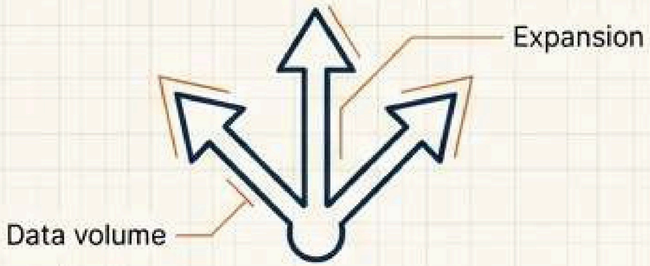
PyKV: A Scalable In-Memory In-Memory Key-Value Store with Persistence

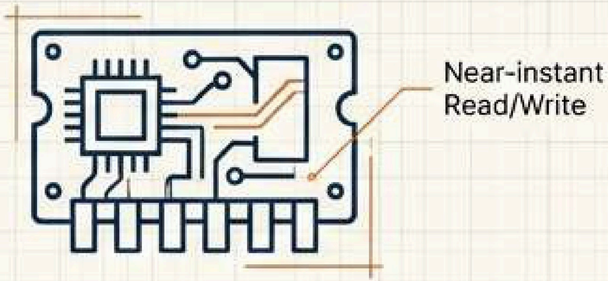
An Architectural Blueprint

Deconstructing the Title: The Four Pillars of PyKV



Scalable

The system is built to handle increasing data volume and growing numbers of client requests without performance degradation.

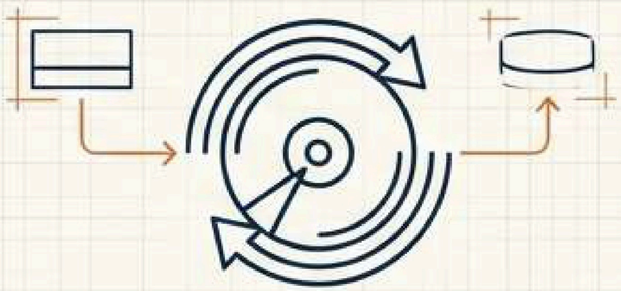


In-Memory

Data is stored directly in RAM, giving near-instant read/write operations for high performance.

Key-Value Store

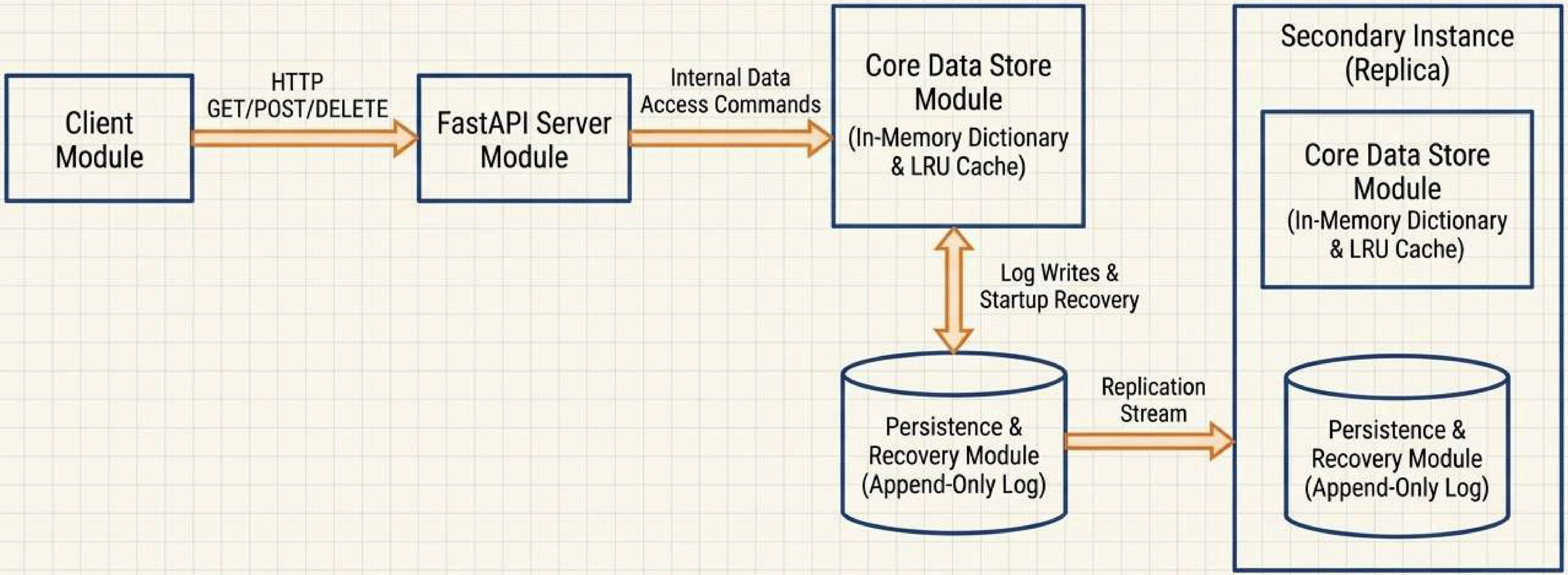
A simple and efficient NoSQL model where data is accessed using a key, enabling O(1) average lookup time.



