Workflow Engine Backend Architecture

Introduction

This document outlines the architecture of a workflow engine designed to define, manage, and execute workflows. The engine automates tasks, handles human interactions, and ensures the successful completion of processes.

Architecture Overview

Assumptions

1. High workload with unpredictable peaks (elastic resource scaling based on needs)
2. The database is already populated with workflow definitions (see the *future improvements* section on the related README.md file)

The workflow engine backend is structured following a microservices architecture, allowing each component to scale independently. Components communicate using message brokers. This enables asynchronous communication and decoupling, enhancing scalability and fault tolerance. The use of background workers with different responsibilities allows for scalable processing of tasks, enabling multiple instances to run concurrently, and it decouples workflow progression logic from task execution.

Components Description

1. **API Server**: provides endpoints for starting workflows, retrieving status, and managing tasks.
2. **Workflow Definition Database**: Stores static workflow templates, including tasks, transitions, and conditions.
3. **Workflow Instance Database**: Records runtime instances of workflows and tasks, tracking their states and data.
4. **Task Queue**: Queues new tasks to be executed by worker services.
5. **Notification Queue**: Queues task completion events and external notifications like human approvals or service triggers.
6. **Task Worker (Background) Service**: Listens to the task queue and executes tasks.
7. **Notification Worker (Background) Service**: Listens to the notification queue to process task completion and manage workflow states.

Components are (logically) glued together using a **CoreEngine** library that contains domain logic, entities and rules.

Workflow Definition

We can define workflows as Directed Acyclic Graphs (DAGs), where nodes represent tasks and edges define the processing sequence. They are defined using a structured format (JSON) stored in the Workflow Definition Database.

Task Execution

The *Task Worker Service* pulls tasks from the task queue, performs the task, which may involve computations or external API calls, and sends the task completion status to the notification queue. **Error handling** and retries should be guaranteed by requeuing mechanisms. With scalable background services, the system can **execute tasks in parallel** (when allowed) and the event-driven architecture allows it to stay on hold when waiting for **external input**.

State Management

Each workflow definition becomes an instance at runtime, which has a state and it is stored in the *Workflow Instance Database*. The same goes for the tasks. The *Notification Worker Service* updates states based on task outcomes and workflow rules. Updating states and sending messages on a queue actions need to be **transactional**: we can handle that by using patterns like the **Transactional Outbox** pattern.

Definitions and instances databases are separated to have more flexibility if we have to scale them independently, given the higher chance of the Instances one growing way more than the Definitions one.

Human Interaction

Tasks requiring human or external input in general are handled via API endpoints. The system will be on hold waiting for the external input. Upon action, the logic will take care of advancing the workflow state.

Challenges and Considerations

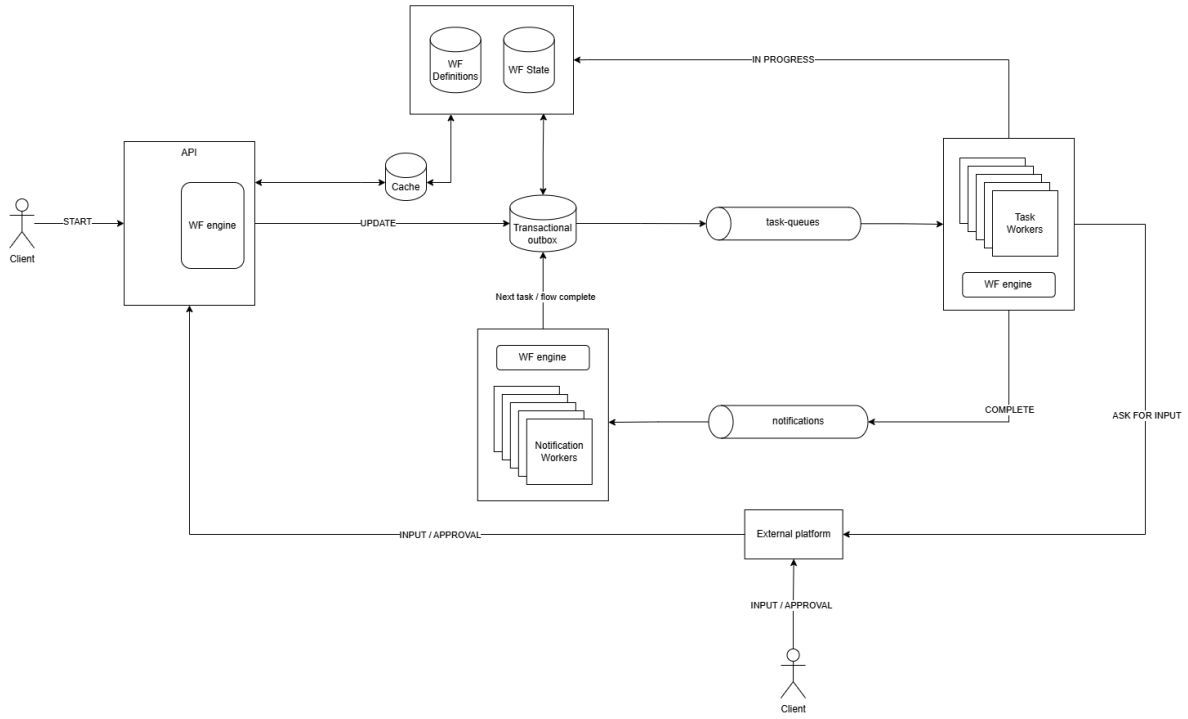
For demo purposes only, the implementation has only 1 worker service per type. Various mechanisms for horizontal autoscaling can be implemented to make sure we have enough workers during high loads.

Building a workflow engine involves tackling a variety of challenges, such as:

- **Complex Workflow Modeling**: workflows can be highly complex, involving conditional branching, parallel execution, loops, and event-driven triggers.
- **Reliability and Fault Tolerance**: ensuring the system can recover from failures without losing data or corrupting workflow states
- **Concurrency and Synchronization**: managing concurrent executions and data access to prevent race conditions and ensure data integrity

Diagrams

High-level diagram with components interactions:



Sequence diagram to showcase a start-workflow action: