Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report

on

"Mini-Redis: A Lightweight In-Memory Key–Value Store in C++"

[Code No: COMP 202]

(For partial fulfillment of Year II / Semsester I in Computer Engineering)

Submitted by

Pratik Karna (037977-24)

Submitted to:

Mr. Sagar Acharya

Department of Computer Science and Engineering

Submission Date: 23rd February, 2026

Bona fide Certificate

This project work on

“Mini-Redis: A Lightweight In-Memory Key–Value Store in C++”

is the bona fide work of

“Pratik Karna”

who carried out the project work under my supervision.

____

Mr. Sagar Acharya

Lecturer

Department of Computer Science and Engineering

Acknowledgement

I would like to express my sincere gratitude to my project supervisor for the valuable guidance, feedback, and encouragement they have provided during the development of the project. I think this has been very helpful in understanding the practical aspects of the project.

I would like to extend my gratitude to the faculty members of the Department of Engineering for providing the learning environment for me to complete this project successfully. The concepts learned during the class have been very useful for the development of this project.

Lastly, I would like to extend my gratitude for the learning opportunities provided during the development of the project, which have significantly enhanced my learning about Data Structures, problem-solving, and system development.

Abstract

The project focuses on developing and implementing a Least Recently Used (LRU) Cache using C++. The stores a fixed number of key-value pairs and removes the least accessed item when the cache is full. It can perform operations such as insertion and retrieval with constant time complexity, i.e., O(1), through a combination of a hash map and a doubly linked list.

The solution will be implemented using a REST-based backend using Crow, which is integrated with a simple web-based frontend. This project can be used to show the practical application of data structures like heaps and linked lists along with caching mechanisms and client-server communication.

Keywords**:** *LRU Cache, CROW, Hash map, REST, C++*

TABLE OF CONTENTS

List of Figures

Acronyms/Abbreviations

LRU	Least Recently Used
GUI	Graphical User Interface
API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
ASIO	Asynchronous Input/Output

Chapter 1 Introduction

Caching is a key concept in computer science, and it is used to speed up the execution of computer programs by storing data in memory for faster retrieval when required. Among the different caching techniques, the LRU caching approach is the most commonly used, and it is the most efficient way to evict the least recently used items when the cache is full.

This project involves implementing an LRU cache using the C++ programming language, and it supports O(1) time complexity for insertion, retrieval, and deletion operations using a combination of a hash map and a doubly linked list data structure. The project involves implementing a simple RESTful API using the Crow library for the backend, allowing users to perform set, get, delete, and view operations on the cache.

Moreover, the project involves developing a simple web interface for users to interact with the LRU cache and view its current status, thus illustrating the concept of caching using web technology. This project not only helps users recall the key concepts of data structures and algorithms but also helps users gain hands-on experience with client-server communication and web development.

1.1 Background

Caches are utilized in computers, where data is stored in a manner closer to the processor or application, resulting in quicker access. There are several cache replacement strategies, including First-In-First-Out (FIFO), Least Frequently Used, and Least Recently Used. In LRU, the data not accessed over the maximum period of time is removed, making it suitable for applications with temporal locality.

LRU Cache can be implemented with a hash map, which provides constant-time complexity for key lookups, along with a doubly linked list, which tracks the usage order. This results in O(1) time complexity for insertion, deletion, or update operations.

Web frameworks, like Crow in C++, can be utilized to develop RESTful APIs, which can be accessed through HTTP requests. These APIs can be utilized in combination with a simple frontend, allowing users to visualize the cache, resulting in better understanding of caching mechanisms.

1.2 Objectives

- To design and develop a cache system based on the Least Recently Used (LRU) eviction system.
- To ensure the best time complexity for cache operations using proper data structures.
- To provide endpoints for communicating with the cache and visualizing operations through HTTP requests.
- To provide a clear and practical example of caching mechanisms and their real-time behaviors.

1.3 Motivation and Significance

The motivation of this project also came from the course COMP 202, where we learned about different data structures and algorithms. The core idea is to bring those concepts to real life and learn more about them.

The motivation behind this project is to gain hands-on experience in implementing a fundamental caching mechanism and to understand how eviction policies impact performance. By combining the backend LRU cache with a frontend interface, the project also demonstrates the practical significance of caching in visualizing data access patterns and system behavior, making abstract concepts tangible.

Chapter 2 Literature Review

Caching is an area that has been extensively researched in the field of computer science, especially in the context of Operating Systems, Databases, and Web Development. Among the different cache replacement strategies, the Least Recently Used (LRU) approach is one of the most frequently used cache replacement strategies because it is simple and tends to provide an optimal solution for the cache replacement problem.

