Dishtha Yantra

PubSub Framework

Requirements Specification Document

*Version 1.0*

*10/27/2025*

# Document Information

**Document Title:** Dishtha Yantra PubSub Framework - Requirements Specification

**Version:** 1.0

**Status:** Draft

**Classification:** Technical Requirements

# Executive Summary

The Dishtha Yantra PubSub Framework is a comprehensive, enterprise-grade publish-subscribe messaging framework designed to provide a unified abstraction layer over multiple messaging technologies. The framework enables seamless integration with various messaging systems including message queues, topics, databases, file systems, and network protocols through a consistent API interface.

Key highlights include support for 15+ messaging backends, thread-safe operations, configurable batching and buffering, monitoring capabilities, and extensibility through custom implementations. The framework is implemented in Python and follows object-oriented design principles with abstract base classes and factory patterns.

# 1. Introduction

## 1.1 Purpose

This document defines the functional and non-functional requirements for the Dishtha Yantra PubSub Framework. It serves as a comprehensive specification for developers, architects, and stakeholders involved in implementing, maintaining, or extending the framework.

## 1.2 Scope

The framework provides:

* Unified publish-subscribe interface for heterogeneous messaging systems
* Support for both queue-based (point-to-point) and topic-based (publish-subscribe) messaging patterns
* Thread-safe, production-ready implementations for multiple messaging backends
* Configuration-driven architecture with factory pattern for component creation
* Monitoring and operational visibility through detailed metrics and logging

## 1.3 Target Audience

* Software architects designing distributed systems
* Developers implementing messaging solutions
* DevOps engineers managing messaging infrastructure
* System integrators connecting disparate systems

# 2. System Overview

## 2.1 Framework Architecture

The Dishtha Yantra PubSub Framework follows a layered architecture with clear separation of concerns:

### 2.1.1 Abstraction Layer

* **DataPublisher:** Abstract base class defining the publisher contract
* **DataSubscriber:** Abstract base class defining the subscriber contract
* **DataAwarePayload:** Envelope class for dynamic routing and metadata

### 2.1.2 Implementation Layer

Concrete implementations for each supported messaging backend, implementing the abstract contracts while handling backend-specific nuances.

### 2.1.3 Factory Layer

Factory methods (create\_publisher, create\_subscriber) that instantiate appropriate implementations based on URI-style destination/source specifications.

## 2.2 Design Principles

* **Abstraction:** Hide backend complexity behind uniform interfaces
* **Extensibility:** Support custom implementations through well-defined contracts
* **Configuration-driven:** Minimize code changes through external configuration
* **Thread-safety:** Built-in concurrency controls for multi-threaded environments
* **Observability:** Comprehensive logging and metrics for operational visibility

# 3. Core Components

## 3.1 DataPublisher

### 3.1.1 Purpose

Abstract base class that defines the contract for all publisher implementations. Manages message publishing, batching, periodic flushing, and operational metrics.

### 3.1.2 Key Attributes

* **name:** String identifier for the publisher instance
* **destination:** URI-style string specifying the target endpoint
* **config:** Dictionary containing configuration parameters
* **publish\_interval:** Seconds between periodic flush operations (0 = immediate)
* **batch\_size:** Number of messages to accumulate before forced flush
* **\_publish\_queue:** Internal queue for buffering messages when batching is enabled
* **\_publish\_count:** Counter tracking total published messages

### 3.1.3 Methods

* **publish(data):** Public method to publish data; handles batching logic
* **\_do\_publish(data):** Abstract method; must be implemented by subclasses
* **details():** Returns JSON-serializable metrics dictionary
* **stop():** Graceful shutdown with queue flush

## 3.2 DataSubscriber

### 3.2.1 Purpose

Abstract base class for all subscriber implementations. Manages background subscription threads, internal message queuing, suspend/resume capabilities, and receive metrics.