With Persistence

Although data is in memory for speed, every change is written to disk so the system can recover its state after a crash or restart.

The PyKV Architectural Blueprint



An Intuitive Analogy: PyKV as a Modern Library System

To understand how PyKV works, we can visualize it as a modern library where patrons interact with a librarian and shelves.

The Cast (Key Mapping)

Patron = Client Module

Librarian = FastAPI Server

Shelves = Core Data Store

Logbook = Persistence Module

Branch Library = Replication Module

The Actions (Operation Flows)

GET (Borrow a Book)

Patron asks the Librarian for a book. The Librarian checks the Shelves and **returns it.**

SET/PUT (Add a New Book)

Patron gives a new book to the Librarian. It's placed on the Shelves, **and the action is recorded in the Logbook.**

DELETE (Remove a Book)

Patron requests removal. The Librarian takes the book off the Shelves and notes the removal in the **Logbook.**

Module Deep Dive: The Core Data Store

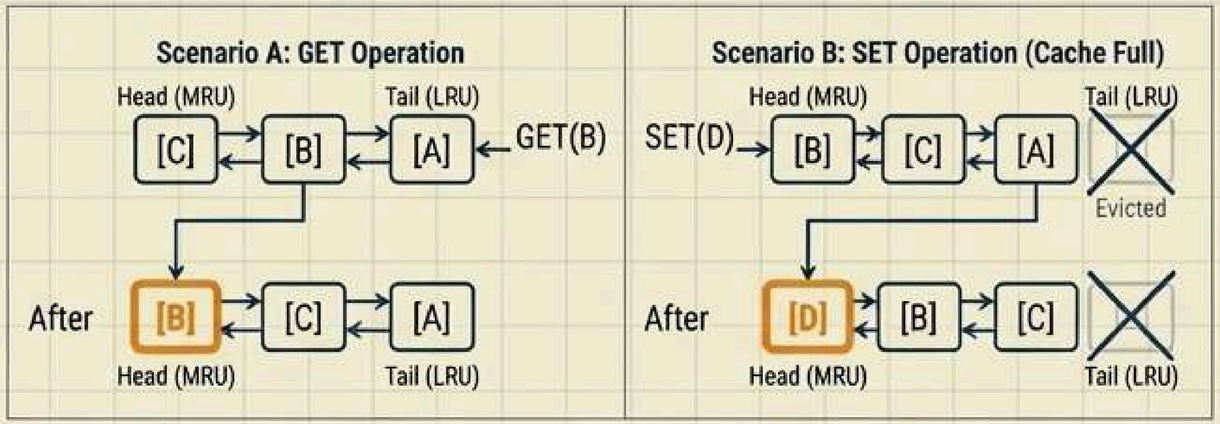
Core Purpose: To hold all key-value pairs in memory for O(1) access and enforce a strict memory usage policy via an LRU (Least Recently Used) eviction policy.

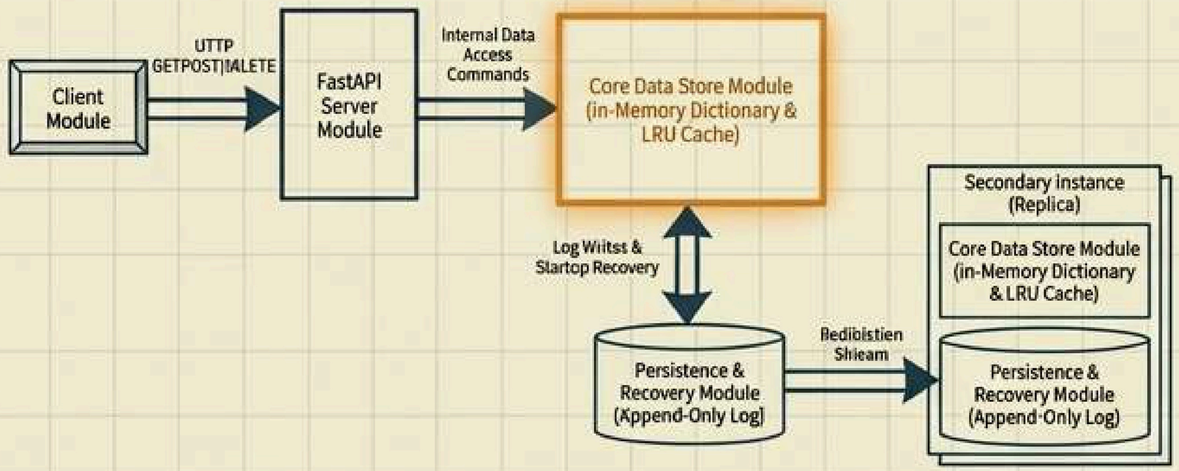
Data Structure — The Chosen Approach:

- A combination of a Python Dictionary and a Doubly Linked List was chosen for true O(1) performance and precise control over eviction.
- Dictionary: Maps keys to nodes for fast lookup.
- Doubly Linked List: Tracks usage order. Head = Most Recently Used (MRU), Tail = Least Recently Used (LRU).

Implementation Plan Highlights:

- Initialize store with MAX_CAPACITY.
- Implement GET, SET/PUT, and DELETE logic that updates both the dictionary and linked list.
- On SET/PUT, if capacity is exceeded, evict the tail node (LRU).
- Integrate with Persistence to trigger asynchronous logging on writes.





Module Deep Dive: The FastAPI Server

Core Purpose

To act as the asynchronous network gateway, handling thousands of concurrent client connections efficiently without blocking.

Key Responsibilities

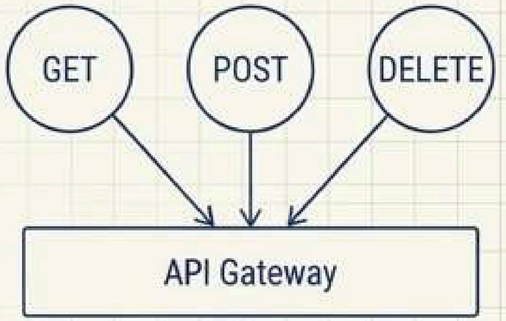
- Handle all incoming HTTP requests (GET, POST, PUT, DELETE).
- Use **Pydantic models** to automatically validate all request payloads and parameters.
- Act as the sole entry point, translating client requests into internal commands for the Core Data Store.

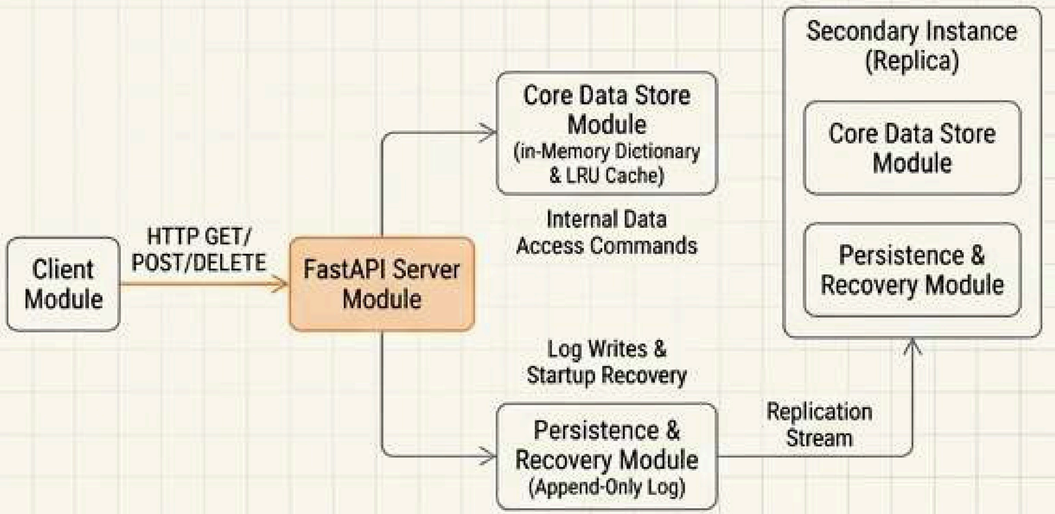
API Endpoints

HTTP Method	Endpoint	Description
POST	/kv/	Add a new key-value pair
GET	/kv/{key}	Retrieve the value for a key
PUT	/kv/{key}	Update an existing key's value
DELETE	/kv/{key}	Delete a key-value pair

Implementation Plan Highlights

- Set up FastAPI application and define all endpoints.
- Define Pydantic models for request body validation.
- Forward validated requests to Core Data Store methods asynchronously.
- For write operations, trigger non-blocking calls to the Persistence module.





