ISSN: 2752-3829 Vol. 5 No.1, (June, 2025)

Stochastic Modelling and Computational Sciences

STUDY OF WORKFLOW ORCHESTRATION ENGINES: OPEN-SOURCE & CLOUD-NATIVE SOLUTIONS

Nikhil Sagar Miriyala
[1]Senior Software Engineer, Oracle America Inc., USA
*Corresponding Author: nmiriya7@gmail.com

ABSTRACT:
Workflow orchestration engines have emerged to become one of the efficient methodologies for automating and managing complex processes across distributed systems. These tools enable seamless coordination between microservices, ETL pipelines, business workflows, and cloud-based applications by efficiently handling task dependencies, failures and retry mechanisms, and scalability concerns. This paper presents an in-depth exploration of four widely used workflow orchestration engines: Apache Airflow, Netflix Conductor, Temporal, and AWS Step Functions. We discuss their execution models, core components, key features, scalability, and use cases. Additionally, a comparative analysis highlights their strengths and limitations, aiding organizations in selecting the most suitable workflow orchestration framework based on their requirements.

Keywords*: Workflow Automation, Orchestration Engines, Netflix Conductor, Temporal, Apache Airflow, AWS Step Functions*

1. INTRODUCTION

With the increasing complexity of modern applications, workflow orchestration has become as an essential framework to develop and handle automation flows with high reliability and scalability. Organizations use workflow orchestrators to define, execute, and monitor tasks in a structured and efficient fashion. These tools abstract the complexities such as state management, error handling, and task dependencies, enabling engineers to focus on business logic rather than infrastructure concerns.

Workflow orchestration engines are particularly valuable in the following scenarios:

1) **Data Pipelines:** ETL (Extract, Transform and Load) operations, especially within organizations handling large volumes of data, require structured workflows to run the pipelines in an efficient manner. In such scenarios, organizations can utilize orchestration engines that designed to move data seamlessly from different sources, while undergoing necessary transformations, with high resiliency and availability [1]. These data pipelines are majorly used in sectors such as finance, healthcare, and e-commerce, where critical decision-making tasks depend on properly organized data and its insights.

2) **Microservices Coordination:** Most of the modern applications are adopting a microservices architecture, where independent services work together to deliver a seamless end user experience. While this architecture minimizes the issues involved with monolithic applications, managing the communication between multiple applications comes with its own challenges. Orchestration engines can be used in such scenarios, in order to manage the complex dependencies among these microservices, ensuring effective communication, graceful failure handling, and efficient scaling techniques [2]. For example, an e-commerce application may depend on a set of microservices for order processing, payment handling, inventory management, and shipping, with all of them required to be communicating efficiently at all times.

3) **Business Process Automation:** Large-Scale Organizations heavily depend on certain predefined business processes that can be automated with minimal human involvement. Orchestration engines can be utilized in such scenarios [3], and help with streamlining approval workflows, contract management, financial auditing, and customer support processes. For example, an insurance firm can utilize an orchestration engine that can automate the processing of claims raised, by managing document verification, detecting fraud, and coordinating final approval, ensuring that workflows are conducted systematically and in compliance, with support for human interruption and approval when needed.

4) **Event-Driven Applications:** In a lot of real-time workflows, applications have to respond dynamically to events triggered by users, sensors, or third-party systems. Orchestration engines can be used within an event-driven systems [4], by designing workflows that can react to incoming data and trigger corresponding actions. For example, in IoT applications, an orchestration engine can process sensor data in real-time, triggering alerts, updating databases, or activating other systems when predefined conditions are met.
5) **Cloud-Native Applications:** Enterprises leveraging cloud infrastructure can automate the process of provisioning the resources, and monitor and scale them dynamically as needed. Workflow orchestration engines can efficiently take care of infrastructure-as-code (IaC) deployments, disaster recovery protocols, and cost-efficient resource management procedures [5]. For example, in a DevOps environment, orchestration engines can be used to automate CI/CD pipelines, rolling updates, and auto-scaling processes across cloud providers such as AWS, Azure, etc.

This paper explores the below four well-known workflow orchestration engines:

1) **Apache Airflow:** A DAG-based orchestration engine optimized for ETL and batch operations.
2) **Netflix Conductor:** A state-machine based orchestrator designed for microservices coordination.
3) **Temporal:** A durable execution system leveraging event sourcing for fault-tolerant workflows.
4) **AWS Step Functions:** An AWS managed serverless orchestration service built on state machines.

The following sections provide an in-depth review of these workflow engines, where we study their architecture, key features, use cases, and limitations.

2. CORE COMPONENTS AND HIGH-LEVEL ARCHITECTURE:

2.1 Apache Airflow:

Apache Airflow follows a scheduler-driven execution model where workflows, treated as a DAG (Directed Acyclic Graph), are parsed, scheduled, and executed based on task dependencies and timing configurations. Below are the core components of Apache Airflow:

1) DAGs (Directed Acyclic Graphs)

a. A Directed Acyclic Graph (DAG) is a graph-based structure where tasks are executed in such a way that the flow is always directional, and does not create any cycles, (i.e., no task can be dependent on itself). DAGs guarantee that workflows move in a topologically sorted sequence, preventing the possibility of infinite loops and deadlocks.

b. DAGs define the workflow structure, ensuring that tasks execute sequentially or in parallel based on predefined dependencies. In addition to the dependencies, each task within the workflow can be configured using multiple options, such as the number of retries, timeout parameters, error scenarios, etc.

c. All the tasks defined in the workflow, are instances or extensions of inbuilt operators (such as PythonOperator, BashOperator, etc.)

d. Figure 1 shows an example DAG workflow.