### 3.2.2 Key Attributes

* **name:** String identifier for the subscriber instance
* **source:** URI-style string specifying the data source
* **config:** Dictionary containing configuration parameters
* **max\_depth:** Maximum size of internal message queue
* **\_internal\_queue:** Thread-safe queue for buffering received messages
* **\_suspend\_event:** Threading event for pause/resume control
* **\_receive\_count:** Counter tracking total received messages

### 3.2.3 Methods

* **start():** Initiates background subscription thread
* **\_do\_subscribe():** Abstract method; must be implemented by subclasses
* **get\_data(block\_time):** Retrieves message from internal queue with optional blocking
* **suspend():** Pauses message reception
* **resume():** Resumes message reception
* **get\_queue\_size():** Returns current depth of internal queue
* **details():** Returns JSON-serializable metrics dictionary
* **stop():** Graceful shutdown of subscription thread

## 3.3 DataAwarePayload

### 3.3.1 Purpose

Envelope class that encapsulates payload data along with destination routing information and custom data elements (CDE). Enables dynamic routing where the destination can be determined at publish time rather than publisher initialization.

### 3.3.2 Attributes

* **destination:** Target destination for this specific message
* **cde:** Dictionary of custom data elements (metadata)
* **payload:** Actual message data

### 3.3.3 Methods

* **add\_to\_cde(key, value):** Adds metadata to CDE dictionary
* **get\_data\_for\_publication():** Returns (destination, data) tuple formatted for publishing
* **to\_dict():** Serializes to dictionary format

# 4. Supported Messaging Systems

The framework provides native implementations for the following messaging technologies:

## 4.1 In-Memory (mem://)

* **URI Format:** mem://queue/<name> or mem://topic/<name>
* **Use Case:** Testing, development, and single-process messaging
* **Features:** Singleton pattern, thread-safe, no external dependencies
* **Configuration:** max\_size (queue capacity)

## 4.2 File System (file://)

* **URI Format:** file:///path/to/file
* **Use Case:** Logging, audit trails, inter-process communication via filesystem
* **Features:** Append-only publishing, tail-following subscription
* **Configuration:** read\_interval (polling frequency for subscriber)

## 4.3 Apache Kafka (kafka://)

* **URI Format:** kafka://topic/<topic\_name>
* **Use Case:** High-throughput distributed messaging, event streaming
* **Features:** Distributed, fault-tolerant, persistent, replay capability
* **Dependencies:** kafka-python library
* **Configuration:** bootstrap\_servers, group\_id, producer\_config, consumer\_config

## 4.4 ActiveMQ (activemq://)

* **URI Format:** activemq://queue/<name> or activemq://topic/<name>
* **Use Case:** Enterprise messaging, JMS integration
* **Features:** STOMP protocol support, both queue and topic patterns
* **Dependencies:** stomp.py library
* **Configuration:** host, port (61613), username, password

## 4.5 RabbitMQ (rabbitmq://)

* **URI Format:** rabbitmq://queue/<name> or rabbitmq://topic/<name>
* **Use Case:** Flexible routing, reliable delivery, enterprise messaging
* **Features:** Exchange patterns, wildcard routing, persistent messages, automatic reconnection
* **Dependencies:** pika library
* **Configuration:** host, port (5672), username, password, virtual\_host, exchange, binding\_key

## 4.6 Redis (redis:// and redischannel://)

* **URI Format:** redis://<key> or redischannel://<channel>
* **Use Case:** High-performance caching, pub/sub, key-value storage
* **Features:** In-memory speed, pub/sub channels, key-based storage
* **Dependencies:** redis-py library
* **Configuration:** host, port (6379), db, \_\_dagserver\_key (for key-value mode)

## 4.7 AshRedis (ashredis:// and ashredischannel://)

* **URI Format:** ashredis:// or ashredischannel://<channel>
* **Use Case:** Custom Redis-compatible implementation with region support
* **Features:** Multi-region support, TTL, channel subscriptions
* **Dependencies:** Custom AshRedisClient
* **Configuration:** host, port, region, ttl\_seconds, \_\_dagserver\_key

