**Objective:**

This document aims to explore methods for retrieving audit logs within the Generic Workflow Framework. The audit process focuses on capturing and tracking job execution details at the table level while ensuring data integrity, transparency, and enabling real-time monitoring and performance tracking across various workflow layers.

**Audit Table Analysis:**

As part of the Generic Workflow Framework, we are analyzing the design of an audit table/view to evaluate methods for comprehensive tracking and monitoring.

1. **Job\_Task\_Audit (Table-Level Audit Table)**

* Examining how execution logs can be captured at the table level to track individual task and child task runs within a workflow.
* Exploring ways to store and retrieve data ingestion details and processing status for each table to ensure accurate auditing.
* The required details should be defined according to the schema below.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Job\_id** | **Job\_Run\_id** | **Layer** | **Task\_Run\_id** | **Table\_Name** | **Start\_Date\_Time** | **End\_Date\_Time** | **Status** | **Error** |
|  |  |  |  |  |  |  |  |  |

**Raw Layer (Non-DLT) -**

**Attempting to Retrieve All Required Table-Level Run Details from Databricks SDK (Workspace Client)**

In this approach, we are leveraging the Databricks SDK (Workspace Client) to retrieve table-level execution details for each pipeline run. The objective is to extract structured metadata at the table level.

To achieve this, we are:

* Using Databricks SDK's Workspace Client (databricks.sdk.WorkspaceClient) to fetch job run metadata.
* Utilizing client.jobs.list\_runs(), client.jobs.get\_run(), and client.jobs.get\_run\_output() to capture job-level execution details.
* Extracting resolved input parameters (e.g., config\_file) from resolved\_values in foreach loops to accurately map table-level execution details.
* Finally, combining all extracted details into a temporary view with the predefined audit schema.
* Sharing the notebook link where we have extracted the necessary audit details at table level using this approach.

[https://adb-4661267993302765.5.azuredatabricks.net/editor/notebooks/2439266503131672?o=4661267993302765#command/2439266503131685](https://adb-4661267993302765.5.azuredatabricks.net/editor/notebooks/2439266503131672?o=4661267993302765%23command/2439266503131685)

**Bronze Layer(DLT) -**

**Approach 1:** Utilizing Published Event Logs for Audit Tracking

* Databricks now enables publishing DLT pipeline event logs to the Metastore, making them accessible to all users. This overcomes the limitation of directly accessing the event log API, which is restricted to the respective pipeline owner.
* We can retrieve table-level execution details at the Bronze layer from the published event logs by leveraging **DLT Pipeline Id** and **DLT Run ID**.
* To retrieve logs from all DLT pipelines and consolidate them into a view or a designated table following the prescribed schema (Job\_Task\_Audit table), the pipeline ID must be dynamically passed in the query. Since SQL does not support dynamic referencing of published event log table names, we use Python to handle this by setting a variable.
* The view must be refreshed whenever a new DLT pipeline is created to ensure it includes logs from newly published event logs.

**Example –**

****

**How Event logs get published?**

* The owner of the pipeline can publish the event log as a public Delta table by toggling the Publish event log to metastore option in the **Advanced** section of the pipeline configuration. We can optionally specify a new table name, catalog, and schema for the event log. Refer the below screenshot.

A screenshot of a computer

AI-generated content may be incorrect.

* When specifying the table name, schema, and catalog, a physical Delta table is created in Unity Catalog. Logs can then be extracted from this table and written to the actual audit table, as demonstrated in Approach 2.
* If a new table name, catalog, and schema are not specified, then a table named as ‘event\_log\_<pipeline\_id>’ will get created in the default schema and catalog named set in the pipeline configuration.

A screenshot of a computer

AI-generated content may be incorrect.

**Approach 2:** Implementing custom audit logic

**A white rectangular box with black text

AI-generated content may be incorrect.**

* The goal is to track and log table level execution details, including Job\_id, Job\_Run\_id, Task\_Run\_id, Table\_Name, Start\_Date\_Time, End\_Date\_Time, Status, and Errors encountered during the DLT pipeline execution. These details will be captured dynamically and stored in the Job\_Task\_Audit table.
* Extract job-level metadata such as job\_id, job\_run\_id, and task\_run\_id from Databricks pipeline configurations. These values in the pipeline configuration will be dynamically updated with the current run IDs in each run by making an API call from the Bronze notebook task before triggering the DLT execution.
* Record timestamps for Start\_Date\_Time and End\_Date\_Time when the execution begins and ends. Capture the Table\_Name being processed and its corresponding status. Log any errors encountered during execution.
* In this approach, we need to maintain two tables: one DLT table and one Delta table. The DLT table is required because, in the event of a failure, the pipeline will not run unless at least one active flow exists in the DLT pipeline run.
* The DLT table stores the latest audit information for the current run and then data from the DLT table gets written to the non-DLT table.

**Limitations in this approach:**

* If a DLT pipeline fails before 'setting up the table' (due to syntax or cluster issues), logging the event in the audit table is not possible.
* DLT table for each pipeline. This is because a DLT table created in one pipeline cannot be used in another.
* We cannot insert logs of every DLT pipeline directly into one DLT table, so a non-DLT table is used instead to combine the logs of all DLT pipelines.

**Conclusion and Next Steps:**

The above approaches for the Raw and Bronze layers present analyzed observations on retrieving logs using Databricks' existing logging mechanisms and custom logic at the Bronze layer. Once the final audit process requirements are clearly defined, we can decide how to incorporate these approaches—whether by consolidating logs into a Delta audit table, a view, or another database—as part of the framework.