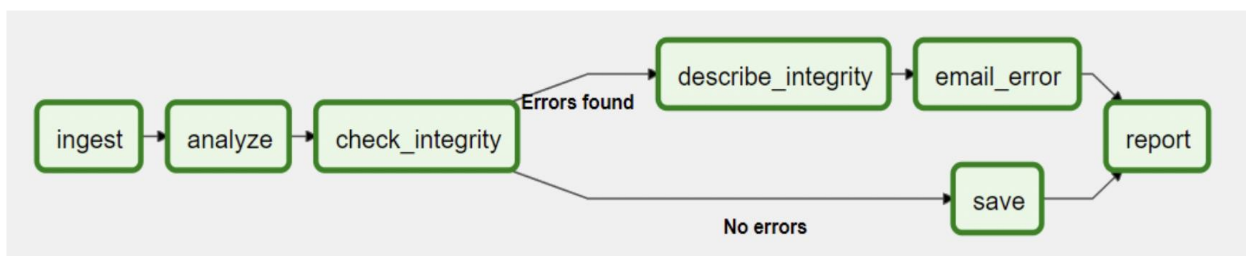

Figure 1: Sample DAG Workflow [6]

2) Scheduler

a. The scheduler is responsible for invoking the workflows and the corresponding tasks based on defined schedules and dependencies.

b. It evaluates the DAG on a periodic fashion, and updates the execution state for each task based on its dependencies. The state of each task can be one among waiting, ready, running, success, failure, shutdown and up for retry.

3) Executor:

a. The tasks scheduled for execution are invoked using the executor. Airflow supports multiple executor backends:

- LocalExecutor: Executes tasks on the same machine as the scheduler.
- CeleryExecutor: Distributes tasks to worker nodes using a queue-based system.
- KubernetesExecutor: Runs tasks as Kubernetes pods, improving scalability.

4) Metadata Database:

a. Airflow uses a relational database (e.g., PostgreSQL) to store all the DAG executions, status of the tasks, and their corresponding logs.

b. The database acts as a single source of truth, tracking DAG execution history and task states. It is used by the Web Server to provide statistics for monitoring purposes and more importantly, by the scheduler and executor systems, when tasks are being executed in a distributed fashion.

5) Web Server:

a. The web UI provides real-time observability features, enabling users to visualize workflows, monitor execution states, and view logs.

6) Task Queue (Only for Celery Executor):

a. When using a CeleryExecutor, all the tasks are placed into a queue, which enables distributed execution using multiple worker nodes.

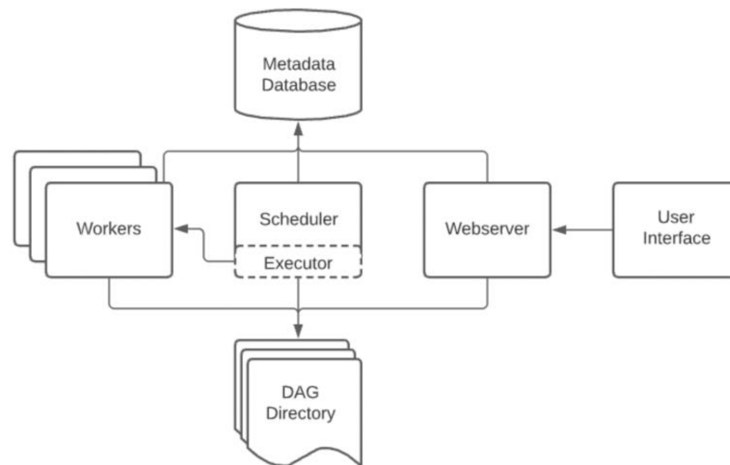

Figure 2: Apache Airflow High Level Architecture [7]

Figure 2 shows the high-level architecture of Apache Airflow. The metadata database acts as the central point for communication across all the components. The Scheduler reads DAG definitions from the DAG Directory and schedules tasks based on dependencies. Tasks are executed using an Executor, which distributes them to Worker nodes for processing. The database stores execution states, task logs, and scheduling information, ensuring persistence and recovery. Users interact with Airflow via a Webserver and UI, which provides monitoring, debugging, and workflow management capabilities. This architecture enables flexible, scalable, and highly customizable batch processing workflows.

3.2 Netflix Conductor

Netflix Conductor follows a state transition model that provides dynamic execution of workflows, including a wide variety of observability features, enabling efficient monitoring and debugging. The primary components of Netflix Conductor include:

1) Workflow Definitions (JSON DSL):

a. Workflows are defined using a JSON-based DSL, that describes the order of execution among the tasks, along with their conditions, and dependencies.

b. This blueprint-based approach decouples business logic from execution logic, allowing changes to workflows without modifying microservices.

c. Figure 3 shows an example Workflow Definition JSON file.

```
{
  "name": "workflow_name",
  "description": "Description of workflow",
  "version": 1,
  "tasks": [
    {
      "name": "name_of_task",
      "taskReferenceName": "ref_name_unique_within_blueprint",
      "inputParameters": {
        "movieId": "${workflow.input.movieId}",
        "url": "${workflow.input.fileLocation}"
      },
      "type": "SIMPLE",
      ... (any other task specific parameters)
    },
    {}
    ...
  ],
  "outputParameters": {
    "encoded_url": "${encode.output.location}"
  }
}
```

Figure 3: Example Workflow in Netflix Conductor (JSON) [8]

2) API Layer:

a. Conductor provides a set of APIs that facilitate communication between the orchestration engine and task workers. These APIs allow workers to poll for tasks, update task statuses, and retrieve workflow definitions, enabling seamless integration and coordination.