## 4.8 SQL Database (sql://)

* **URI Format:** sql://<table\_name>
* **Use Case:** Persistent message storage, database-driven workflows
* **Features:** ACID guarantees, SQL query support, polling-based subscription
* **Dependencies:** pymysql or psycopg2
* **Configuration:** sql\_config\_file, db\_type (mysql/postgres), host, port, user, password, database, insert\_statement, select\_statement, poll\_interval

## 4.9 REST API (rest://)

* **URI Format:** rest://<endpoint>
* **Use Case:** HTTP-based integration, webhook delivery, API consumption
* **Features:** Multiple auth methods, retry logic, polling or push modes
* **Dependencies:** requests library
* **Configuration:** rest\_config\_file with base\_url, endpoints, http\_method, auth\_type (none/basic/bearer/api\_key), headers, timeout, verify\_ssl, max\_retries

## 4.10 gRPC (grpc://)

* **URI Format:** grpc://<host>:<port>/<topic>
* **Use Case:** High-performance RPC, microservices communication
* **Features:** Streaming support, protocol buffers, SSL/TLS
* **Dependencies:** grpcio, generated protobuf stubs
* **Configuration:** host\_port, use\_ssl, timeout, max\_retries

## 4.11 Aerospike (aerospike://)

* **URI Format:** aerospike://<namespace>/<set>
* **Use Case:** High-performance NoSQL database writes
* **Features:** Sub-millisecond latency, horizontal scaling
* **Dependencies:** aerospike library
* **Configuration:** hosts [(host, port)], \_\_dagserver\_key

## 4.12 Custom Implementation (custom://)

* **URI Format:** custom://<identifier>
* **Use Case:** Integration with proprietary or unsupported systems
* **Features:** Delegate pattern for custom logic
* **Configuration:** delegate\_module, delegate\_class, delegate\_config

## 4.13 Metronome (metronome)

* **URI Format:** metronome
* **Use Case:** Scheduled message generation, heartbeats, time-triggered events
* **Features:** Automatic periodic message emission with timestamps
* **Configuration:** interval (seconds), message (payload content)

# 5. Functional Requirements

## 5.1 FR-01: Publisher Message Transmission

* **Priority:** Critical
* **Description:** Publishers MUST transmit messages to configured destinations
* **Acceptance Criteria:** 
  + Messages are successfully delivered to the backend system
  + Publish count metrics increment correctly
  + Errors are logged and exceptions raised appropriately

## 5.2 FR-02: Subscriber Message Reception

* **Priority:** Critical
* **Description:** Subscribers MUST receive messages from configured sources
* **Acceptance Criteria:** 
  + Messages are retrieved from backend systems
  + Messages are placed in internal queue for consumer retrieval
  + Receive count metrics increment correctly
  + Backpressure handling when queue reaches max\_depth

## 5.3 FR-03: Batching and Buffering

* **Priority:** High
* **Description:** Publishers MUST support configurable batching
* **Acceptance Criteria:** 
  + Messages accumulate in internal queue when publish\_interval > 0
  + Automatic flush when batch\_size threshold reached
  + Periodic flush at publish\_interval frequency
  + Flush on stop() for clean shutdown

## 5.4 FR-04: Dynamic Routing via DataAwarePayload

* **Priority:** Medium
* **Description:** Publishers MUST support DataAwarePayload for per-message routing
* **Acceptance Criteria:** 
  + Detect DataAwarePayload type in \_do\_publish()
  + Extract destination and data via get\_data\_for\_publication()
  + Route to specified destination or fall back to default
  + Include CDE metadata in published data structure

## 5.5 FR-05: Thread-Safe Operations

* **Priority:** Critical
* **Description:** All components MUST be thread-safe
* **Acceptance Criteria:** 
  + Use threading.Lock for shared state access
  + Thread-safe queue implementations (queue.Queue)
  + No race conditions under concurrent usage

