Dishtha Yantra In-Memory Cache Framework

**Requirements and Architecture Document**

*Version 1.0*

*October 2025*

|  |  |
| --- | --- |
| **Document Title** | Dishtha Yantra In-Memory Cache Framework Requirements |
| **Version** | 1.0 |
| **Date** | October 2025 |
| **Status** | Implemented |

# 1. Executive Summary

The Dishtha Yantra In-Memory Cache Framework is a comprehensive, thread-safe, high-performance caching solution designed as a pure Python implementation of Redis functionality. It provides a complete in-memory data store with persistence capabilities, pub/sub messaging, and seamless integration with data pipeline architectures.

The framework is designed for environments where network connectivity is limited or where a lightweight, embedded cache solution is preferred over external dependencies. It offers complete Redis API compatibility while adding enterprise features like automatic persistence, web-based management UI, and integration with the DataPubSub pattern for event-driven architectures.

# 2. System Overview

## 2.1 Core Philosophy

Dishtha Yantra (Sanskrit: 'direction instrument') embodies the principles of providing clear direction and instrumentation for data management. The framework is built on three foundational pillars:

* **Zero External Dependencies:** Pure Python implementation requiring no external cache servers or network configuration
* **Enterprise Persistence:** Automatic periodic dumps with atomic write operations ensuring data durability
* **Event-Driven Architecture:** Native integration with pub/sub patterns for real-time data distribution

## 2.2 Architecture Components

The framework consists of three major components working in harmony:

| **Component** | **Purpose** | **Key Features** |
| --- | --- | --- |
| **InMemoryRedisClone** | Core cache engine implementing Redis data structures and operations | Thread-safe operations, TTL management, persistence, pub/sub |
| **DataPubSub Framework** | Messaging abstraction layer for publishers and subscribers | Batch publishing, queue management, suspend/resume controls |
| **Web Management UI** | Flask-based administrative interface for cache operations | Real-time monitoring, CRUD operations, dump management |

# 3. Core Cache Engine (InMemoryRedisClone)

## 3.1 Design Principles

The InMemoryRedisClone is implemented as a Singleton class using the SingletonMeta metaclass, ensuring a single shared instance across the application. It provides complete Redis API compatibility while maintaining thread safety through RLock-based synchronization.

## 3.2 Supported Data Structures

| **Type** | **Implementation** | **Key Operations** |
| --- | --- | --- |
| **String** | Python str type | GET, SET, MGET, MSET, INCR, DECR, APPEND |
| **List** | Python collections.deque | LPUSH, RPUSH, LPOP, RPOP, LRANGE, LLEN, LINDEX |
| **Set** | Python set type | SADD, SREM, SMEMBERS, SINTER, SUNION, SDIFF |
| **Sorted Set** | List of (member, score) tuples with heapq | ZADD, ZREM, ZRANGE, ZRANK, ZSCORE, ZINCRBY |
| **Hash** | Python dict type | HSET, HGET, HDEL, HGETALL, HINCRBY, HKEYS |

## 3.3 Key Management Features

### 3.3.1 Time-To-Live (TTL) Management

The framework provides comprehensive TTL management with automatic expiration:

* **Expire:** Set expiration time in seconds
* **ExpireAt:** Set expiration at specific Unix timestamp
* **TTL:** Query remaining time to live
* **Persist:** Remove expiration from a key
* **Background Cleanup:** Daemon thread checking every second for expired keys

### 3.3.2 Pattern Matching and Key Discovery

Supports Redis-style pattern matching using glob patterns:

* \* matches zero or more characters
* ? matches exactly one character
* Character classes [abc] and ranges [a-z] supported

## 3.4 Persistence Architecture

The persistence layer ensures data durability through periodic JSON dumps with atomic write operations:

### 3.4.1 Periodic Dump Mechanism

1. **Background Daemon Thread:** Runs continuously checking for dump interval
2. **Configurable Interval:** Default 300 seconds (5 minutes), customizable per deployment
3. **Atomic Writes:** Write to temporary file, then atomic rename
4. **Metadata Tracking:** Includes dump timestamp, version, total keys

### 3.4.2 Dump File Format

The dump file is structured JSON containing:

* **Metadata Section:** Dump timestamp, key count, format version
* **Data Section:** Key-value pairs with type and TTL information
* **Type Preservation:** Complex types serialized appropriately (deque→list, set→list, etc.)

### 3.4.3 Auto-Reload on Startup

On initialization, the cache automatically checks for existing dump files and reloads data, including:

* Reconstructing all data structures
* Restoring type information
* Recalculating expiration times based on remaining TTL

## 3.5 Pub/Sub Implementation

Built-in publish/subscribe messaging system supporting both channel-based and pattern-based subscriptions:

### 3.5.1 Channel Subscriptions

* **Subscribe/Unsubscribe:** Register callback functions for specific channels
* **Publish:** Send messages to all subscribers of a channel
* **Message Delivery:** Callback-based with thread-safe execution

### 3.5.2 Pattern Subscriptions

* Subscribe to multiple channels using glob patterns (e.g., news.\*)
* Pattern matching executed on publish for efficient routing
* Pattern unsubscribe support

## 3.6 Transaction Support

Implements Redis-style transactions with optimistic locking:

* **MULTI/EXEC Pattern:** Queue multiple commands for atomic execution
* **WATCH/UNWATCH:** Optimistic locking with version checking
* **Pipeline Support:** Context manager for batching operations
* **DISCARD:** Abort transaction and clear queue

# 4. DataPubSub Framework Integration

## 4.1 Framework Overview

The DataPubSub framework provides an abstraction layer for message producers and consumers, enabling loose coupling between components in data pipeline architectures. It defines abstract base classes for publishers and subscribers with lifecycle management, statistics tracking, and configurable behavior.

## 4.2 DataPublisher Abstract Base Class

Core features of the publisher abstraction:

* **Batch Publishing:** Configurable batch\_size for grouping messages
* **Periodic Flushing:** Time-based flush intervals (publish\_interval parameter)
* **Queue Management:** Internal thread-safe queue for buffering
* **Statistics:** Publish count, last publish time, queue depth tracking

## 4.3 DataSubscriber Abstract Base Class

Core features of the subscriber abstraction:

* **Background Thread:** Daemon thread for continuous subscription
* **Internal Queue:** Configurable max\_depth with overflow protection
* **Suspend/Resume:** Flow control through threading.Event
* **Blocking/Non-blocking Reads:** get\_data() with configurable timeout
* **Statistics:** Receive count, last receive time, queue depth

## 4.4 InMemoryRedis Publishers

### 4.4.1 InMemoryRedisDataPublisher

Key-based publisher for setting cache entries:

* **Protocol:** inmemoryredis://
* **Key Extraction:** Uses \_\_dagserver\_key from payload
* **Key Prefix:** Optional namespace prefix (key\_prefix config)
* **TTL Support:** Per-message \_\_ttl\_seconds or default ttl\_seconds
* **JSON Serialization:** Automatic JSON encoding of payload

### 4.4.2 InMemoryRedisChannelDataPublisher

Channel-based publisher using pub/sub:

* **Protocol:** inmemoryredischannel://channel\_name
* **Channel Extraction:** Parsed from destination URL or config
* **Broadcast:** Messages delivered to all channel subscribers
* **Subscriber Count:** Returns number of active subscribers

## 4.5 InMemoryRedis Subscribers

### 4.5.1 InMemoryRedisDataSubscriber

Polling-based subscriber for cache keys:

* **Key Pattern:** Configurable glob pattern (key\_pattern config)
* **Seen Keys Tracking:** Avoids duplicate reads
* **Delete on Read:** Optional delete\_on\_read for queue-like behavior
* **Poll Interval:** Configurable sleep between checks
* **JSON Deserialization:** Automatic with fallback to raw string

### 4.5.2 InMemoryRedisChannelDataSubscriber

Event-driven subscriber for pub/sub channels:

* **Channel Subscription:** Registers callback with Redis pub/sub
* **Message Queue:** Thread-safe queue for received messages
* **Non-blocking Reads:** get\_data() returns immediately
* **JSON Deserialization:** Automatic with error handling

## 4.6 DataAwarePayload

A structured message container used throughout the framework:

* **Destination:** Target queue or topic
* **CDE (Context Data Elements):** Metadata dictionary
* **Payload:** Actual message data
* **Methods:** add\_to\_cde(), to\_dict(), get\_data\_for\_publication()

# 5. Web Management UI

## 5.1 Technology Stack

* **Backend:** Flask web framework
* **Frontend:** Bootstrap 5, vanilla JavaScript
* **Authentication:** Session-based with role-based access control
* **API:** RESTful JSON endpoints

## 5.2 Dashboard Features

### 5.2.1 Real-Time Statistics

The dashboard displays live cache metrics:

* **Total Keys:** Current number of keys in cache
* **Keys with TTL:** Count of expiring keys
* **Keys without TTL:** Count of persistent keys
* **Last Dump Time:** Timestamp of most recent persistence
* **Dump Interval:** Configured persistence frequency
* **Type Distribution:** Breakdown by data structure type

### 5.2.2 Search and Query Interface

* **Pattern Matching:** Glob-style key search (\*, ?, [])
* **Pagination:** Configurable rows per page (10, 50, 100, all)
* **Value Preview:** Truncated values with click-to-expand
* **Type Indication:** Visual badges for data types
* **TTL Display:** Human-readable expiration times

## 5.3 Administrative Operations

Role-based access control restricts these operations to admin users:

### 5.3.1 CRUD Operations

1. **Create Entry:** Add new key-value pairs with optional TTL
2. **View Full Value:** Modal display with copy-to-clipboard
3. **Update TTL:** Modify expiration time or remove TTL
4. **Delete Entry:** Remove specific keys with confirmation

### 5.3.2 Bulk Operations

* **Manual Dump:** Trigger immediate persistence to disk
* **Download Cache:** Export entire cache as JSON file
* **Clear Cache:** FLUSHALL operation with double confirmation

## 5.4 API Endpoints

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| **/cache/api/stats** | GET | Retrieve cache statistics (all users) |
| **/cache/api/query** | GET | Search keys by pattern with pagination (all users) |
| **/cache/api/create** | POST | Create new cache entry (admin only) |
| **/cache/api/delete** | DELETE | Delete cache entry (admin only) |
| **/cache/api/ttl** | PUT | Update TTL for entry (admin only) |
| **/cache/api/clear** | POST | Clear entire cache (admin only) |
| **/cache/api/download** | GET | Download cache as JSON file (admin only) |
| **/cache/api/dump/trigger** | POST | Trigger manual cache dump (admin only) |

# 6. Technical Requirements

## 6.1 System Requirements

* **Python Version:** 3.8 or higher
* **Memory:** Depends on cache size, minimum 512MB RAM
* **Disk Space:** Sufficient for periodic dumps
* **CPU:** Multi-core recommended for concurrent operations

## 6.2 Python Dependencies

* **Standard Library:** threading, queue, time, json, collections, heapq, datetime, logging
* **Flask:** Web framework for UI (>=2.0)
* **defusedxml:** Secure XML parsing (if needed)

## 6.3 Performance Characteristics

* **String Operations:** O(1) for GET/SET
* **List Operations:** O(1) for LPUSH/RPUSH/LPOP/RPOP, O(n) for LRANGE
* **Set Operations:** O(1) for SADD/SREM, O(n) for set operations
* **Sorted Set Operations:** O(log n) for ZADD/ZREM, O(k) for ZRANGE
* **Hash Operations:** O(1) for HGET/HSET
* **Pattern Matching:** O(n) where n is total keys

## 6.4 Thread Safety

All operations are protected by RLock (reentrant lock), enabling:

* Concurrent read/write operations
* Safe multi-threaded access
* Atomic operations within transactions
* Separate locks for pub/sub to minimize contention

## 6.5 Scalability Considerations

* **Memory Constraints:** Limited by available RAM
* **Key Count:** Efficiently handles millions of keys
* **Concurrent Clients:** Limited by GIL (Global Interpreter Lock)
* **Pub/Sub Subscribers:** Unlimited subscribers per channel

# 7. Use Cases and Applications

## 7.1 Session Management

Store user session data with automatic expiration:

* Session tokens with TTL
* User preferences and state
* Shopping cart data
* Authentication context

