**Objective:**

The objective of this document is to define the implementation of auditing process within the Generic Workflow Framework. This audit process will capture and track job execution details at table level. The implementation ensures data integrity, transparency, and enables real-time monitoring and performance tracking across different workflow layers.

**Audit Table Design:**

The **Generic Workflow Framework** maintains an audit table to ensure comprehensive tracking and monitoring:

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

* Maintains detailed execution logs at the table level, tracking individual task & child task runs within a workflow.
* Stores information on data ingestion, and processing status for each table.

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

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

**Approach 1:** Attempting to retrieve all the required table-level run details from databricks system table.

* Audit details for child tasks within a for-each task can be retrieved from the 'system.lakeflow.job\_task\_run\_timeline' table. However, table names are not visible, and the details may take a few hours to become available.
* While table-related information is available in the 'system.access.table\_lineage' table, it cannot be correlated with a specific child\_task\_run\_id of a for-each task.

**Approach 2:** Attempting to retrieve all the required table-level run details from Databricks REST APIs.

* Attempted to retrieve table-level audit details through an API call by accessing the input parameters passed to the foreach child task. The goal was to extract the JSON config file name corresponding to the table name. However, the resolved **config\_file** parameter could not be fetched.

**Goal -** A screenshot of a computer

AI-generated content may be incorrect.

**API Response -**

A screenshot of a computer

AI-generated content may be incorrect.

**APIs used -** /api/2.2/jobs/runs/get

* Attempted to use the /api/2.2/jobs/runs/get-output API to retrieve the output for a single run. This API allows fetching the output and metadata of an individual task run when a notebook task returns a value via dbutils.notebook.exit().
* However, this API only supports retrieving the output of a single task at a time, as it requires a Task\_Run\_Id rather than a Job\_Run\_Id.
* Since at Raw Layer we will multiple sub child Task runs, this limitation prevents retrieving outputs for all child tasks in a single API call.
* Explored multiple Databricks workflow APIs and reviewed various documentation sources, but none provided the necessary details at the table level required for capturing logs.

**Approach 3:** Implement custom audit logic in the Ingest\_notebook for the Raw layer to capture table-level execution logs in **Job\_Task\_Audit**.

**Bronze Layer(DLT) Implementation**

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

* Databricks now allows publishing DLT pipeline event logs to the Metastore, making them accessible to all users.
* We can create a view that merges audit logs from the raw layer at the table level with the logs available in the published event logs using DLT\_Pipeline\_Id and DLT\_Run\_ID as joining keys to correlate raw layer logs with DLT pipeline event logs.
* To retrieve logs from all DLT pipelines, the pipeline ID must be passed dynamically in the query. Since dynamically referencing the published event log table name is not feasible in SQL, we use Python to manage this dynamically 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.

**Note –** To implement this approach, a custom audit logic is required at the raw layer to capture current **DLT\_Pipeline\_id** and **DLT\_Run\_id**, which are essential for retrieving the necessary details from the published event logs.

**Example –**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**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:** Leveraging the Event Log API

Generic Workflow:

A diagram of a diagram

AI-generated content may be incorrect.

Based on the above generic workflow, the following explains the Bronze notebook task and its role in implementing the audit mechanism for this approach.

A flowchart of a project

AI-generated content may be incorrect.

* After triggering the DLT pipeline, the Bronze Notebook task will remain in a running state until the execution of the DLT pipeline completes.
* This will be achieved by continuously polling the DLT API in a loop to monitor the pipeline's status.
* Once the DLT pipeline execution ends, logs for the current run will be retrieved from the published event logs.
* The extracted logs will then be written to the Job\_Task\_Audit table from Bronze notebook.

**Limitations of Approach 2:**

* The Bronze notebook task will remain in a running state until the associated DLT pipeline completes execution. This leads to additional costs as one cluster remains active for monitoring the DLT pipeline run, while another cluster runs the actual DLT pipeline execution.
* However, a workaround exists by implementing a separate process to capture audit logs after a defined delay, such as a few hours post-ingestion workflow completion, including the DLT run. The timing of this process depends on how the audit table is utilized—whether real-time data is necessary immediately after the workflow run or if a delay of a few hours or a day is acceptable.

**Approach 3:** 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.

**Note –** This is currently in the exploration and testing phase. There are multiple challenges in implementing this within DLT, which we are actively investigating and testing to identify the most efficient approach.

**Error While Accessing Event Logs**

* Encountering an internal error when trying to access the event log of recently created DLT pipelines.

**A computer screen shot of a computer error

AI-generated content may be incorrect.**