## 5.6 FR-06: Subscriber Lifecycle Management

* **Priority:** High
* **Description:** Subscribers MUST support start/stop/suspend/resume operations
* **Acceptance Criteria:** 
  + start() initiates background thread
  + suspend() blocks message reception without stopping thread
  + resume() unblocks message reception
  + stop() gracefully terminates thread within timeout

## 5.7 FR-07: Metrics and Monitoring

* **Priority:** High
* **Description:** Components MUST expose operational metrics
* **Acceptance Criteria:** 
  + details() method returns JSON-serializable dictionary
  + Publisher metrics: name, destination, publish\_count, last\_publish, queue\_depth
  + Subscriber metrics: name, source, receive\_count, last\_receive, current\_depth, suspended state

## 5.8 FR-08: Factory-Based Instantiation

* **Priority:** High
* **Description:** Framework MUST provide factory methods for component creation
* **Acceptance Criteria:** 
  + create\_publisher(name, config) instantiates correct publisher type
  + create\_subscriber(name, config) instantiates correct subscriber type
  + URI scheme determines implementation class
  + Validation of required configuration parameters

## 5.9 FR-09: Error Handling and Logging

* **Priority:** Critical
* **Description:** Framework MUST handle errors gracefully with comprehensive logging
* **Acceptance Criteria:** 
  + All exceptions logged with appropriate severity
  + Connection failures trigger retry logic (where applicable)
  + Transient errors don't crash background threads
  + Critical errors propagate to caller

## 5.10 FR-10: Connection Management

* **Priority:** High
* **Description:** Network-based implementations MUST manage connections reliably
* **Acceptance Criteria:** 
  + Automatic reconnection on connection loss
  + Configurable retry attempts and delays
  + Connection pooling where supported
  + Graceful cleanup on shutdown

# 6. Non-Functional Requirements

## 6.1 NFR-01: Performance

* **Publisher Throughput:** MUST support at least 1000 messages/second per publisher instance for in-memory/file backends
* **Subscriber Throughput:** MUST support at least 1000 messages/second per subscriber instance
* **Latency:** P95 latency SHOULD be under 10ms for local backends, under 100ms for network backends
* **Memory:** Internal queues MUST be bounded to prevent memory exhaustion

## 6.2 NFR-02: Scalability

* **Concurrent Components:** MUST support 100+ publishers and subscribers per process
* **Message Size:** MUST handle messages up to 10MB (subject to backend limits)
* **Queue Depth:** Configurable to at least 1,000,000 messages

## 6.3 NFR-03: Reliability

* **Message Delivery:** At-least-once delivery semantics (subject to backend capabilities)
* **Fault Tolerance:** Automatic retry with exponential backoff for transient failures
* **Data Integrity:** No message corruption during transmission

## 6.4 NFR-04: Maintainability

* **Code Quality:** Python PEP 8 compliance
* **Documentation:** Inline docstrings for all public methods
* **Logging:** Structured logging at appropriate levels (DEBUG, INFO, WARNING, ERROR)
* **Testing:** Unit test coverage of 80%+ for core components

## 6.5 NFR-05: Extensibility

* **Plugin Architecture:** Custom implementations via custom:// protocol
* **Interface Stability:** Abstract base class interfaces MUST remain backward compatible
* **Configuration:** All backend-specific settings via config dictionaries

## 6.6 NFR-06: Security

* **Authentication:** Support for username/password, token-based auth where applicable
* **Encryption:** TLS/SSL support for network protocols
* **Credentials:** No hardcoded credentials; configuration-based only
* **Validation:** Input validation to prevent injection attacks

# 7. Configuration Requirements

## 7.1 Common Publisher Configuration

* **name:** (string, required) Unique identifier
* **destination:** (string, required) URI-format destination
* **publish\_interval:** (int, optional, default=0) Seconds between flushes
* **batch\_size:** (int, optional) Messages to accumulate before flush

## 7.2 Common Subscriber Configuration

* **name:** (string, required) Unique identifier
* **source:** (string, required) URI-format source
* **max\_depth:** (int, optional, default=100000) Internal queue size