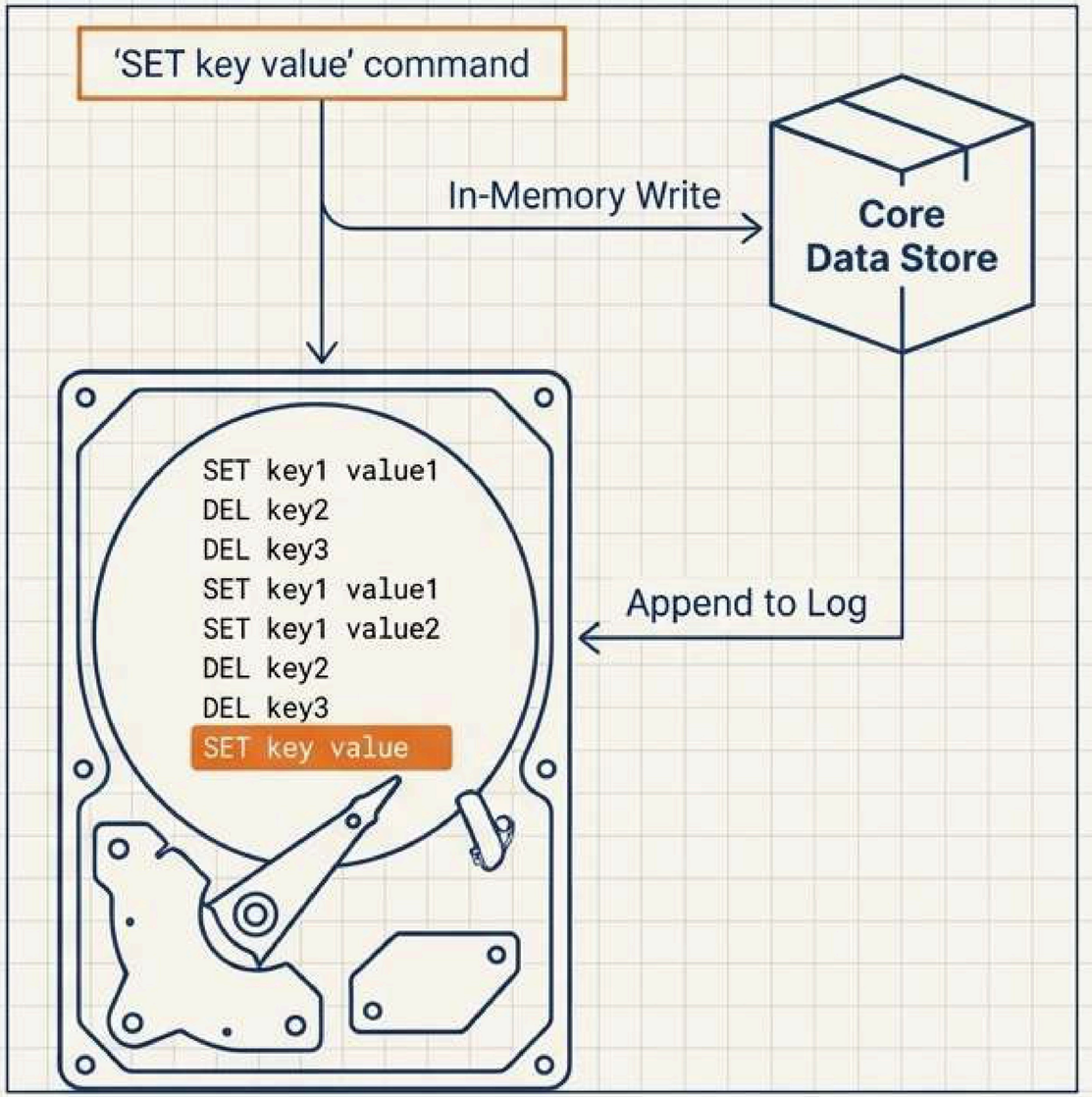
Module Deep Dive: Persistence & Recovery

Core Purpose
To ensure data durability and crash recovery by safely recording all state changes on disk.

