Dishtha Yantra DAG Framework

Requirements Specification

Version 1.0

*Directed Acyclic Graph Compute Framework*

# 1. System Overview

## 1.1 Purpose

The Dishtha Yantra DAG Framework is a comprehensive directed acyclic graph execution engine designed for building and managing complex data processing pipelines. The framework provides a flexible, event-driven architecture that supports real-time data transformation, calculation, and routing across distributed systems.

## 1.2 Key Features

* Directed Acyclic Graph execution model with topological sorting
* Event-driven dirty flag propagation for efficient computation
* Pluggable calculators, transformers, publishers, and subscribers
* Time window-based execution control with duration support
* AutoClone feature for dynamic workload scaling
* Multi-DAG management with Zookeeper-based leader election
* Thread-safe execution with suspend/resume capabilities
* Comprehensive error tracking and monitoring

# 2. Core Architecture

## 2.1 Graph Elements

### 2.1.1 Node

The Node is the fundamental abstract base class representing a computation unit in the DAG. Each node maintains its own state and performs transformations on data.

| **Attribute** | **Description** |
| --- | --- |
| **name** | Unique identifier for the node |
| **\_input** | Dictionary containing input data from incoming edges |
| **\_output** | Dictionary containing computed output data |
| **\_isdirty** | Boolean flag indicating if node needs recomputation |
| **\_calculator** | Optional calculator instance for processing input data |
| **\_errors** | Deque storing the last 10 error occurrences with timestamps |

### 2.1.2 Edge

Edges connect nodes in the DAG and control data flow between them. Each edge can optionally transform data as it passes through.

| **Attribute** | **Description** |
| --- | --- |
| **from\_node** | Source node providing data |
| **to\_node** | Destination node receiving data |
| **data\_transformer** | Optional transformer applied to data in transit |
| **pname** | Optional pseudoname for namespacing edge data at destination |

## 2.2 Node Types

### 2.2.1 CalculationNode

Standard node that processes input through calculators and transformers. Uses the default Node compute implementation.

### 2.2.2 SubscriptionNode

Pulls data from external DataSubscriber instances. Marks itself dirty when new data is available in the subscriber queue.

* Checks subscriber queue in pre\_compute phase
* Non-blocking data retrieval with configurable timeout
* Automatically propagates changes to downstream nodes

### 2.2.3 PublicationNode

Publishes output to external DataPublisher instances when output changes.

* Tracks last published output to avoid redundant publications
* Supports multiple publishers per node
* Error handling for individual publisher failures

### 2.2.4 PublisherSinkNode

Terminal node that publishes data from incoming edges. Tracks edge data to publish only on changes.

* Maintains per-edge data tracking
* Publishes only when edge data changes
* Supports multiple publisher destinations

### 2.2.5 MetronomeNode

Time-based trigger node that executes at regular intervals using a dedicated thread.

* Configurable interval in seconds
* Daemon thread for background execution
* Optional subscriber for external data input
* Optional publishers for periodic output
* Clean shutdown support with stop event

### 2.2.6 SinkNode

Passive terminal node with no compute operations. Used as a graph endpoint that performs no processing.

## 2.3 Compute Graph

The ComputeGraph class orchestrates DAG execution, managing all nodes, edges, and external components.

### 2.3.1 Core Responsibilities

* Build and maintain the DAG structure from JSON configuration
* Perform topological sorting for correct execution order
* Manage lifecycle of subscribers, publishers, calculators, and transformers
* Execute compute loop in dedicated thread
* Support time window-based execution control
* Provide clone functionality for dynamic scaling

### 2.3.2 Execution Model

The compute loop follows a three-phase execution model:

1. **Pre-compute Phase:** Nodes check external conditions and mark themselves dirty if needed
2. **Compute Phase:** Dirty nodes execute their compute logic in topological order
3. **Post-compute Phase:** Nodes perform cleanup or logging operations

## 2.4 DAG Server

The DAGComputeServer provides multi-DAG management with high availability and dynamic scaling capabilities.

### 2.4.1 Server Features

* Load DAG configurations from file system
* Zookeeper-based leader election for high availability
* RESTful API for DAG management operations
* AutoClone feature for time-based workload scaling
* Thread-safe concurrent DAG operations
* Comprehensive status monitoring and reporting

# 3. Functional Requirements

## 3.1 Node Computation

### 3.1.1 Input Consolidation

Nodes must consolidate data from all incoming edges before computation.

* If edge has no pseudoname, merge data directly into input dictionary
* If edge has pseudoname, create nested dictionary under that key
* Support multiple edges with same pseudoname by merging their data

### 3.1.2 Change Detection

The system must implement intelligent change detection to minimize unnecessary computations.