## 7.3 Backend-Specific Configuration Examples

### 7.3.1 Kafka Configuration

*Example configuration for Kafka publisher:*

{

"name": "my\_kafka\_pub",

"destination": "kafka://topic/events",

"bootstrap\_servers": ["kafka1:9092", "kafka2:9092"],

"producer\_config": {

"acks": "all",

"retries": 3

}

}

### 7.3.2 RabbitMQ Configuration

*Example configuration for RabbitMQ subscriber:*

{

"name": "my\_rabbit\_sub",

"source": "rabbitmq://topic/notifications",

"host": "rabbitmq.example.com",

"port": 5672,

"username": "guest",

"password": "guest",

"exchange": "amq.topic",

"binding\_key": "app.\*.critical"

}

### 7.3.3 REST API Configuration

*REST configurations require a separate JSON file referenced in config:*

{

"name": "my\_rest\_pub",

"destination": "rest://webhook",

"rest\_config\_file": "/path/to/rest\_config.json"

}

# 8. Use Cases

## 8.1 UC-01: Event Distribution System

* **Actor:** Microservices application
* **Scenario:** Distribute domain events across multiple services
* **Implementation:** 
  + Create Kafka publisher for event sourcing
  + Multiple subscribers listen to different topic partitions
  + Use consumer groups for load balancing

## 8.2 UC-02: Data Pipeline

* **Actor:** ETL system
* **Scenario:** Extract data from REST API, transform, and load to database
* **Implementation:** 
  + REST subscriber polls API endpoint
  + Transform logic processes data
  + SQL publisher inserts transformed data
  + Batching enabled for efficient database writes

## 8.3 UC-03: Hybrid Messaging Architecture

* **Actor:** Enterprise integration platform
* **Scenario:** Bridge between legacy ActiveMQ and modern Kafka
* **Implementation:** 
  + ActiveMQ subscriber receives from legacy systems
  + Business logic transformation layer
  + Kafka publisher forwards to modern applications
  + DataAwarePayload for dynamic topic routing

## 8.4 UC-04: Development and Testing

* **Actor:** Development team
* **Scenario:** Test application without external dependencies
* **Implementation:** 
  + Use in-memory publishers/subscribers for unit tests
  + File-based for integration testing
  + Swap to production backends via configuration changes
  + No code changes required between environments

## 8.5 UC-05: Real-Time Monitoring Dashboard

* **Actor:** Monitoring application
* **Scenario:** Display live metrics from multiple sources
* **Implementation:** 
  + Redis channel subscribers for real-time metrics
  + Dashboard polls details() method for status
  + Metronome for heartbeat verification
  + Suspend/resume controls for backpressure management

# 9. API Specifications

## 9.1 DataPublisher API

### 9.1.1 \_\_init\_\_(name, destination, config)

* **Parameters:**
  + name (str): Unique identifier
  + destination (str): URI-format destination
  + config (dict): Configuration parameters
* **Returns:** Publisher instance

### 9.1.2 publish(data)

* **Parameters:**
  + data (dict or DataAwarePayload): Message to publish
* **Returns:** None
* **Behavior:** Queues message if batching enabled, otherwise publishes immediately

### 9.1.3 details()

* **Returns:** dict with keys: name, destination, publish\_interval, batch\_size, last\_publish, publish\_count, queue\_depth

### 9.1.4 stop()

* **Returns:** None
* **Behavior:** Flushes pending messages, stops background threads, closes connections

## 9.2 DataSubscriber API

### 9.2.1 \_\_init\_\_(name, source, config)

* **Parameters:**
  + name (str): Unique identifier
  + source (str): URI-format source
  + config (dict): Configuration parameters
* **Returns:** Subscriber instance

### 9.2.2 start()

* **Returns:** None
* **Behavior:** Initiates background subscription thread

### 9.2.3 get\_data(block\_time=None)

* **Parameters:**
  + block\_time (float or None): None=non-blocking, -1=block indefinitely, >0=timeout seconds
* **Returns:** dict or None