## 7.2 Data Pipeline Coordination

Coordinate between DAG nodes and pipeline stages:

* Intermediate computation results
* Pipeline state management
* Job queue implementation
* Progress tracking

## 7.3 Real-Time Messaging

Pub/sub patterns for event-driven architectures:

* Notification systems
* Event broadcasting
* Status updates
* Live dashboards

## 7.4 Leaderboards and Rankings

Using sorted sets for competitive features:

* Gaming leaderboards
* Sales rankings
* Performance metrics
* Time-series data

## 7.5 Distributed Locking

Implement distributed locks using SET with NX and EX:

* Resource exclusion
* Critical section protection
* Scheduled job coordination

# 8. Integration Architecture

## 8.1 System Integration Diagram

The Dishtha Yantra framework integrates seamlessly with the DAG Compute Server and pub/sub messaging infrastructure:

### 8.1.1 Core Components Integration

1. **DAG Compute Server:** Uses InMemoryRedisClone as shared state store for DAG nodes
2. **Flask Web Application:** Initializes singleton Redis instance and provides management UI
3. **DataPubSub Publishers:** Write computation results to cache with configurable TTL
4. **DataPubSub Subscribers:** Poll cache for new data or subscribe to channels

### 8.1.2 Data Flow Patterns

**Pattern 1: Key-Based Message Queue**

* Producer uses InMemoryRedisDataPublisher to SET keys
* Consumer uses InMemoryRedisDataSubscriber to poll with pattern
* Optional delete\_on\_read for queue behavior

**Pattern 2: Channel-Based Pub/Sub**

* Publisher broadcasts to inmemoryredischannel://channel\_name
* Multiple subscribers receive same message
* Real-time event distribution

**Pattern 3: Shared State Store**

* DAG nodes write intermediate results
* Downstream nodes read when dependencies are met
* Web UI provides visibility into computation state

## 8.2 Configuration Management

The framework supports extensive configuration:

### 8.2.1 Cache Configuration

* **dump\_file:** Path to persistence file (default: data/inmemory\_redisclone.json)
* **dump\_interval:** Seconds between dumps (default: 300)

### 8.2.2 Publisher Configuration

* **redis\_instance:** Shared InMemoryRedisClone instance
* **key\_prefix:** Namespace for keys
* **ttl\_seconds:** Default expiration time
* **publish\_interval:** Batch flush interval
* **batch\_size:** Messages per batch

### 8.2.3 Subscriber Configuration

* **redis\_instance:** Shared InMemoryRedisClone instance
* **key\_pattern:** Glob pattern for polling
* **delete\_on\_read:** Queue-like behavior
* **poll\_interval:** Seconds between polls
* **max\_depth:** Internal queue size

# 9. Best Practices

## 9.1 Key Naming Conventions

* **Use Hierarchical Namespaces:** user:1000:profile, session:abc123:data
* **Avoid Special Characters:** Stick to alphanumeric, colon, underscore
* **Include Type in Name:** Makes debugging easier (user:1000:hash, queue:tasks:list)
* **Keep Names Short:** Reduces memory overhead

## 9.2 TTL Management

* **Always Set TTL for Temporary Data:** Prevents memory leaks
* **Use Appropriate Expiration Times:** Sessions: 30 minutes, Cache: 1 hour, Locks: seconds
* **Monitor TTL:** Check statistics for keys without TTL
* **Refresh TTL on Access:** For sliding expiration windows

## 9.3 Persistence Strategy

* **Choose Appropriate Dump Interval:** Balance between durability and performance
* **Monitor Dump Success:** Check logs and last dump timestamp
* **Backup Dump Files:** External backup strategy recommended
* **Test Restore Process:** Verify auto-reload works correctly

## 9.4 Pub/Sub Patterns

* **Use Channels for Broadcast:** When multiple consumers need same data
* **Use Keys for Queues:** When single consumer should process each message
* **Keep Messages Small:** Store large data separately, send references
* **Handle Subscriber Failures:** Implement retry logic and error handling

## 9.5 Performance Optimization

* **Use Pipelines for Bulk Operations:** Reduce lock contention
* **Avoid Large Value Sizes:** Keep individual values under 1MB
* **Use Appropriate Data Structures:** Hashes for objects, sets for unique collections
* **Monitor Memory Usage:** Track total keys and cache size
* **Limit Pattern Searches:** KEYS command scans all keys, use sparingly

# 10. Conclusion

The Dishtha Yantra In-Memory Cache Framework represents a comprehensive solution for embedded caching needs in Python applications. By providing Redis-compatible operations without external dependencies, it enables developers to build robust, high-performance applications with simplified deployment requirements.