There are a number of existing works and projects that emphasize the significance of the LRU cache replacement strategy:

- Operating Systems: The LRU cache replacement strategy is frequently used for the management of the virtual memory.
- Databases/Web Caching: The LRU cache replacement strategy is used by many databases and web caches for the optimization of query responses.
- Programming Practices: The implementation of the LRU cache replacement strategy using data structures such as doubly linked lists along with hash maps is one of the most frequently used programming practices for the implementation of the cache replacement problem, especially for achieving an average time complexity of O(1).

Recently, there have been a number of open-source projects, such as Redis, implemented in C++ and other languages such as Python, for the implementation of the cache replacement problem, along with APIs for setting, getting, and deleting data, along with statistics monitoring, which have been used as the base for the development of this project.

Chapter 3 Design and Implementation

The project is designed to simulate a lightweight Redis-like key-value store with an LRU cache and a simple frontend to visualize cache operations. It consists of two main components: backend (C++ server along with crow), frontend (HTML/CSS/JavaScript).

3.1 Backend Design

- **Language & Framework:** C++17 with the **Crow** microframework for HTTP server functionality and **ASIO** for asynchronous networking.
- **Core Component:**
    - Datastore Class: Implements the LRU cache with a maximum capacity.
    - Uses a hash map for O(1) lookups and a doubly linked list to maintain access order.
    - Supports set, get, delete, and stats operations.
    - Tracks cache statistics: hits, misses, evictions, and total requests.
- **REST API Endpoints:**
    - POST /set – Insert or update key-value pair with optional TTL.
    - GET /get?key=... – Retrieve value by key.
    - DELETE /delete?key=... – Remove key-value pair.
    - GET /stats – Return cache statistics.
    - GET /keys – Return all current keys in the cache.

- **Concurrency:** The server is **multithreaded**, allowing multiple simultaneous requests to be handled efficiently.

3.2 Frontend Design

- **Technologies:** HTML, CSS, JavaScript.
- **Functionalities:**
    - Display cache statistics and all current keys.
    - Allow users to **set**, **get**, and **delete** key-value pairs via a simple form-based interface.
    - Dynamically update cache view without refreshing the page using **Fetch API** for AJAX requests.
- **User Interaction:**
    - Form inputs for key, value, and optional TTL.
    - Buttons to trigger set, get, and delete operations.
    - Lists to display all keys and messages for operation results.

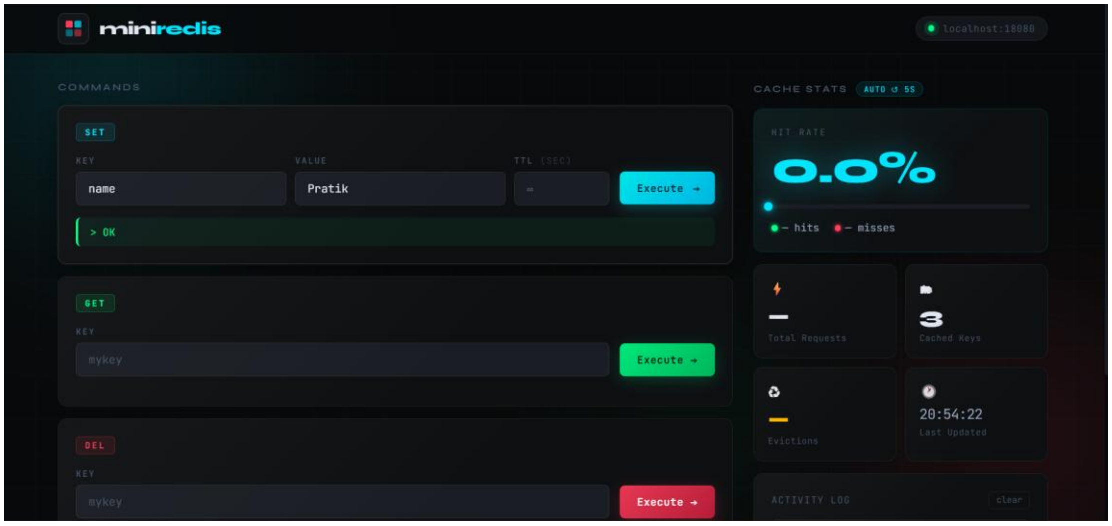
3.3 Implementation Details

- **LRU Cache Logic:**
    - On set, the key is moved to the front of the linked list; if the cache exceeds capacity, the least recently used key is evicted.
    - On get, the accessed key is also moved to the front to indicate recent use.
- **Crow Integration:**
    - Crow routes handle HTTP requests and interact with the Datastore class.

- JSON responses are used for stats and keys endpoints.

