GOTO: Serverless Extract-Transform framework for a workflow orchestrator

|  |  |  |  |
| --- | --- | --- | --- |
| Author | Email ID | Version | Modified date |
| Kishen Das Kondabagilu Rajanna | kishen.das@gmail.com | 0.01 | 02/08/2024 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[Workflow Orchestrator 2](#_Toc158302756)

[Why the name GOTO? 3](#_Toc158302757)

[High-level Architecture 3](#_Toc158302758)

[DAG Registration Architecture 3](#_Toc158302759)

[DAG RUN architecture 4](#_Toc158302760)

[Serverless 5](#_Toc158302761)

[Directed Acyclic Graph (DAG) 5](#_Toc158302762)

[DAG Register 5](#_Toc158302763)

[SLA (Service-Level Agreement) 6](#_Toc158302764)

[Time to completion 6](#_Toc158302765)

[Cost to completion 7](#_Toc158302766)

[Registering a DAG 7](#_Toc158302767)

[DAG Types 7](#_Toc158302768)

[Scheduler 7](#_Toc158302769)

[DAG Orchestrator 7](#_Toc158302770)

[Task Manager 7](#_Toc158302771)

[Task 7](#_Toc158302772)

[Cluster Manager 8](#_Toc158302773)

[Failure monitor 8](#_Toc158302774)

[Metadata 8](#_Toc158302775)

[Metadata schema 8](#_Toc158302776)

[Why we need Sharding? 8](#_Toc158302777)

[Multi-Tenancy 9](#_Toc158302778)

[Supported Storage types 9](#_Toc158302779)

[Input file formats 9](#_Toc158302780)

[Output file formats 10](#_Toc158302781)

[Output table formats 10](#_Toc158302782)

[Security 10](#_Toc158302783)

[Tracing 10](#_Toc158302784)

[Metric 10](#_Toc158302785)

[Alerts 10](#_Toc158302786)

[On-Call 11](#_Toc158302787)

[Deployment 11](#_Toc158302788)

[Deployment Regions 11](#_Toc158302789)

[Rolling Upgrades 11](#_Toc158302790)

[Feature Flags 11](#_Toc158302791)

[Testing 11](#_Toc158302792)

[Bechmarking 12](#_Toc158302793)

# Workflow Orchestrator

GOTO workflow orchestrator is a framework for executing ET (= extract-transform) workloads that are specified in the form of a DAG. A task within a DAG can typically read from and write to various object storages in various file formats and do transformation of data and store it back in various file and table formats. Also, a task has dependency with one more task.

As part of this framework, there is no support provided for bringing the operational, time-series, or transactional data into object storage or writing them directly into platforms like Snowflake. This framework assumes the data is already there in object storage that must be transformed and written back in a specific file and a table format.

To avoid the complexity around dealing with connectors eco-system, for which there are numerous solutions already exist, this framework caters more towards doing all the transformations and aggregations within an object storage and being good at it. You can assume this framework to be a direct replacement for AWS Glue ETL or spark batch jobs, with the ‘L’ missing.

Some of the key features of this framework include:

1. SLA-based
2. Zero-configuration
3. Serverless
4. Composable
5. Cost-effective
6. Near real-time
7. Reliable
8. Workflow management for Extract-transform data pipelines

The main aim is to save cost for customers, without them having to deal with complex infrastructure and thousands of configurations and reducing the overall data footprint and egress costs on expensive data platforms.

# Why the name GOTO?

Typedef as a company loves C language constructs and hence, we picked GOTO. This workflow orchestrator should also become the default **go-to** framework where customers love to schedule their ET jobs.

# High-level Architecture

## DAG Registration Architecture

Benchmarks Store

DAG Register

* Validate DAG
* Check SLA

DAG registration

(DAG, SLA, Schedule)

DAG Author

propose a different SLA, if its SLA related

If its DAG validation, return specific error code and message

DAG and SLA look fine

No

Metadata Store

persist the DAG in the metadata store, along with the serverless configuration to execute it.

Yes

## DAG RUN architecture

Scheduler

Check DAGS that have be to scheduled

Metadata Store

Update DAG Run table and trigger a DAG orchestrator

Failure Monitor

Pick a DAG and execute

DAG Orchestrator

Update Task table, with SLA and infra. Trigger task manager instance for each task

Pick a Task, spin up infra, and execute

Update Task table

Task Manager

Execute

Benchmarks Store

Query Engine

Update metrics

Worker N

Worker 1

Given the scope of this document, it’s not possible to show complete internal workings of all the components. Basic idea is that scheduler, DAG orchestrator, and task manager are all functions that run on serverless architecture, and each has a specific task to manage.