Persistence Mechanism: Append-Only Log
- All state-changing operations (SET, DELETE, UPDATE) are written sequentially to a log file.
- This provides fast, efficient disk writes and a complete, ordered history for perfect reconstruction.

Key Processes
- **Crash Recovery Protocol**: On startup, the system replays the log entries in order to reconstruct the in-memory state exactly as it was before shutdown.
- **Log Compaction**: A background process runs to create a new, optimized log file, keeping only the latest state for each key to manage disk space.

Implementation Plan Highlights
- Logic for asynchronously appending operations to the log file as JSON objects or binary records.
- A startup routine that reads the log sequentially to rebuild the Core Data Store.
- A background task for periodic log compaction.



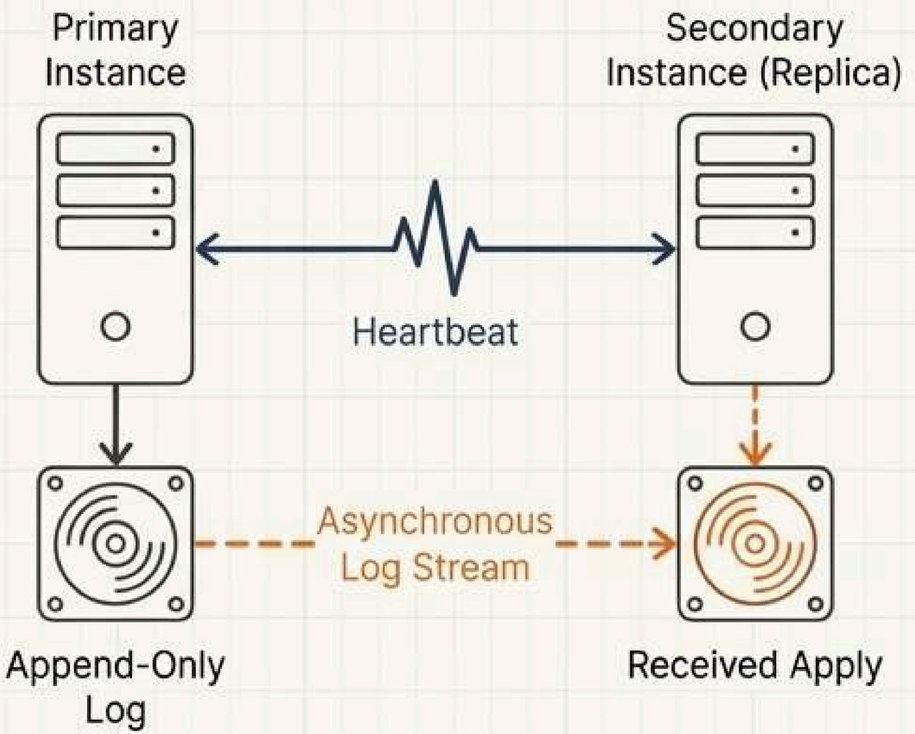
Module Deep Dive: Replication for High Availability

Core Purpose

To provide system resilience and ensure minimal downtime through a primary-secondary replication model.

Replication Logic

- **Asynchronous Streaming:** The primary server streams its append-only log entries to the secondary instance without blocking client operations.
- **Eventual Consistency:** The secondary node maintains a near-real-time copy of the primary's data.
- **Failover:** If the primary fails, the secondary can be promoted to serve requests, ensuring continuous availability.



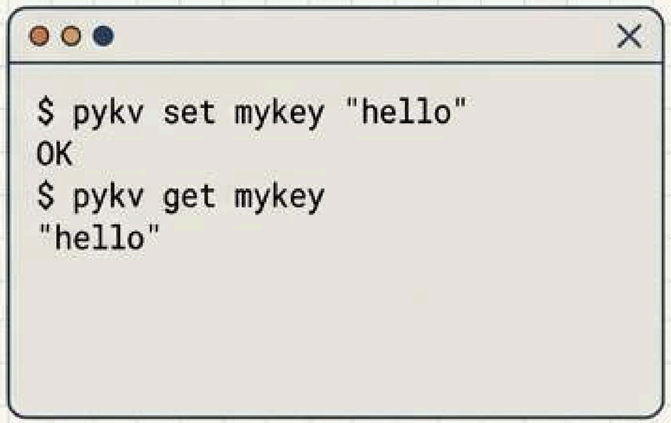
Implementation Plan Highlights

- Stream log entries from the primary's Persistence module to the secondary instance.
- Maintain a network queue or buffer for reliable log transmission.
- The secondary node continuously applies incoming log entries to its own in-memory Core Data Store.

