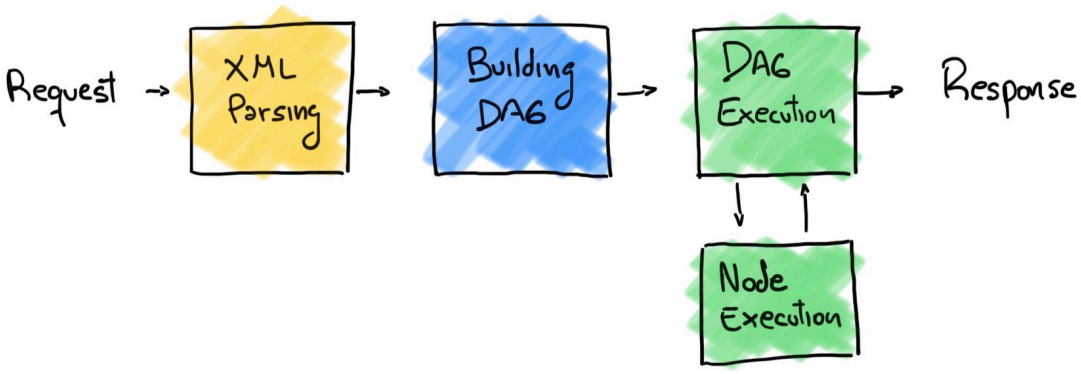
Take-Home Assignment

The following document outlines the architecture and design choices made for solving the DAG execution take-home assignment.

Author: Juan Ignacio Vimberg
Target Audience: Microsoft Interview Panel
Last Updated On: Oct 6, 2024

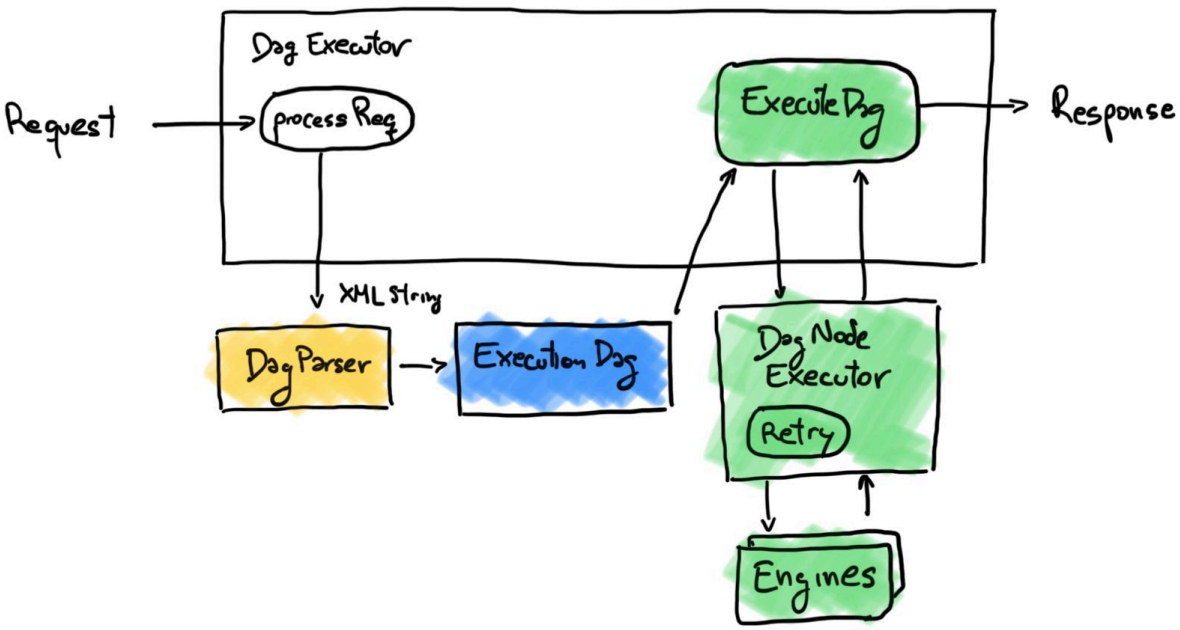
Architecture



Conceptually I split the problem in three sections:

1. **XML Parsing:** Parsing the XML input string and transforming it into objects that model nodes and dependencies.
2. **Building the Execution DAG:** Using the nodes model to generate an Execution DAG. Also, checking that the given input is valid.
3. **DAG Execution:** Executing the DAG. This is further split into two tasks:
    a. **DAG Orchestration:** Deciding which node to execute next and halting execution upon failure.
    b. **Node Execution:** Dividing node execution across M engines, simulating failures and handling retries.

On code these responsibilities are divided into the following components:



Design Choices

DAG Execution

The first step for executing a request is to generate an `ExecutionDag` from the list of nodes and dependencies provided. Note that this step inverts the DAG edges changing its semantics from: "*Node A depends on Node B"* to *"Node B executes before Node A".*

As part of this step we also check that the given input is valid, making sure there are no dependency cycles, edges pointing to non-existing nodes, etc.

Next, we submit the graph for execution. To determine which node to execute next, we use an adapted version of Kahn's algorithm for Topological Sorting. We maintain a map that tracks each node's in-degree, which represents the number of nodes that must complete execution before that node can be submitted for execution. As each node finishes, we decrement the in-degree of all its neighbors by one. When a node's in-degree reaches zero, we add it to the execution queue.

We use a number of concurrent data structures to control the execution:

- **A BlockingQueue:** To have the orchestrator thread wait until the submitted node executions complete. If a node fails executing, a message is sent to halt execution.
- **A ConcurrentHashMap:** For the in-degree map since it is updated by multiple threads concurrently.
- **An AtomicBoolean:** To record execution failure.
- **A Semaphore:** For the orchestrator to wait on the completion of all nodes before returning a result.

The execution of individual nodes is managed by the `DagNodeExecutor` class, which uses a thread pool of size M to represent the execution engines. When a task is submitted, it may randomly fail based on the configured failure rate. If a failure occurs, the `DagNodeExecutor` checks the configured `RetryStrategy` and schedules a retry accordingly. . The available retry strategies are:

- `ExponentialBackoffRetryStrategy`
- `InfiteRetryStrategy`
- `NoRetryStrategy`
- `TimedRetryStrategy`

Testing and validation

Following the Single-Responsibility Principle each part of the solution is handled by a different component. This made it easy to test each component in isolation through unit tests.

A number of test dubs were used to validate specific behaviors, for example:

- `FakeDagNodeExecutor`
    - Allows us to specify which specific nodes will fail. Useful to test that node failure is properly handled.
    - Records nodes execution order. Useful to validate that no dependencies are violated.
    - Uses a semaphore to control execution. Useful to confirm that multiple nodes are scheduled for execution in parallel.
- `FailingDagNode`
    - Allows us to specify after how many retries a node will succeed executing. Useful to validate retry strategies.

The library Awaitability is used to have tests wait until a certain condition is met. This is useful to validate time dependent behavior such as delayed retries.

To validate concurrent behavior, an integration test called `EndToEndTest` is included. Each test runs 50 times with a randomly generated input XML, which specifies the number of nodes and edges. The test evaluates different system configurations by varying the number of execution engines, failure rate, retry policy, the number of DAGs executed in parallel, and the size of the input DAG.

Possible Improvements

The current solution does not prioritize the execution of any particular DAG when there is resource contention. Nodes from all running DAGs are enqueued in a shared thread pool and processed in a first-in, first-out (FIFO) manner. Drawing from lessons learned from Polaris, a potential improvement would be to introduce configurable strategies that allow the system to make smarter decisions about which DAG node to execute next. These strategies could consider various metadata to optimize scheduling, such as DAG size, execution progress, or runtime metrics like DAG's average task time and failure rate.