3) Service Layer:

a. The service layer consists of all the core components described below, which are combinedly used for processing workflows and tasks:

- Workflow Service: Manages workflow lifecycle, tracking execution progress and state transitions.
- Task Service: Handles individual task execution, tracking statuses, retries, and failure handling.
- Decider Service: Evaluates workflow states and determines which tasks to execute next. It is responsible for scheduling, dependencies, and conditional execution logic.
- Queue Service – Manages distributed task queues, ensuring efficient polling and worker assignments.

4) Task Queues & Workers:

a. Conductor utilizes a distributed queue to manage and schedule tasks. This system ensures tasks are assigned to workers efficiently and can handle delays or retries as necessary. At Netflix, this is implemented using dyno-queues on top of Dynomite, facilitating distributed delayed queues.

b. Tasks within workflows are executed by worker applications. These workers can either expose REST endpoints for the orchestration engine to invoke or implement a polling mechanism to fetch and execute pending tasks. Workers are designed to be idempotent and stateless, allowing for scalability and resilience. The polling model helps manage backpressure and supports auto-scaling based on queue depth.

5) Metadata and Persistence Layer:

a. Workflow definitions, task metadata, and execution logs are stored in a persistence layer. Conductor supports various storage backends, including Cassandra and Dynomite, to maintain the state and history of workflows.

6) Monitoring & Observability

a. A real-time monitoring system can be setup for each workflow, with support for integration using Prometheus, Datadog, and ELK Stack.

b. A Web UI is available for use as well, which provides an intuitive dashboard for debugging and tracking workflow execution.

Figure 4 shows a high-level architecture of Netflix Conductor. As mentioned, the system follows a poll-based task execution model where workers communicate asynchronously with the orchestration engine. The Orchestrator schedules tasks and persists workflow execution details in a Database and Index. Tasks are pushed into Task Queues, where worker nodes continuously poll for available tasks (blue lines in the diagram). Once a worker picks up a task, it executes the logic and updates the task status via the Management/Execution Service (red lines in the diagram). This architecture enables fault-tolerant, scalable, and distributed workflow execution, making Conductor suitable for microservices orchestration and business process automation.

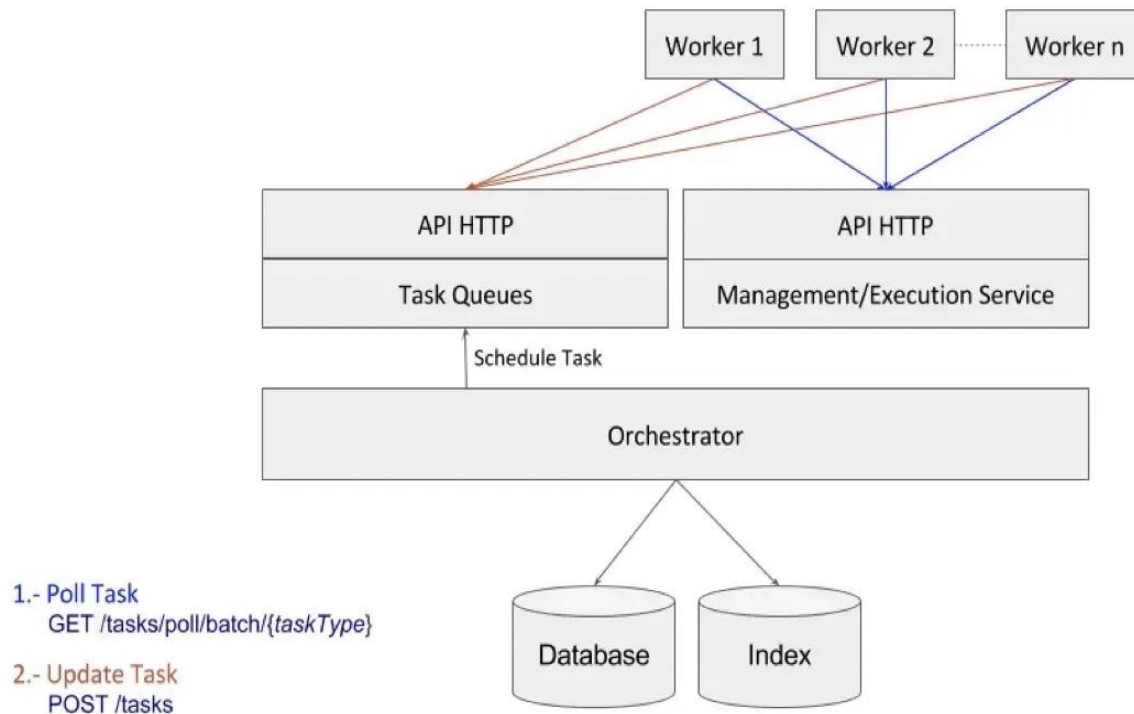

Figure 4: Netflix Conductor – High Level Architecture [8]

2.3 Temporal:

Temporal works based on event sourcing model, which is designed to manage robust executions, meaning the workflows will keep running even after the occurrence of a failure. The core components of Temporal are as follows:

1) Temporal Server

a. Temporal Server acts as the brain behind all the operations. It takes care of scheduling workflows, making sure they are persistence, executing them and maintain the state of the tasks.

b. This component provides multiple services. The Frontend Service offers a gRPC-based API for interaction while the History Service keeps track of the workflow event history.

2) Workflows

a. In Temporal, workflows are authored as executable code directly, unlike other orchestrators, which use configurable files using DSL or YAML. This provides the workflows to be fully programmable and flexible in nature.

b. The workflow in Temporal runs deterministically, so that they follow the same execution path for the same inputs. Because of this consistency it allows them to be replayed from the history logs if necessary.

3) Activities & Activity Workers

a. Activities are external operations that are executed within the workflows like database queries, API calls or computational tasks.

b. Activity workers perform the activities asynchronously and handles other features like failures, retries and state persistence on their own.

4) Task Queues & Matching Service

a. The tasks that are to be performed are placed in the queue, managed by the worker processes, that communicate with the Matching Service to fetch and execute the workflow steps.

b. Task queues separate the workflow logic from execution which enables horizontal scaling of the worker nodes.

5) History Service & Event Logs

a. The History Service is used to records the logs of all the events during a workflow's execution. This captures every state transition that occurs like the start of workflow, scheduling of task, completion of task and also termination of the workflow.

b. This is a crucial component as it allows the workflows to resume their execution from the last-known stable state in case of failures.

c. The event logs function as an audit trail, which helps developers to debug the workflow issues & failures, and analyze execution patterns.

d. Temporal provides multiple views of workflow history:

o Timeline View: Shows execution primitives and their duration, making it easier to analyze workflow execution times.

o Compact View: It provides the expandable sections of the groups related workflow events, like scheduled, started & completed events for an activity, for easier navigation.

o Full History View: Displays the complete event history of a workflow, which can be sorted by either ascending or descending order. It captures all workflow tasks, activities, and state changes.

Figure 5 shows the high-level architecture of a Temporal Cluster. Once the workflow execution starts, all the execution metadata is persisted using the Temporal History Service, ensuring that execution can be replayed if needed. Workers poll Temporal's Task Queues to fetch tasks and execute them, while the Temporal Frontend Service handles API requests and workflow coordination. The Matching Service routes tasks to appropriate workers, and the History Service maintains execution logs, allowing workflows to resume even after failures. This model enables deterministic execution, long-running workflows, and high scalability across distributed environments.