Module Deep Dive: The Client Interface (CLI & GUI)

Core Purpose: To provide powerful and user-friendly tools for interacting with the PyKV server and for performance benchmarking.

Component 1: Command-Line Interface (CLI)



- Supports `SET`, `GET`, and `DEL` operations.
- Sends HTTP requests to the FastAPI server.
- Includes a multi-threaded benchmarking mode to stress-test performance.

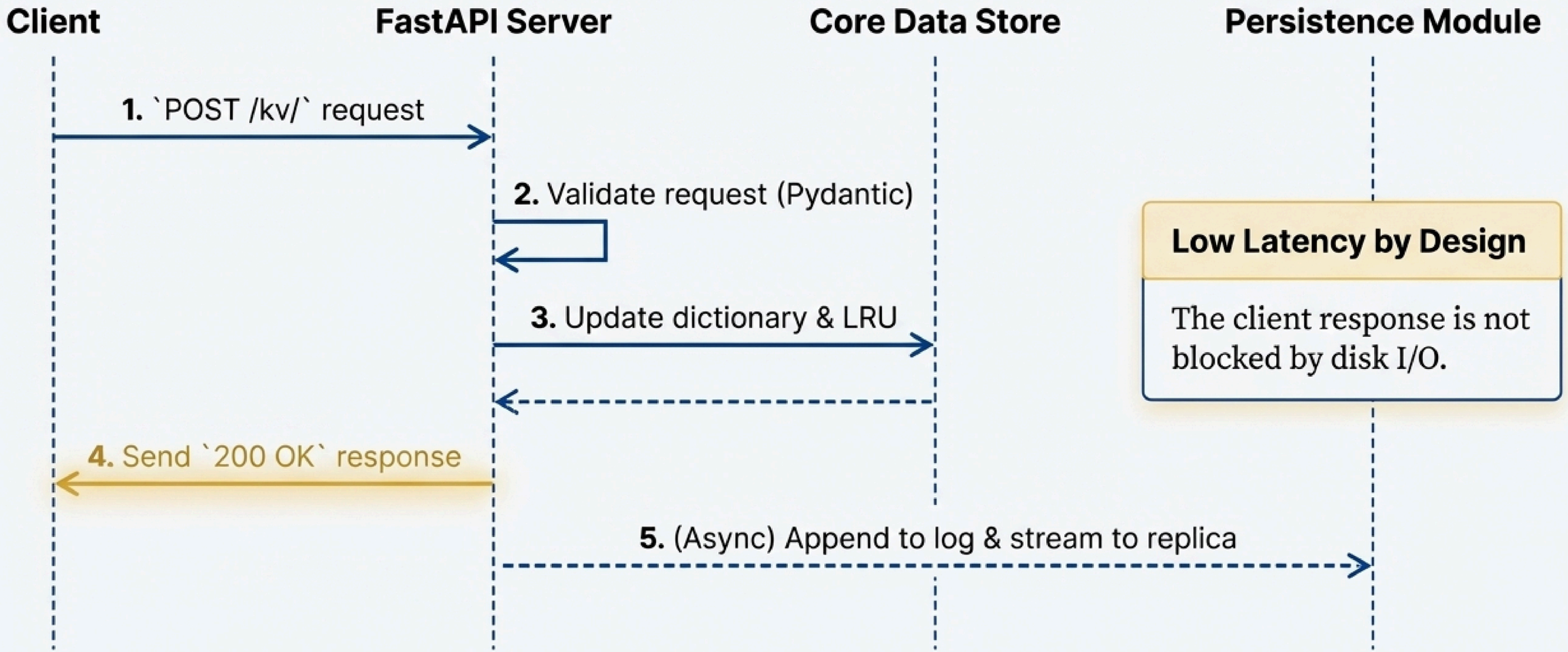
Component 2: Streamlit GUI

- A visual web-based interface for performing key-value operations.
- Features input fields for keys/values and buttons for GET, SET, UPDATE, DELETE.
- An output area shows server responses in real-time.