* Compare transformed input with previous input
* Skip computation if input unchanged
* Compare output after transformations
* Propagate dirty flag to children only if output changed

### 3.1.3 Transformation Pipeline

Data transformations follow a strict pipeline order:

1. Apply input transformers to consolidated edge data
2. Execute calculator on transformed input
3. Apply output transformers to calculation result

### 3.1.4 Error Handling

* Catch and log all exceptions during computation
* Store error information with timestamp in bounded deque
* Maintain last 10 errors per node
* Continue graph execution even if individual nodes fail

## 3.2 Graph Execution

### 3.2.1 Topological Sorting

The system must maintain correct execution order through topological sorting.

* Implement Kahn's algorithm for cycle detection
* Raise exception if cycles detected in graph
* Cache sorted order for performance
* Ensure all dependencies computed before dependent nodes

### 3.2.2 Lifecycle Management

ComputeGraph must support complete lifecycle control:

* **start():** Initialize compute thread and time window monitor
* **stop():** Gracefully shutdown all threads and components
* **suspend():** Pause execution and subscriber processing
* **resume():** Resume from suspended state

## 3.3 Time Window Management

### 3.3.1 Duration-Based Configuration

The system supports flexible time window specification using duration strings.

* **Duration Format:** Support hour and minute specifications
  + '1h' = 1 hour
  + '30m' = 30 minutes
  + '1h30m' = 1 hour 30 minutes
  + '2h15m' = 2 hours 15 minutes
* **Time Format:** Support HHMM or HH:MM format for start time
* **Default Behavior:** If duration not provided, default to -5 minutes from start time

### 3.3.2 Window Enforcement

Time window monitoring thread enforces execution boundaries.

* Check current time every 60 seconds
* Compare against start\_time and end\_time
* Automatically suspend when outside window
* Automatically resume when entering window
* Support perpetual execution when no time window configured

### 3.3.3 Legacy Support

The system maintains backward compatibility with end\_time configuration.

* Support deprecated end\_time parameter
* Log warning message when end\_time used
* Recommend migration to duration-based approach

## 3.4 AutoClone Feature

### 3.4.1 Purpose

AutoClone enables dynamic workload scaling by automatically creating and managing clone DAGs during high-traffic periods. Clones run perpetually without time windows to maximize processing capacity.

### 3.4.2 Configuration

* **enabled:** Boolean to activate AutoClone
* **count:** Number of clones to create during ramp up
* **ramp\_up\_time:** Time to start creating clones (HHMM format)
* **ramp\_down\_time:** Time to start removing clones (HHMM format)

### 3.4.3 Ramp Up Process

1. Monitor current time against ramp\_up\_time
2. Create one clone per minute until count reached
3. Each clone created with no time window
4. Automatically start each clone after creation
5. Track clone names and status

### 3.4.4 Ramp Down Process

1. Monitor current time against ramp\_down\_time
2. Remove one clone per minute
3. Stop clone before deletion
4. Wait 2 seconds for graceful shutdown
5. Continue until all clones removed

### 3.4.5 Constraints

* Only primary DAG servers can create clones
* Clones cannot have AutoClone configuration
* Clones always run perpetually without time windows
* AutoClone config removed from all clones to prevent recursion

## 3.5 DAG Cloning

### 3.5.1 Clone Operation

The clone method creates independent DAG instances with optional time window override.

* Generate unique name with timestamp suffix
* Deep copy configuration
* Apply new start\_time and duration if provided
* Remove AutoClone configuration from clone
* Initialize new ComputeGraph instance

### 3.5.2 Time Window Override Rules

* If start\_time is None: Remove time window completely
* If start\_time provided, duration None: Use default duration (-5m)
* If both provided: Calculate new end\_time from start\_time + duration

# 4. Configuration Specification

## 4.1 DAG Configuration Structure

DAG configuration uses JSON format with the following top-level structure:

{

"name": "string",

"start\_time": "HHMM or HH:MM",

"duration": "XhYm",

"subscribers": [ ... ],

"publishers": [ ... ],

"calculators": [ ... ],

"transformers": [ ... ],

"nodes": [ ... ],

"edges": [ ... ],

"autoclone": { ... }

}

## 4.2 Node Configuration

* **name:** Unique identifier (required)
* **type:** Node class name or module path (default: CalculationNode)
* **config:** Node-specific configuration dictionary
* **subscriber:** Name of subscriber to attach (optional)
* **publishers:** Array of publisher names (optional)
* **calculator:** Name of calculator to use (optional)
* **input\_transformers:** Array of transformer names for input (optional)
* **output\_transformers:** Array of transformer names for output (optional)

## 4.3 Edge Configuration

* **from:** Source node name (required)
* **to:** Destination node name (required)
* **transformer:** Name of edge transformer (optional)
* **pname:** Pseudoname for namespacing at destination (optional)

## 4.4 Component Configuration

### 4.4.1 Calculator Configuration

* **name:** Unique identifier
* **type:** Calculator class or module path
* **config:** Calculator-specific parameters

### 4.4.2 Transformer Configuration

* **name:** Unique identifier
* **type:** Transformer class or module path
* **config:** Transformer-specific parameters