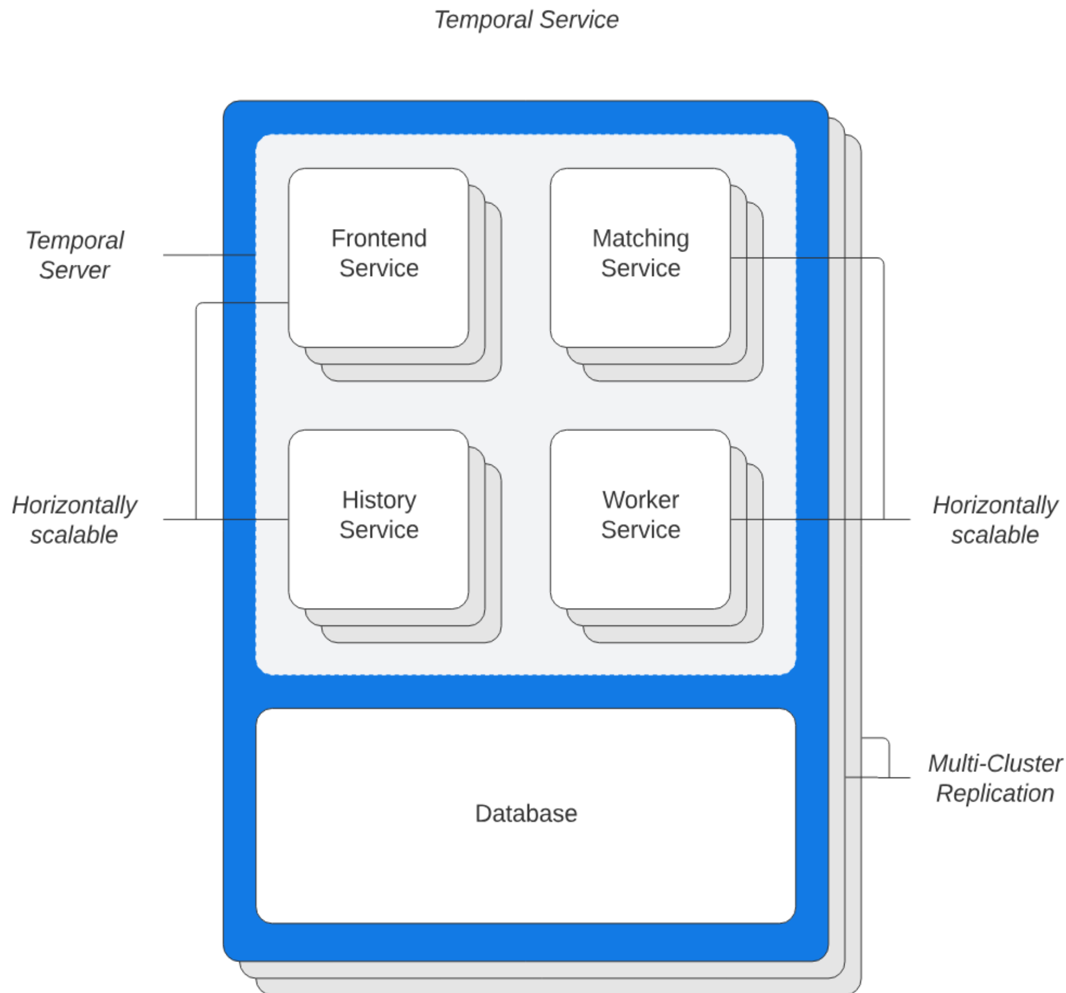

Figure 5: Temporal – High Level Architecture [9]

2.4 AWS Step Functions

AWS Step Functions provides a serverless orchestration service by following a state machine model which seamlessly integrates with AWS services. The core components of AWS Step Functions include:

1) State Machine Execution:

a. In the AWS Step Functions, the state machines are called as workflows. Each step in the workflow represented as a state.

b. Each state transition occurs based on the predefined set of conditions which enables different type of work like sequential, parallel& execution flows.

c. Using the Amazon States Language (ASL), which is JSON based used to define the state of the machine i.e. execution logic & transitions.

d. Figure 6 shows the sample workflow.

```json
{
  "StartAt": "Task1",
  "States": {
    "Task1": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:MyFunction",
      "Next": "Task2"
    },
    "Task2": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:MyFunction",
      "End": true
    }
  }
}
```

Figure 6: AWS Step Functions Sample Workflow

2) Task States & AWS Integrations:

a. Task States enables serverless and container-based execution which invoke the various AWS services like AWS Lambda, Elastic Cloud Compute (ECS), Batch, DynamoDB and SageMaker.

b. Supports the external service invocation using API Gateway which can directly integrate with over 200 AWS Services.

c. It integrates the services and simplify the automation thus reducing the need for standard custom code in AWS environments.

3) Choice & Parallel States:

a. Choice States allow conditional branching, allowing for decision-making within workflows.

b. Parallel States enable concurrent execution of multiple workflows, enhancing efficiency for large-scale batch operations.

c. Wait States introduce delays for scheduled execution or external event handling.

4) Execution Logging & Monitoring:

a. AWS Step Functions capture every step, state transitions and failure points by maintaining an execution log.

b. Real-time monitoring and alerting can be performed for workflows based on its performance and failures by integrating with the AWS CloudWatch service.

c. The AWS Management console supports step-by-step debugging, providing details on inputs, outputs, and execution timelines.

5) Workflow History & Event Logs:

a. AWS Step Functions log workflow history, detailing all the aspects of the execution, from inputs and outputs to any error state encountered along the flow.

b. Execution History acts as an audit trail, monitoring workflow progress, failures & retries.

c. Event logs allow for the workflows to be retired from the last known state without having to restart the entire workflow.

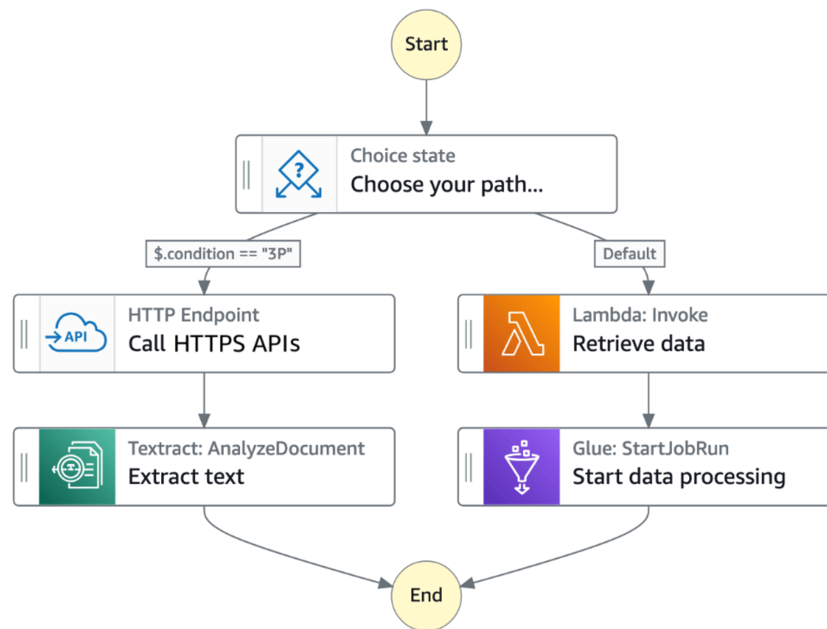

Figure 7: An example of AWS Step Function workflow

Figure 7 shows a sample high level flow diagram for AWS Step functions. In this case, the workflows begin with a choice state, directing execution based on conditions. Tasks are executed using integrations with AWS services such as Lambda (compute logic), Glue (data processing), Textract (document analysis), and API Gateway (external API calls). In addition, Step Functions can handle parallelism, retries, and error handling automatically. Further, the execution history is persisted and maintained, ensuring reliable and scalable workflow management. This architecture allows for event-driven automation, complex business logic execution, and seamless AWS-native service coordination.