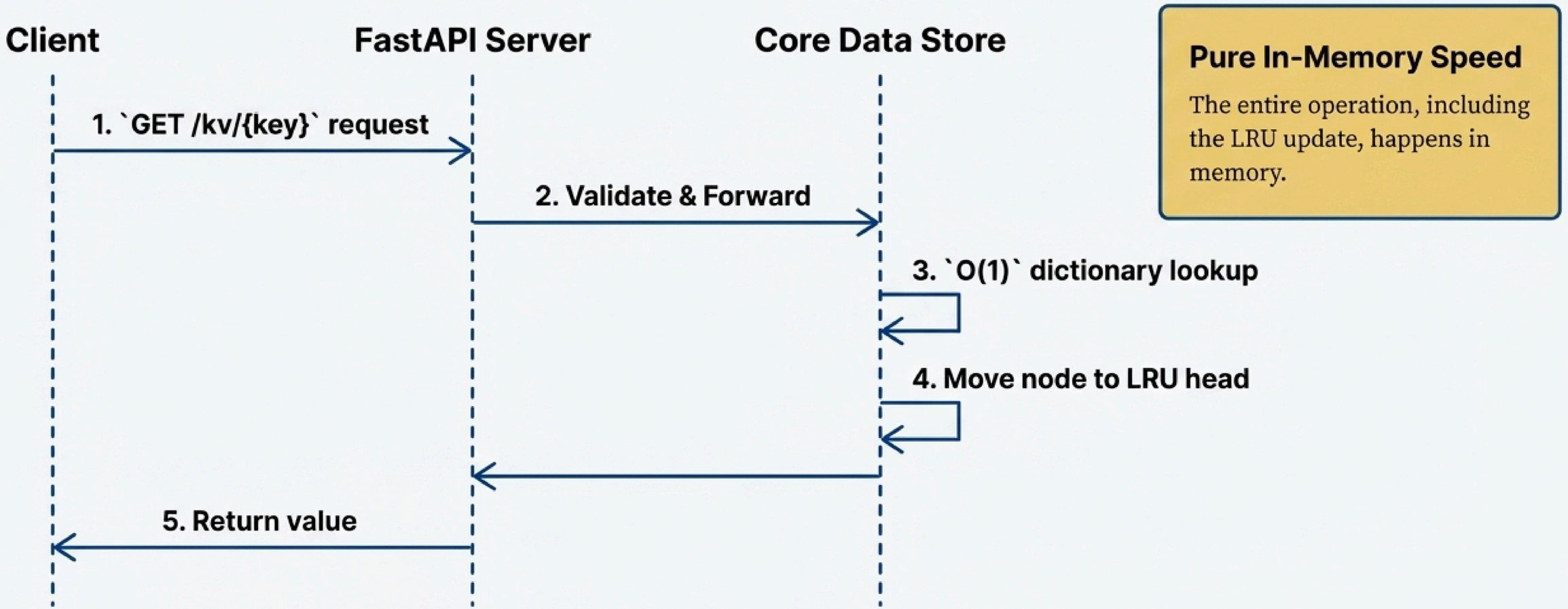
Implementation Plan Highlights

- Build CLI using a library like `argparse` or `click`.
- Create a Streamlit app with components like `st.text_input` and `st.button`.
- Use `requests` or `httpx` library in both clients to communicate with the FastAPI server API.

Anatomy of a Write Operation: Tracing a `SET` Request



Anatomy of a Read Operation: Tracing a `GET` Request

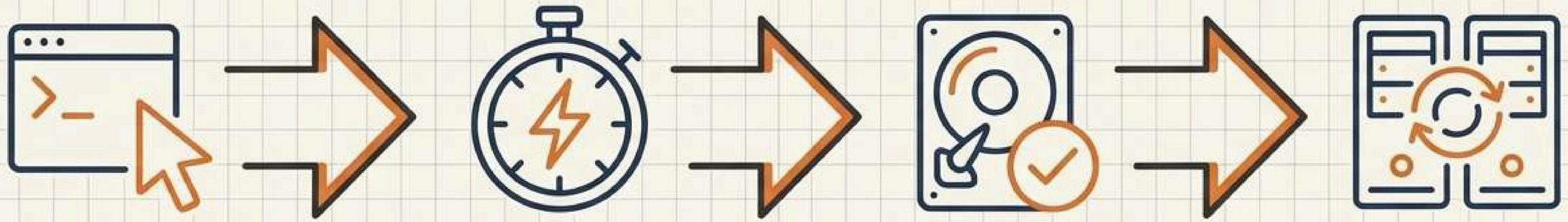


Pure In-Memory Speed

The entire operation, including the LRU update, happens in memory.

