Dishtha Yantra Calculator Framework

Requirements Specification

*Version 1.0
Calculator-Node Integration System*

# 1. Executive Summary

The Dishtha Yantra Calculator Framework is a modular, extensible computation system designed for building directed acyclic graph (DAG) based data processing pipelines. The framework provides a clean separation between computation logic (Calculators) and graph structure (Nodes and Edges), enabling flexible and reusable data transformation pipelines.

This document focuses on the Calculator subsystem and its integration with the Node infrastructure, providing detailed specifications for implementation, extension, and usage.

# 2. System Overview

## 2.1 Architecture Components

The framework consists of three primary components:

* **Graph Elements:** Node and Edge classes that form the computation graph structure
* **Calculators:** Pluggable computation units that perform data transformations
* **Node Implementations:** Specialized node types for different use cases (subscription, publication, calculation, etc.)

## 2.2 Data Flow Model

Data flows through the system in a structured pipeline:

1. Edge connections propagate output data from source nodes to target nodes
2. Input transformers process incoming edge data
3. Calculators perform core computations on transformed input
4. Output transformers process calculator results
5. Output propagates to connected child nodes via edges

# 3. Calculator Framework Requirements

## 3.1 Abstract Base Class: DataCalculator

All calculators must inherit from the abstract DataCalculator base class, which provides:

| **Attribute** | **Description** |
| --- | --- |
| **name** | Unique identifier for the calculator instance |
| **config** | Configuration dictionary containing calculator-specific parameters |
| **\_calculation\_count** | Counter tracking number of calculations performed |
| **\_last\_calculation** | ISO timestamp of the most recent calculation |

**Required Methods:**

* **calculate(data):** Abstract method that must be implemented by all concrete calculators. Receives input dictionary and returns output dictionary.
* **details():** Returns calculator metadata including name, type, calculation count, and last calculation timestamp in JSON format.

# 4. Calculator Implementations

The framework provides eight concrete calculator implementations for common data transformation operations.

## 4.1 NullCalculator

**Purpose:** Returns a deep copy of the input data without modification.

**Use Cases:**

* Creating data isolation points in the computation graph
* Preventing unintended data mutations
* Testing and debugging data flow

**Configuration:** No configuration parameters required.

**Behavior:** Creates a complete deep copy of the input dictionary using copy.deepcopy(), ensuring no references to original data objects are retained.

## 4.2 PassthruCalculator

**Purpose:** Returns the input data as-is without copying or modification.

**Use Cases:**

* Performance-critical paths where copying is unnecessary
* Pass-through nodes that don't require data transformation
* Placeholder calculators during development

**Configuration:** No configuration parameters required.

**Behavior:** Returns the input dictionary reference directly without any copying. Subsequent modifications to the data will affect all references.

## 4.3 AttributeFilterAwayCalculator

**Purpose:** Removes specified attributes from the input data.

**Use Cases:**

* Data sanitization and privacy filtering
* Removing sensitive or unnecessary fields
* Reducing data payload size

**Configuration:**

* **filter\_attributes:** List of attribute names to remove from the input data

**Behavior:** Creates a deep copy of input data and removes all attributes listed in filter\_attributes. Non-existent attributes are silently ignored.

## 4.4 AttributeFilterCalculator

**Purpose:** Keeps only specified attributes, removing all others.

**Use Cases:**

* Selecting specific fields for downstream processing
* Creating focused data views
* API response filtering

**Configuration:**

* **keep\_attributes:** List of attribute names to retain in the output

**Behavior:** Creates a new dictionary containing only the attributes listed in keep\_attributes. Non-existent attributes are silently omitted from the result.

## 4.5 ApplyDefaultsCalculator

**Purpose:** Applies default values for missing or None attributes.

**Use Cases:**

* Ensuring required fields have values
* Configuration management with fallback values
* Data normalization and validation

**Configuration:**

* **defaults:** Dictionary mapping attribute names to default values

**Behavior:** Creates a deep copy of input data. For each key in the defaults dictionary, if the key is missing from input or has a None value, the default value is applied.

## 4.6 AdditionCalculator

**Purpose:** Sums numeric values from specified attributes.

**Use Cases:**

* Calculating totals and aggregations
* Financial computations
* Metric accumulation

**Configuration:**

* **arguments:** List of attribute names whose values should be summed
* **output\_attribute:** Name of the output attribute (default: 'result')

**Behavior:** Attempts to convert each argument value to float and sum them. Non-numeric or missing values are skipped with a warning. Returns dictionary with single key containing the sum.

## 4.7 MultiplicationCalculator

**Purpose:** Multiplies numeric values from specified attributes.

**Use Cases:**

* Calculating products and ratios
* Unit conversions
* Scaling operations

**Configuration:**

* **arguments:** List of attribute names whose values should be multiplied
* **output\_attribute:** Name of the output attribute (default: 'result')

**Behavior:** Starts with product = 1, then multiplies each argument value after converting to float. Non-numeric or missing values are skipped with a warning. Returns dictionary with single key containing the product.

## 4.8 AttributeNameChangeCalculator

**Purpose:** Renames attributes according to a mapping configuration.

**Use Cases:**

* API response standardization
* Schema transformation
* Data migration and compatibility layers

**Configuration:**

* **name\_mapping:** Dictionary mapping old attribute names to new attribute names

**Behavior:** Creates a deep copy of input data and renames attributes according to the mapping. Only attributes present in the input are renamed; missing attributes are ignored.