Scheduler is responsible for ensuring all the DAGs are scheduled correctly.

DAG orchestrator is responsible for picking a specific DAG that needs to be executed and managing its full life cycle.

Task manager is responsible for picking a task, spinning corresponding driver and worker nodes and invoking the query engine and monitoring the task till it finishes.

All the components rely on Metadata store for managing a DAG and task executions and do not talk to each other directly.

We would also need external monitoring for certain events.

For example: if an instance of DAG orchestrator dies without finishing its assigned DAG run, we should automatically spin up a new instance that should pick from where the previous instance left off and complete the DAG execution. Same goes with the task manager as well.

Also, it is very important that the metadata store is highly available and provides ACID guarantees and scale as per the workloads without increasing the read/write SLAs.

# Serverless

All the components in the above architecture are run in a serverless environment including the metadata and benchmarking stores.

# Directed Acyclic Graph (DAG)

Directed acyclic graph consists of one more task. It’s also possible to create dependencies between the tasks. If there is a cycle, the DAG would be rejected.

Few examples of a DAG given below.

|  |
| --- |
| task1 -> task2 -> task3 (Here, task3 must be completed before task2, and task2 must be completed before task1) |
| task1 (A DAG can consist of a single task) |
| task1 -> task2 (task2 must be completed before task1) |

During the execution of a DAG, order of execution of tasks must be based on the dependencies.

Specific example of a task is given below.

|  |
| --- |
| CREATE TABLE IF NOT EXISTS  db\_name.schema\_name.table\_name  PARTITIONED BY (col\_name\_2),  LOCATION '<s3://customer/time>-series/transformed/'  TBLPROPERTIES ('table\_type' ='ICEBERG')  AS  (SELECT col\_name\_1, col\_name\_2, col\_name\_2 FROM PARQUET\_TABLE\_ON\_S3 WHERE col\_name\_2 = "2024-02-08"  ORDER BY col\_name\_2  LIMIT 10,  CONNECTION\_TYPE='S3',  FORMAT="parquet",  CONNECTION\_OPTIONS = {  "path": "<s3://customer/time>-series"  }  ) |

Each task can be a SQL operating on the data sitting in the object storage.

# DAG Register

DAG Register is responsible for the initial registration of a DAG from the DAG author.

Its main responsibility include:

* Validating the DAG
* Validating the SLA

DAG Register talks to the benchmark store to decide a given DAG can be finished within the SLA.

If the given user SLA cannot be committed, then the DAG proposes a different SLA to the DAG author.

Once all the validations are complete, the DAG gets persisted in the metadata store, along with its SLA and schedule.

# SLA (Service-Level Agreement)

Service level agreements can be specified at the time of a DAG creation.

Each DAG run should honor these SLAs. If there are failures and SLA cannot be met, then proper alerts should be sent, so that the issue can be further investigated.

Below can be two different ways in which a DAG author can specify an SLA.

## Time to completion

Some DAGs might be time sensitive and must be completed within the stipulated time. In this case time to completion can be specified.

Whether a DAG run can meet time to completion SLA or not can be determined by retrieving relevant information from the benchmarking store. The basic idea is here is that you maintain some high-level metadata for datasets involved in the DAG and the operations that are being performed on those datasets within each task. Some of the high-level metadata that would be useful for a dataset are dataset size, cardinality, number of rows, partitions if any, file format, and table format.

Now let’s consider an example of a full table scan. A full table scan is an operation which scans all the rows of a dataset. We can run a query which does full table scan for various datasets with different sizes on different cloud providers and this can be repeated for different file and table formats and cardinalities, and we can determine the average time taken. There can be some more parameters like worker node size, number of threads, assuming certain operations can benefit from parallelism, hardware, etc. This benchmarking should be comprehensive and aim to include all the critical parameters that may affect the performance and SLA of a given task.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dataset size | operation | file format | table format | cardinality | object storage | worker node size (memory, # of cores) | # of threads | time in ms |

Once we know the average time taken for a given operation for a dataset with known metadata, we can then look our benchmarking store to determine how much time it would take to complete the operation. So, a task can be broken down into a bunch of operations and DAG itself can be broken down into bunch of phases, where each phase has one or more tasks. Average time taken for a given phase is determined by the task that takes the highest time within that phase.

For complex DAGs with multiple phases, to keep up with the SLA, we should dynamically determine the infra requirements, before a given phase is executed. For example: if some phases take longer than expected, then we should expedite remaining phases by providing a greater number of larger worker nodes.