The framework's integration with the DataPubSub pattern creates a powerful event-driven architecture, while the web-based management UI provides essential operational visibility. Together with automatic persistence and comprehensive data structure support, Dishtha Yantra offers enterprise-grade caching capabilities in a pure Python implementation.

Key differentiators include:

* **Zero External Dependencies:** No Redis server, no network configuration
* **Full Redis API Compatibility:** Drop-in replacement for common operations
* **Enterprise Persistence:** Automatic, atomic dumps with reload
* **Event-Driven Integration:** Native pub/sub with DataPubSub framework
* **Operational Visibility:** Web UI for monitoring and management

The framework is production-ready and actively used in DAG-based compute pipelines, providing the caching and messaging infrastructure needed for complex data workflows. Its thread-safe design and comprehensive feature set make it suitable for a wide range of applications, from session management to real-time data distribution.

# Appendix A: API Reference Summary

Complete listing of supported Redis commands:

| **Category** | **Commands** | **Count** |
| --- | --- | --- |
| **Strings** | GET, SET, MGET, MSET, INCR, DECR, APPEND, STRLEN, GETRANGE, SETRANGE | 12 |
| **Lists** | LPUSH, RPUSH, LPOP, RPOP, LRANGE, LLEN, LINDEX, LSET, LREM, LTRIM | 11 |
| **Sets** | SADD, SREM, SMEMBERS, SCARD, SISMEMBER, SINTER, SUNION, SDIFF, SPOP, SRANDMEMBER | 12 |
| **Sorted Sets** | ZADD, ZREM, ZRANGE, ZREVRANGE, ZRANK, ZSCORE, ZCARD, ZINCRBY, ZCOUNT, ZREMRANGEBYSCORE | 14 |
| **Hashes** | HSET, HGET, HDEL, HEXISTS, HGETALL, HKEYS, HVALS, HLEN, HINCRBY, HMSET, HMGET | 12 |
| **Keys** | KEYS, EXISTS, DELETE, EXPIRE, EXPIREAT, TTL, PERSIST, RENAME, TYPE, DBSIZE, FLUSHALL | 11 |
| **Pub/Sub** | SUBSCRIBE, UNSUBSCRIBE, PSUBSCRIBE, PUNSUBSCRIBE, PUBLISH | 5 |
| **Transactions** | WATCH, UNWATCH, MULTI, EXEC, DISCARD | 5 |
| **Total** | **All Categories** | **80+** |

# Appendix B: Configuration Examples

## B.1 Basic Initialization

from core.pubsub.inmemory\_redisclone import InMemoryRedisClone
# Initialize with default settings
redis = InMemoryRedisClone()
# Initialize with custom dump configuration
redis = InMemoryRedisClone(
dump\_file='data/custom\_cache.json',
dump\_interval=600 # 10 minutes
)

## B.2 Publisher Configuration

from inmemoryredis\_datapubsub import InMemoryRedisDataPublisher
# Key-based publisher
publisher = InMemoryRedisDataPublisher(
name='my\_publisher',
destination='inmemoryredis://',
config={
'redis\_instance': redis,
'key\_prefix': 'app:',
'ttl\_seconds': 3600,
'publish\_interval': 5,
'batch\_size': 100
}
)

## B.3 Subscriber Configuration

from inmemoryredis\_datapubsub import InMemoryRedisChannelDataSubscriber
# Channel subscriber
subscriber = InMemoryRedisChannelDataSubscriber(
name='my\_subscriber',
source='inmemoryredischannel://notifications',
config={
'redis\_instance': redis,
'max\_depth': 10000
}
)
subscriber.start()

*--- End of Document ---*