3. Use Cases

Each of these orchestration engines can serve distinct use cases, thus catering to various operational needs. Below are the specific use cases best suited for each of the workflow orchestration engine.

3.1 Apache Airflow [10]

1) ETL & Analytics

- Apache Airflow is widely used to orchestrate ETL (Extract, Transform, Load) workflows, ensuring reliable data ingestion, transformation, and storage in data warehouses.
- It enables data pipeline automation, handling dependencies, scheduling jobs, and monitoring task execution across different data sources and analytical platforms.

2) Business Operations Automation

- Airflow helps automate business processes such as report generation, financial transactions, customer onboarding, and supply chain management.
- It streamlines operational workflows by enabling event-driven task execution, reducing manual effort, and ensuring process efficiency.

3) Infrastructure Management & DevOps

- Airflow is used for cloud infrastructure provisioning, continuous deployment, and resource scaling, integrating with tools like Terraform and Kubernetes.
- It allows organizations to automate CI/CD workflows, infrastructure updates, and system monitoring, ensuring operational stability and efficiency.

4) MLOps & Machine Learning Pipelines

- Airflow is a key component in MLOps workflows, orchestrating data preprocessing, model training, validation, and deployment.
- It supports automated model retraining, hyperparameter tuning, and version tracking, ensuring ML models are continuously updated with fresh data for improved performance.

3.2 Netflix Conductor [11]

1) Microservices Workflow Orchestration

- Netflix Conductor is designed to orchestrate microservices in distributed environments, ensuring that dependent services communicate efficiently.
- It enables asynchronous execution, fault tolerance, and retry mechanisms, helping teams coordinate multi-step workflows across independently deployed services.

2) Realtime API Orchestration

- Conductor allows dynamic API workflows where multiple APIs are invoked in sequence or parallel to perform complex business operations.
- It supports conditional branching, error handling, and rate-limiting to ensure seamless real-time API processing while reducing latencies.

3) Event-Driven Architecture

- Conductor integrates well with event-driven architectures, allowing workflows to be triggered by external events such as Kafka messages, database changes, or webhook calls.
- It supports asynchronous task execution, making it ideal for reactive workflows where components communicate through events rather than synchronous calls.

4) Agentic Workflows

- Conductor enables agentic workflows where AI-driven decision-making can be integrated into automated processes.
- It facilitates adaptive and intelligent orchestration by incorporating AI/ML models, feedback loops, and autonomous decision-making workflows.

3.3 Temporal [12]

1) Long-Running & Fault-Tolerant Workflows:

- Manages long-running tasks without separate state management, ensuring high reliability for workflows that need to persist state over time.
- It allows workflows to continue seamlessly when failures occur. By persisting state over time, it allows workflows to resume from the last known stable state.

2) Distributed Transactions & Human-in-the-Loop Processes:

- Temporal enables consistency across multiple services, ensuring reliable and accurate transactions.
- Temporal enables human approval, making the process more flexible and adaptable. It adds an extra layer of reliability since critical decisions can be made with human insights.

3) Reliable Event-Driven Microservices Execution:

- Temporal provides a central place to manage all the tasks, which helps in managing complex workflows more efficiently.
- Temporal maintains the state of the workflows, providing high resilience and automatic retries in case of failures for cloud-based event-driven workflows.

3.4 AWS Step Functions [13]

1) Serverless Application Orchestration:

- Coordinates the AWS services like AWS Lambda, DynamoDB queries & API Gateways, in the workflow. It can execute the tasks in parallel and handle errors using retry and catch clauses.
- AWS Event Bridge service can be used to trigger Step Function workflow process for specific events.
- Express workflows can be used to manage large volumes of events and workflow executions, offering scalability and high throughput.

2) Cloud-Native Infrastructure Automation:

- Step Functions can be utilized for synchronization by running parallel independent workflows, maintaining consistency, and optimizing the performance.
- Step Functions can be used to configure and manage the cloud provisioning for a smoother, more reliable setup.

3) CI/CD Pipelines & Secure AWS Resource Management:

- Step Functions can be used to automate CI/CD workflows thus creating a fully managed continuous delivery of system that can build, test & deploy applications in a dependable & repeatable manner.
- Step Functions are utilized to automatically create security incident responses with the capability to trigger manual approval steps when needed, providing an additional layer of security.

4) ETL & Data Processing:

- Step Functions enable the automation of the ETL pipeline procedure which includes data extraction, transformation, and loading. The jobs can run in parallel which improves the performance, reduce processing time and ensure fault-tolerant data pipelines.
- Step Functions enables scalable, fault-tolerant large scall data processing via parallel workflows and integrating with other AWS Service.

4. Key Features and Limitations

4.1 Key Features

1) Apache Airflow:

a. DAG-based workflow orchestration with support for sequential, parallel, and conditional task execution.

b. Flexible scheduling options through time-based and event-driven triggers.

c. Supports various execution backends (CeleryExecutor, KubernetesExecutor, LocalExecutor) allowing efficient scalability of the system.

d. Provides extensive integrations with databases, APIs, cloud services (AWS, GCP, Azure), and other external platforms.

e. Includes observability features bundled and made available through an interactive web interface that tracks tasks in real-time.

f. Built-in support for error management & retry mechanisms which increases the resiliency of the system.