## Cost to completion

Some DAGs might be cost sensitive and must be completed within the given cost. In this case cost to completion can be specified.

It should also be possible to pick time and cost and fine-tune both based on the DAG author’s needs.

Cost to completion is an entirely optional feature, which needs more research work.

# Registering a DAG

During the registration of a DAG, DAG should be checked for cycles. Also, SLA should be verified to ensure that the DAG can finish based on the specified SLA. If not, the registration should be denied with appropriate message.

# DAG Types

DAG is of two types.

* On-demand, run-once.
* Scheduled, runs at regular frequency.

# Scheduler

Scheduler is responsible for keeping track of all DAG runs and executing the DAG runs based on their schedule.

# DAG Orchestrator

DAG orchestrator is responsible for taking a DAG and its SLA and ensuring that DAG run completes successfully within a given SLA. Orchestrator is also responsible for fault tolerance and resiliency for a DAG run.

Based on the dependency graph, orchestrator can execute multiple tasks in parallel.

# Task Manager

Task manager is responsible for executing a task and ensuring the task completes within a given SLA. Task manager is also responsible for fault tolerance and resiliency for the task that it is managing.

# Task

Task consists of a single operation.

For ex: executing a CTAS query on S3, which reads a parquet file from s3 and writes it into a table with Iceberg table format.

Each task would leverage the serverless query engine framework to execute the queries. Design and implementation of the serverless query engine is beyond the scope of this document.

# Cluster Manager

This component is only relevant if we are managing the query-engine driver and worker node infrastructure ourselves.

If they are implemented as Lambda functions and use Cloud provider’s serverless infra, then we do not need this component.

# Failure monitor

Failure monitor is responsible for monitoring for specific failures or the heartbeats of a DAG and task execution.

For example: if a DAG orchestrator fails to update the update the heartbeat of a DAG execution for a certain interval, failure monitor should trigger a new instance of DAG orchestrator, which resumes the DAG execution. Similar thing can be done for a task manager as well.

# Metadata

Metadata includes information about the DAG author, DAG, DAG runs, tasks, SLA, security groups, and datasets.

Metadata is stored in a distributed cloud native transactional store, which is highly available and can elastically scale as per the workload, without affecting the read/write SLAs.

For metadata store, we can consider any of the modern cloud native NewSQL databases that can be run on a serverless infrastructure. For ex: CockroachDB

## Metadata schema

Please refer to <https://airflow.apache.org/docs/apache-airflow/stable/database-erd-ref.html> .

We can take inspiration from Airflow’s metadata schema. However, since all the components are serverless and we are looking at providing multi-tenancy, wherever its appropriate, we should also have a column for partitioning the customer workloads. For example, we can introduce a column called SHARD\_ID in the DAG tables, which can be used to distribute customer workloads. We can use the CUSTOMER\_ID to determine the corresponding SHARD\_ID.

## Why we need Sharding?

Let’s say our serverless scheduler runs at specific frequency, gets all the relevant DAGs by talking to the metadata store, then it is also responsible for scheduling the DAGs that are due, based on their schedule.

However, a given scheduler can only handle so many entries in the metadata store. To scale horizontally, each scheduler can now be responsible for all the DAGs associated with a specific SHARD\_ID and it will only retrieve and schedule DAGs for the SHARD\_ID that it is responsible for. This allows scheduler itself to scale horizontally.

Also, in terms of security, isolating customer workloads using sandboxing is already an established practice. So, all the relevant tables should also use SANDBOX\_ID, along with CUSTOMER\_ID.

So, to conclude, we would need a metadata schema very similar to that of airflow, but with additional columns for enhanced security, isolation, and to make the workloads more appropriate for serverless-first architecture.

All the components completely rely on the metadata to finish a DAG run and doesn’t necessarily talk to each other directly.

# Multi-Tenancy

Workflow orchestrator will be multi-tenant. We can maintain customer/sandbox/user hierarchy for the DAG authors. We can also introduce security user groups to control the visibility of datasets, DAGs, queries, etc. For more predictability of workload performance and horizontal scalability we will customer shards or partitions. Please read Metadata section to learn more.

Access control can be implemented little later, once the basic functionality is working end-end. As part of access control, we can consider role-based, row-based, and column-based granular controls.

# Supported Storage types

|  |
| --- |
| AWS S3 |
| Azure Blob Storage |
| ADLS - Gen 1 |
| ADLS - Gen 2 |
| GCP Cloud Storage |

We can initially just start with AWS S3.

