Keep checking for new files * Immediately process them in a batch as soon as it can * Wait for the next files * Repeat..  Pros and Cons of ****not defining a trigger****  |  |  | | --- | --- | | **✅ Pros** | **❌ Cons** | | Low latency | Can overload cluster if files arrive rapidly | | Easy for continuous ingestion | No built-in pause between batches | | Ideal for near real-time use | No control over compute cost/performance |  How does this compare to defining a trigger?  |  |  | | --- | --- | | **Mode** | **Behavior** | | .trigger(processingTime="10 seconds") | Runs a batch **every 10 seconds** | | (No trigger defined) | Runs batches **as fast as it can** |  Best Practice  |  |  | | --- | --- | | **Use case** | **Trigger Recommendation** | | Production / predictable loads | ✅ Define .trigger(processingTime="xx sec") for controlled behavior | | Development / testing | ❌ OK to skip trigger — default mode is fast | | Low-latency pipeline | ✅Use no trigger or short interval (1–5s) |  Summary  * **Not defining .trigger()** → Spark uses **default “process ASAP” mode** * You **can control batch frequency** by setting .trigger(processingTime="xx seconds") * It’s optional, but defining it helps control **resource usage and job behavior** |

Scenario : Trigger Vs No Trigger

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test Scenario SetupIngest Path: /mnt/data/autoloader/incoming/ Files (3): CopyEdit  people\_1.csv – arrives at 00:00  people\_2.csv – arrives at 00:03  people\_3.csv – arrives at 00:06 Target Table: /mnt/data/delta/bronze\_people Auto Loader Config 1: ****Without**** .trigger() python  CopyEdit  df.writeStream \  .format("delta") \  .option("checkpointLocation", "/mnt/checkpoints/people") \  .start("/mnt/data/delta/bronze\_people") Behavior:  |  |  | | --- | --- | | **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** |   ➡️ **3 batches for 3 files** ➡️ Latency = **as fast as Spark can respond** Auto Loader Config 2: ****With**** .trigger(processingTime="5 seconds") python  CopyEdit  df.writeStream \  .format("delta") \  .option("checkpointLocation", "/mnt/checkpoints/people") \  .trigger(processingTime="5 seconds") \  .start("/mnt/data/delta/bronze\_people") Behavior:  |  |  | | --- | --- | | **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 |   ➡️ **Only 2 batches for 3 files** ➡️ Latency = **bounded by 5-second interval** 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** |