2) Netflix Conductor:

a. Designed with a microservices-first approach, it supports asynchronous execution and workflow management through APIs.

b. Allows for dynamic execution paths in workflows, enabling real-time adjustments based on external factors.

c. Is highly distributed and scalable, capable of handling high-throughput and concurrent executions.

d. Incorporates fault tolerance with automatic retries, compensation strategies, and prioritization of tasks.

e. Supports human-in-the-loop workflows, facilitating manual approvals and human decision-making points.

3) Temporal:

a. Temporal uses event sourcing techniques to ensure durable workflow execution, which helps in maintaining the integrity of the workflow even in the case of intermediate failures.

b. It supports deterministic replay and automatic state preservation, increasing the resiliency of long-running workflows.

c. Temporal framework can scalable easily, so that the cluster can simultaneously handle thousands of workflows using efficient task queues and concurrent workers.

d. Temporal SDKs are available in a variety of programming languages (Java, Go, Python, TypeScript, .NET), which improves the flexibility in terms of implementing a workflow.

4) AWS Step Functions:

a. AWS Step functions are serverless, with infrastructure fully managed by AWS. It comes with in-built deep integration with multiple AWS services, such as Lambda, DynamoDB, ECS, API Gateway, SageMaker, S3, etc.

b. All the AWS security and compliance features such as IAM permissions, audit logging, and encryption are available for AWS Step functions, similar to other services that AWS provides.

c. Step functions are built on top of state machine-based execution principles, which means, they support parallel runs, choice states, and wait states for complex workflows.

d. Provides high resiliency using integrated error management, retries, and automatic recovery techniques, thus reducing the operational burden.

e. Step functions can scale easily to support millions of workflow executions, making it an ideal fit for cloud-native applications.

4.2 Limitations

1) **Apache Airflow:**

a. Complex Configuration and Maintenance: Unlike fully managed solutions like AWS Step Functions, the operational complexity of deploying Airflow based workflows to production like environments can be comparatively high, since it includes managing several components, such as the scheduler, executors, and the metadata database. [14]

b. No Native Support for Pipeline Versioning: Airflow does not support versioning of workflows, which can make rollbacks difficult when needed. [15] [16]

c. Limited Data Quality Monitoring: Airflow lacks built-in capabilities for monitoring data quality, thus requiring engineers to utilize external tools in order to validate data accuracy and consistency. [16]

d. Scaling Limitations: Although the workers can scale horizontally, Airflow workflows can potentially experience performance challenges under heavy workloads, requiring careful resource management.

2) Netflix Conductor:

a. Complex Deployment and Management: Unlike fully managed solutions like AWS Step Functions, implementing Conductor based workflows requires managing several components, such as Elasticsearch, Redis, and a relational database, which increases the operational complexity of the system.

b. Possible Database Limitations: When dealing with workflows that can be resource heavy, components like the relational database or Elasticsearch can introduce some bottlenecks, effecting the overall performance of the system. [17]

c. Performance Issues in Highly Nested Workflows: Since Conductor uses a polling-based task execution model, the performance of the system could be degraded, when dealing with workflows that have significant nesting or excessive branching. [18]

3) **Temporal**

a. Complex Deployment and Maintenance: Similar to Airflow and Conductor, Temporal also requires handling multiple services, which can make self-hosted deployments difficult.

b. Strict Deterministic Execution Model: Given that the execution model of temporal is strictly deterministic, developers must make sure that the workflows are authored accordingly, which requires certain code changes, especially when transitioning from non-deterministic orchestrators.

c. High Storage Demands for maintaining Execution History: Since Temporal retains complete execution history of workflows, the cluster requires an efficient large storage system for workloads with significant throughput, which can add up to the overall cost. [18]

d. Limited built-in UI and monitoring tools: While Temporal comes with a user interface for visualizing workflows, detailed tracing and debugging requires the use of external tools.

4) **AWS Step Functions:**

a. Payload Size Restrictions: The request payloads sent to Step Functions cannot exceed a maximum size of 256KB, thus requiring developers to carefully construct the workflow. [19]

b. Execution Duration and Event Limitations: Step Functions based workflows can only last up to one year and support a maximum of 25,000 events per workflow. Workflows that exceed these thresholds will require segmentation in order to prevent hitting hard limits configured by AWS.

c. Execution history limitations: The execution history is captured and stored for a maximum of 90 days, which can go against applications needing longer audit trails or historical evaluations. [19]

d. Vendor Lock-In & Cost Considerations: Since Step Functions are proprietary to AWS, it could be difficult to transition to other cloud providers without extensive re-engineering. Moreover, costs can increase significantly at scale, as there are charges for each state transition, leading to high expenses for workflows with large number of events.

5. CONCLUSION

Workflow orchestration engines are essential for managing complex, distributed workflows in data engineering, microservices architecture, machine learning, and cloud automation. The four orchestration tools examined—

Apache Airflow, Netflix Conductor, Temporal, and AWS Step Functions—each have unique strengths and weaknesses, making them more suitable for specific scenarios.

Apache Airflow is known for its DAG-centric approach to workflow orchestration, efficient scheduling features, and extensive integrations with databases, public cloud providers, and monitoring tools. Nonetheless, it is not well-suited for real-time event-driven workflows, has limited fault tolerance outside of task retries, and can experience scalability issues when faced with high loads.