# Input file formats

|  |
| --- |
| JSON |
| Parquet |

We can provide support for more file formats in future.

# Output file formats

|  |
| --- |
| Parquet |

We can provide support for more file formats in future.

# Output table formats

The project <https://onetable.dev/> will be handy to create the output tables in all the below table formats.

|  |
| --- |
| Iceberg |
| Delta |
| Hudi |

# Security

This framework will incorporate the concept of sandboxing to provide isolation and include the concept of sharding for providing reliable workflow executions. If there are costumers that need complete isolation, we can dedicate a SHARD\_ID for a single customer.

During the task execution, all the secrets should be accessed through an authorization token whose scope is restricted to CUSTOMER\_ID/SANDBOX\_ID/USER\_ID.

# Tracing

Tracing is an important aspect of this framework. Since all the tasks are associated with a given DAG run ID, this can be used to trace the complete flow of a DAG run, along with the time spent at various components, cost, and failures.

# Metric

Extensive metrics should be collected in all the phases of a DAG run execution. This metric, along with the metric from the benchmarking should help us better predict the infra that needs to be spun up to complete a given DAG run, within the SLA.

# Alerts

Since the entire system is built around the concept of SLA and high resiliency, it is important to try our best to keep up the SLA. There can be times where a given DAG run cannot be finished within the SLA. In such cases, the system should generate an alert, which goes to the person on-call. On-call person should have enough knowledge to root cause the issue and propose a solution to avoid the same in future. Since this framework can get complex over time, it is very important to document all the decision making and reasoning behind and how to proceed when things go wrong. Since the entire eco-system would live on multiple cloud providers and would eventually interact with other vendors as well, we should set similar expectation from those services as well, in terms of support and response to failures.

# On-Call

This can come later, but ideally an on-call person has 8-hour shift, and we would need 3-people sharing on-call duties in a day. These folks should have sufficient knowledge about the system and enough runbooks and documentation for them to deal with the production issues.

We can also consider 12-hour shifts initially, if the 8-hour shifts are not viable financially for a smaller company.

# Deployment

Considering all the components are implemented as on-demand lambda-like functions, we should have the ability to ship each component separately or together with daily trains to dev, stage, pre-prod, and prod. These trains should auto-promote if the e2es have executed successfully for a given environment.

# Deployment Regions

For better performance and reduce network latencies, each regional deployment should be self-sufficient. This means during the registration of a DAG or a DAG run, any of the component need not be talking to any component that is deployed in a different region. Only exception would be if there is a regional failure. In that case, till the region recovers, components can either execute or talk to other components in a different region. But we should avoid cross-regional communication as much as possible and make each regional deployment self-sufficient.

This also helps with better performance as we can bring compute closer to where the customer data exists.

There is some scope of optimization in this space, as customer datasets themselves can come from different regions, in which case we can pick the deployment region for compute, where the largest datasets are residing.

# Rolling Upgrades

With all the components realized as functions, this should be easy to achieve.

# Feature Flags

All the features should be behind a feature by default. This improves the confidence for daily trains and results in less firefighting in production, if things go wrong, after a release. When things go wrong, instead of rolling back the new version, we can just turn off the features that are not working.

# Testing

Apart from the unit tests of each component, developers should have the ability to execute end-end tests locally within a laptop. This helps the developers to be more productive and debug the issues quicker.

Various types of tests

* Unit tests: Extensive unit tests coverage. It would be good to aim at least 70% unit test coverage and fail the build if it falls below that threshold. Unit test coverage is not a very good metric to measure the quality of code. So, more emphasis should be given to covering all the positive and negative tests that cover a given functionality.
* Integration tests: Tests which test the touch points between two components.
* E2Es: End to end tests which tests the complete runs of a workflow, correctness of query execution, and time to completion, and SLA.

# Bechmarking

Benchmarking is a very important aspect of this framework, since it relies on benchmarking metric and metric from the actual DAG runs to determine a given SLA can be accepted or not, during the DAG registration. In this regard, its important keep a pre-prod environment which runs on the same hardware, same version of the components that are running in production. Sample workloads running in the pre-prod environment should cover all the query patterns that one would expect during an ET job from the customer.

In this regard, a DAG can be realized as a set of phases, where each phase contains one or more tasks. Time to completion of a phase would be same as the time taken by the largest running task in that phase.

Also, each task can be realized as a set of operations on a dataset. For example: scan, filter, sort, aggregate. Based on the dataset size and the operations involved in a task and the corresponding infra that is allocated, we should be able to calculate how much time might be taken by a given task.