# 5. Integration with Nodes

## 5.1 Calculator Assignment

Calculators are attached to Node instances using the set\_calculator() method:

node.set\_calculator(calculator\_instance)

## 5.2 Computation Pipeline

During node computation, the calculator is invoked as part of a multi-stage pipeline:

1. **Edge Data Collection:** Node gathers input from all incoming edges
2. **Input Transformation:** Input transformers process the merged edge data
3. **Calculator Execution:** If calculator is set, it processes the transformed input
4. **Output Transformation:** Output transformers process calculator results
5. **Propagation:** Changed output triggers dirty flag on connected child nodes

## 5.3 Optional Calculator Usage

Calculators are optional components. If no calculator is set on a node:

* The node passes transformed input directly to output
* This allows nodes to function as pure data routing or transformation points

## 5.4 Node Types and Calculator Usage

| **Node Type** | **Calculator Support** | **Primary Purpose** |
| --- | --- | --- |
| Node (base) | Yes - via set\_calculator() | General computation with edge-based data flow |
| CalculationNode | Yes - uses Node.compute() | Dedicated calculation with default behavior |
| SubscriptionNode | Yes - processes subscriber data | Pull data from external subscribers |
| PublicationNode | Yes - via parent compute() | Push computed output to publishers |
| MetronomeNode | Yes - interval-based execution | Periodic calculation and publication |
| SinkNode | No - compute() is no-op | Terminal node with no computation |
| PublisherSinkNode | No - direct edge publishing | Publish edge data changes without calculation |

# 6. Configuration Requirements

## 6.1 Calculator Instantiation

All calculators must be instantiated with two parameters:

* **name:** String identifier for the calculator instance
* **config:** Dictionary containing calculator-specific configuration parameters

## 6.2 Configuration Dictionary Structure

The config dictionary must contain only parameters relevant to the specific calculator type. Required and optional parameters are defined in the calculator specifications (Section 4).

**Example Configuration:**

# Addition Calculator

config = {

'arguments': ['price', 'tax', 'shipping'],

'output\_attribute': 'total\_cost'

}

calculator = AdditionCalculator('cost\_calc', config)

# Attribute Filter Calculator

config = {

'keep\_attributes': ['user\_id', 'username', 'email']

}

calculator = AttributeFilterCalculator('user\_filter', config)

# 7. Error Handling and Monitoring

## 7.1 Calculator Error Behavior

Calculators should handle errors gracefully:

* **Type Conversion Errors:** Numeric calculators log warnings for non-convertible values and skip them
* **Missing Attributes:** Silently ignored unless required by calculator logic
* **Exception Propagation:** Unhandled exceptions are caught by Node.compute() and logged to error deque

## 7.2 Node-Level Error Tracking

Each node maintains error history:

* **\_errors:** Deque with maxlen=10 stores recent error information
* **Error Structure:** Each error entry contains timestamp and error message
* **Access:** Error history available via node.details()['errors']

## 7.3 Monitoring Metrics

Calculators and nodes provide monitoring data:

* **Calculation Count:** Total number of calculate() invocations
* **Last Calculation Time:** ISO timestamp of most recent calculation
* **Compute Count:** Node-level computation count
* **Last Compute Time:** Node-level computation timestamp

# 8. Extension Guidelines

## 8.1 Creating Custom Calculators

To create a custom calculator:

1. Inherit from DataCalculator abstract base class
2. Implement the calculate(data) method
3. Call parent \_\_init\_\_(name, config) in constructor
4. Increment \_calculation\_count and update \_last\_calculation
5. Return output dictionary from calculate() method

## 8.2 Best Practices

* **Immutability:** Use copy.deepcopy() when modifying input data to prevent side effects
* **Type Safety:** Validate input types and handle conversion errors gracefully
* **Configuration Validation:** Verify required config parameters in \_\_init\_\_()
* **Logging:** Use logger.warning() for recoverable issues, logger.error() for failures
* **Documentation:** Include docstrings explaining purpose, config parameters, and return format
* **Performance:** Consider computational complexity for large datasets

# 9. Usage Examples

## 9.1 Simple Calculation Pipeline

# Create nodes

input\_node = CalculationNode('input', {})

calc\_node = CalculationNode('calculator', {})

output\_node = CalculationNode('output', {})

# Configure calculator

calc\_config = {

'arguments': ['a', 'b', 'c'],

'output\_attribute': 'sum'

}

calculator = AdditionCalculator('adder', calc\_config)

calc\_node.set\_calculator(calculator)

# Create edges

Edge(input\_node, calc\_node)

Edge(calc\_node, output\_node)

## 9.2 Data Transformation Chain

# Filter, transform, then calculate

filter\_node = CalculationNode('filter', {})

rename\_node = CalculationNode('rename', {})

compute\_node = CalculationNode('compute', {})

# Setup calculators

filter\_calc = AttributeFilterCalculator('filter',

{'keep\_attributes': ['price', 'quantity']})

filter\_node.set\_calculator(filter\_calc)

rename\_calc = AttributeNameChangeCalculator('rename',

{'name\_mapping': {'price': 'unit\_price', 'quantity': 'qty'}})

rename\_node.set\_calculator(rename\_calc)

compute\_calc = MultiplicationCalculator('multiply',

{'arguments': ['unit\_price', 'qty'],

'output\_attribute': 'total'})

compute\_node.set\_calculator(compute\_calc)

# 10. Conclusion

The Dishtha Yantra Calculator Framework provides a robust, modular architecture for building data processing pipelines. The separation of concerns between graph structure (Nodes/Edges) and computation logic (Calculators) enables:

* **Reusability:** Calculators can be reused across different nodes and graphs
* **Extensibility:** New calculators can be added without modifying core framework
* **Testability:** Calculators can be unit tested independently of graph structure
* **Composability:** Complex transformations built from simple calculator chains
* **Maintainability:** Clear interfaces and separation of concerns simplify debugging

*— End of Document —*