Netflix Conductor, originally created for orchestrating microservices, works well within asynchronous processing flows, supporting high concurrency, and allowing real-time modifications of workflows. Its adaptability makes it an excellent choice for API-driven workflows and event-driven setups, but it demands more effort to configure and manage, and has a higher learning curve compared to declarative orchestrators.

Temporal offers stateful, fault-tolerant execution of workflows by utilizing event sourcing and deterministic execution to guarantee that workflows can resume without issues even after failures. It is highly scalable, making it perfect for long-running workflows, managing distributed transactions, and coordinating microservices. However, its intricate architecture, the code changes required for supporting deterministic execution, and the operational burden, add up to the overall complexity of the system.

AWS Step Functions serves as a fully managed, serverless orchestration solution that streamlines workflow execution by tightly integrating with AWS services while providing built-in security, error management, and scalability. While it is a great option for AWS-native applications, it can incur high expenses for large-scale workflows, offers limited flexibility due to the constraints of Amazon States Language (ASL), and can lead to vendor lock-in.

Selecting the most appropriate workflow orchestration tool depends on certain business requirements, architectural considerations, and operational limitations. Airflow is a reliable option for ETL and batch processing, Conductor excels in microservices orchestration, Temporal is best suited for stateful long-running workflows, and AWS Step Functions is a great fit for serverless cloud automation. While each of these frameworks has its own limitations, they improve the overall reliability, scalability, and automation of workflows within modern distributed systems.

ACKNOWLEDGEMENT

The authors would like to thank open-source and cloud community for developing and contributing to the Workflow Orchestration Frameworks used by the industry.

6. REFERENCES

[1] Anas Nadeem and Muhammad Zubair Malik. 2022. A case for microservices orchestration using workflow engines. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '22). Association for Computing Machinery, New York, NY, USA, 6–10. https://doi.org/10.1145/3510455.3512777

[2] A. Barker, J. B. Weissman and J. van Hemert, "Orchestrating Data-Centric Workflows," *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, Lyon, France, 2008, pp. 210-217, doi: 10.1109/CCGRID.2008.50

[3] Stoilov, Todor, and Krasimira Stoilova. "Automation in business processes." *Proceedings of the 20th International Conference SAER-2006*. 2006

[4] Chen, Wei, et al. "Developing a concurrent service orchestration engine based on event-driven architecture." *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008

[5] Pandey, Suraj, Dileban Karunamoorthy, and Rajkumar Buyya. "Workflow engine for clouds." *Cloud computing: principles and paradigms* (2011): 321-344

[6] Apache Software Foundation, "Apache Airflow Overview," Accessed: Feb 2025. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html

[7] Run.ai, "Apache Airflow: Use Cases," Accessed: Feb 2025. [Online]. Available: https://www.run.ai/guides/machine-learning-operations/apache-airflow#Airflow-Use-Cases

[8] Netflix, "Netflix Conductor: A Microservices Orchestrator," 2016. Accessed: Feb 2025. [Online]. Available: https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40

[9] Temporal Technologies, "Clusters Overview," Accessed: Feb 2025. [Online]. Available: https://docs.temporal.io/clusters

[10] Apache Software Foundation, "Apache Airflow Use Cases," Accessed: Feb 2025. [Online]. Available: https://airflow.apache.org/use-cases/

[11] Orkes, "Microservices Orchestration Use Cases," Accessed: Feb 2025. [Online]. Available: https://www.orkes.io/use-cases/microservices-orchestration

[12] Temporal Technologies, "Use Cases and Design Patterns," Accessed: Feb 2025. [Online]. Available: https://docs.temporal.io/evaluate/use-cases-design-patterns

[13] Amazon Web Services, "AWS Step Functions Use Cases," Accessed: Feb 2025. [Online]. Available: https://aws.amazon.com/step-functions/use-cases/

[14] Datamation, "Apache Airflow Review," 2024. Accessed: Feb 2025. [Online]. Available: https://www.datamation.com/applications/apache-airflow-review/

[15] Restack, "Apache Airflow: A Knowledge of Apache Airflow and Its Use Cases," 2024. Accessed: Feb 2025. [Online]. Available: https://www.restack.io/docs/airflow-knowledge-apache-airflow-cons

[16] Decube, "Why Apache Airflow Is Not the Best Tool for Data Quality Checks," 2024. Accessed: Feb 2025. [Online]. Available: https://www.decube.io/post/why-apache-airflow-is-not-the-best-tool-for-data-quality-checks

[17] N. Kumar, "Decoding Challenges with Netflix Conductor," 2024. Accessed: Feb 2025. [Online]. Available: https://nitish1503.medium.com/decoding-challenges-with-netflix-conductor-6a623b47291f

[18] N. Somanna, "Comparing Orchestration Frameworks: Uber's Cadence, Netflix Conductor, and Temporal," 2024. Accessed: Feb 2025. [Online]. Available: https://medium.com/%40natesh.somanna/comparing-orchestration-frameworks-ubers-cadence-netflix-conductor-and-temporal-3778cff24574

[19] A. Helton, "When Not to Use Step Functions," 2022. Accessed: Feb 2025. [Online]. Available: https://www.readysetcloud.io/blog/allen.helton/when-not-to-use-step-functions/

Copyrights @ Roman Science Publications Ins.