### 9.2.4 suspend() / resume()

* **Returns:** None
* **Behavior:** Pauses/resumes message reception

### 9.2.5 get\_queue\_size()

* **Returns:** int - current number of messages in internal queue

### 9.2.6 details()

* **Returns:** dict with keys: name, source, max\_depth, current\_depth, last\_receive, receive\_count, suspended

### 9.2.7 stop()

* **Returns:** None
* **Behavior:** Stops subscription thread, closes connections

## 9.3 Factory Functions

### 9.3.1 create\_publisher(name, config)

* **Parameters:**
  + name (str): Publisher name
  + config (dict): Must include 'destination' key
* **Returns:** DataPublisher instance
* **Raises:** ValueError if destination unsupported or config invalid

### 9.3.2 create\_subscriber(name, config)

* **Parameters:**
  + name (str): Subscriber name
  + config (dict): Must include 'source' key
* **Returns:** DataSubscriber instance
* **Raises:** ValueError if source unsupported or config invalid

# 10. Appendices

## 10.1 Appendix A: URI Scheme Reference

|  |  |  |
| --- | --- | --- |
| **URI Scheme** | **Pattern** | **Example** |
| In-Memory Queue | mem://queue/<name> | mem://queue/orders |
| In-Memory Topic | mem://topic/<name> | mem://topic/events |
| File System | file://<path> | file:///tmp/log.jsonl |
| Kafka | kafka://topic/<name> | kafka://topic/events |
| ActiveMQ Queue | activemq://queue/<name> | activemq://queue/orders |
| RabbitMQ Topic | rabbitmq://topic/<name> | rabbitmq://topic/notifications |
| Redis Key | redis:// | redis:// |
| Redis Channel | redischannel://<name> | redischannel://updates |
| SQL Database | sql://<table> | sql://messages |
| REST API | rest://<endpoint> | rest://webhook |
| gRPC | grpc://<host>:<port>/<topic> | grpc://localhost:50051/events |
| Metronome | metronome | metronome |

## 10.2 Appendix B: Error Codes and Handling

* **ConnectionError:** Network connectivity issues, triggers automatic reconnection
* **ValueError:** Invalid configuration or unsupported destination/source
* **queue.Full:** Internal queue at max capacity, backpressure signal
* **queue.Empty:** No messages available in non-blocking get\_data()

## 10.3 Appendix C: Best Practices

* **Always call start() on subscribers:** Subscribers don't auto-start; explicit start() required
* **Graceful shutdown:** Call stop() on publishers and subscribers before process exit
* **Monitor queue depths:** Use details() and get\_queue\_size() to detect bottlenecks
* **Use batching for high throughput:** Configure publish\_interval and batch\_size for efficiency
* **Tune max\_depth:** Balance memory usage against buffering capacity
* **Use DataAwarePayload sparingly:** Performance overhead; prefer static destinations when possible

## 10.4 Appendix D: Glossary

* **Publisher:** Component that sends messages to a destination
* **Subscriber:** Component that receives messages from a source
* **Destination:** Target endpoint where messages are published
* **Source:** Origin endpoint from which messages are consumed
* **Queue:** Point-to-point messaging pattern (one producer, one consumer)
* **Topic:** Publish-subscribe pattern (one producer, multiple consumers)
* **Batching:** Accumulating multiple messages before transmission
* **Backpressure:** Flow control mechanism when consumer can't keep up with producer
* **CDE:** Custom Data Elements - metadata accompanying message payload

## 10.5 Appendix E: Dependencies

* **Core Python:** threading, queue, json, logging, abc
* **Kafka:** kafka-python
* **ActiveMQ:** stomp.py
* **RabbitMQ:** pika
* **Redis:** redis
* **SQL:** pymysql (MySQL), psycopg2 (PostgreSQL)
* **REST:** requests
* **gRPC:** grpcio, protobuf
* **Aerospike:** aerospike

***--- End of Document ---***