### 4.4.3 Subscriber Configuration

* **name:** Unique identifier
* **config:** Subscriber implementation configuration

### 4.4.4 Publisher Configuration

* **name:** Unique identifier
* **config:** Publisher implementation configuration

## 4.5 AutoClone Configuration

"autoclone": {

"enabled": true,

"count": 5,

"ramp\_up\_time": "0800",

"ramp\_down\_time": "1700"

}

# 5. DAG Server API Specifications

## 5.1 Lifecycle Management APIs

| **Method** | **Parameters** | **Description** |
| --- | --- | --- |
| **start()** | dag\_name: str | Start a specific DAG's compute loop |
| **start\_all()** | None | Start all DAGs in the server |
| **stop()** | dag\_name: str | Stop a specific DAG |
| **stop\_all()** | None | Stop all DAGs |
| **suspend()** | dag\_name: str | Suspend DAG execution |
| **resume()** | dag\_name: str | Resume suspended DAG |

## 5.2 DAG Management APIs

| **Method** | **Parameters** | **Description** |
| --- | --- | --- |
| **add\_dag()** | config\_data, config\_filename | Add new DAG from configuration |
| **delete()** | dag\_name: str | Remove DAG from server |
| **list\_dags()** | None | Return list of all DAG names |
| **get\_dag\_details()** | dag\_name: str | Return complete DAG details including nodes, edges, status |
| **clone\_dag()** | dag\_name, start\_time, duration | Create clone with optional time window override |

## 5.3 Monitoring APIs

| **Method** | **Parameters** | **Description** |
| --- | --- | --- |
| **get\_status()** | dag\_name: str | Return runtime status (running, suspended, stopped) |
| **get\_autoclone\_status()** | dag\_name: str | Return AutoClone status and clone list |
| **is\_primary()** | None | Return whether this server instance is primary |

# 6. Non-Functional Requirements

## 6.1 Performance

* Compute loop must execute with minimal latency (target: 10ms cycle time)
* Dirty flag propagation must be instant
* Topological sort cached for repeated execution
* Support hundreds of nodes per DAG

## 6.2 Reliability

* Individual node failures must not crash entire DAG
* Comprehensive error logging with timestamps
* Bounded error history to prevent memory leaks
* Graceful shutdown of all threads and resources

## 6.3 Scalability

* Support multiple DAGs per server instance
* AutoClone feature for horizontal scaling
* Thread-safe operations with proper locking
* Leader election for multi-instance deployment

## 6.4 Maintainability

* Pluggable architecture for calculators, transformers, publishers, subscribers
* JSON-based configuration for easy modification
* Comprehensive logging at appropriate levels
* Clear separation of concerns between components
* Support for custom node types via module loading

## 6.5 High Availability

* Zookeeper integration for leader election
* Automatic primary/secondary failover
* Resume operations when elected as primary
* Graceful degradation if Zookeeper unavailable

## 6.6 Thread Safety

* Comprehensive locking for DAG dictionary access
* Thread-safe AutoClone management
* Proper event signaling for suspend/resume
* Daemon threads for background operations

## 6.7 Monitoring and Observability

* Track computation count per node
* Record last computation timestamp
* Maintain error history with timestamps
* Expose detailed status via API
* Comprehensive debug-level logging

# 7. Technical Constraints

## 7.1 Platform Requirements

* Python 3.7 or higher
* Threading support required
* JSON configuration file support

## 7.2 External Dependencies

* Kazoo library for Zookeeper integration
* Zookeeper cluster for high availability (optional)
* Custom calculator, transformer, publisher, subscriber implementations

## 7.3 Graph Constraints

* Must be directed acyclic graph (no cycles allowed)
* Node names must be unique within a DAG
* Edge names auto-generated as from\_to pattern
* Referenced components must exist in configuration

# 8. Implementation Notes

## 8.1 Dirty Flag Propagation

The dirty flag mechanism is central to the framework's efficiency. When a node's output changes, it automatically marks all downstream nodes as dirty, triggering their recomputation in the next cycle. This event-driven approach minimizes unnecessary computations.

## 8.2 Deep Copy Strategy

The framework uses deep copy for input and output data to ensure complete isolation between nodes. This prevents unintended side effects but requires careful consideration of data structures for performance.

## 8.3 Module Loading

Custom components can be specified using module.path.ClassName notation. The framework uses dynamic instantiation to load and initialize these components at runtime.

## 8.4 Time Window Precision

Time window checks occur every 60 seconds, providing minute-level precision. For applications requiring second-level precision, this interval can be adjusted with performance trade-offs.

## 8.5 AutoClone Rate Limiting

AutoClone creates and destroys one clone per minute to avoid resource spikes. This gradual scaling ensures stable system behavior during ramp up and down periods.

## 8.6 Error Isolation

Each node maintains its own error deque, preventing error propagation across the graph. The compute loop continues execution even if individual nodes fail, maximizing overall system resilience.