The Final User Experience: Fast, Reliable, and Interactive

A Seamless Workflow



1. Interact

Users launch the CLI or Streamlit GUI to perform operations.

2. Instant Feedback

Every SET, GET, or DELETE operation returns an immediate response, thanks to in-memory processing.

3. Durability by Default

All changes are automatically logged. If the server crashes, a restart restores all data perfectly.

4. Always Available

In a replicated setup, a primary failure results in a seamless transition to the secondary node.

Key Takeaways

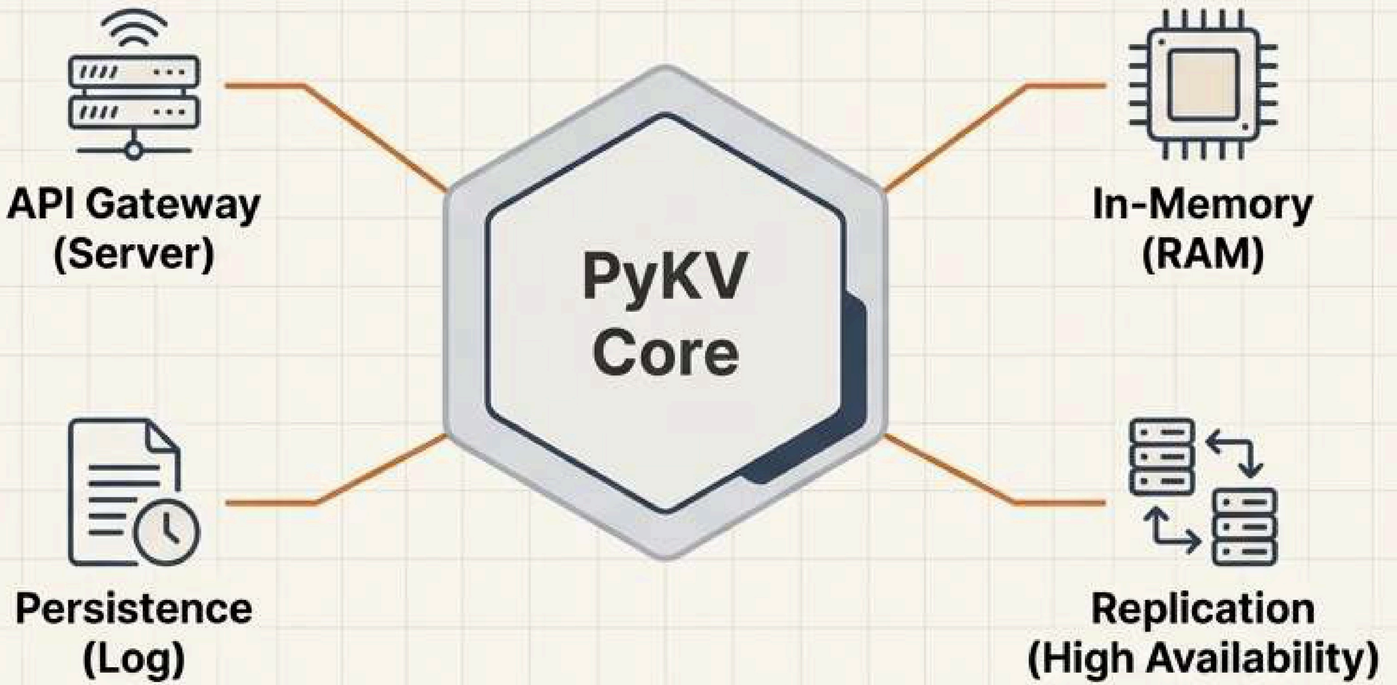
Performance: Handle thousands of concurrent requests with minimal latency.

Reliability: No data is lost on crash, and service remains available during failures.

Accessibility: Interact via a powerful CLI or an intuitive GUI.

Conclusion: A Robust Blueprint for Modern Data Systems

PyKV successfully combines an **asynchronous FastAPI** server, **LRU-based memory management**, **persistent logging**, and **replication** to create a system that is simultaneously fast, scalable, and reliable.



It stands as a robust, user-friendly system engineered to handle the heavy loads and failure scenarios of real-world applications, delivering on the promise of a modern, digital library system where data is always accessible, safe, and instantly available.