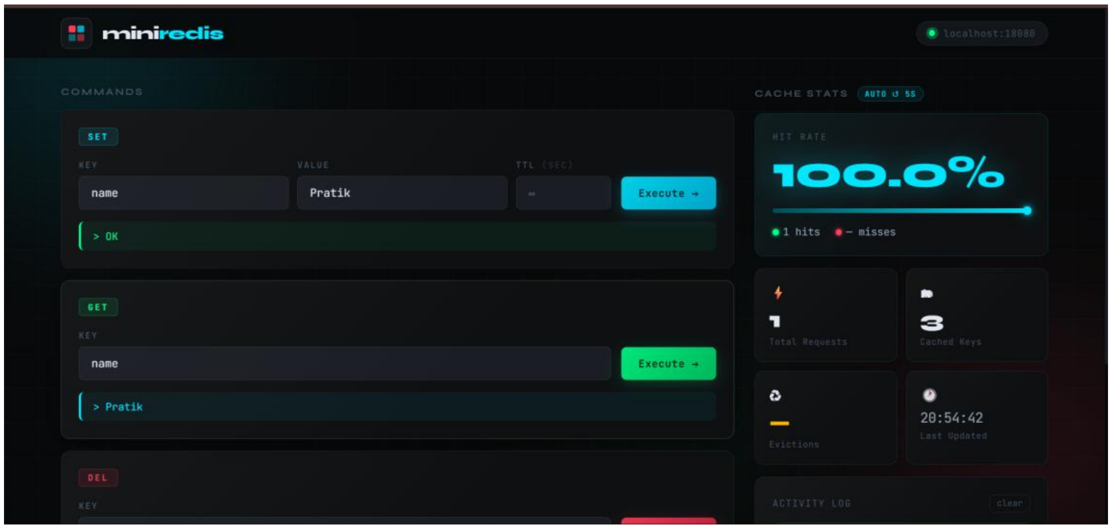
- **Frontend Integration:**
  - Frontend fetches API data and updates the DOM dynamically.
  - Handles errors gracefully and provides real-time visualization of cache operations.

3.4 GUI Overview



KEY: name
VALUE: Pratik
TTL (SEC): ∞
> OK
KEY: mykey
KEY: mykey
Cached Keys: 3
Last Updated: 20:54:22

Fig 3.1: Landing Dashboard


miniredis
localhost:18080

COMMANDS

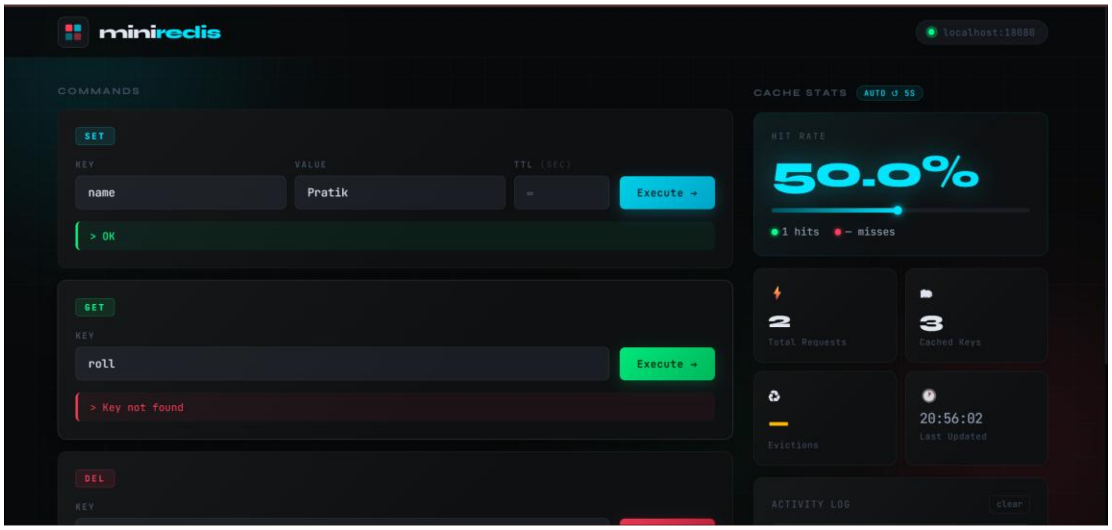
SET
KEY: name
VALUE: Pratik
TTL (SEC): ∞
Execute →
> OK

GET
KEY: name
Execute →
> Pratik

DEL
KEY

CACHE STATS AUTO ↺ 5S
HIT RATE
100.0%
● 1 hits ● – misses
1 Total Requests
3 Cached Keys
– Evictions
20:54:42 Last Updated
ACTIVITY LOG clear

Fig 3.2: Insert Operation


miniredis
localhost:18080

COMMANDS

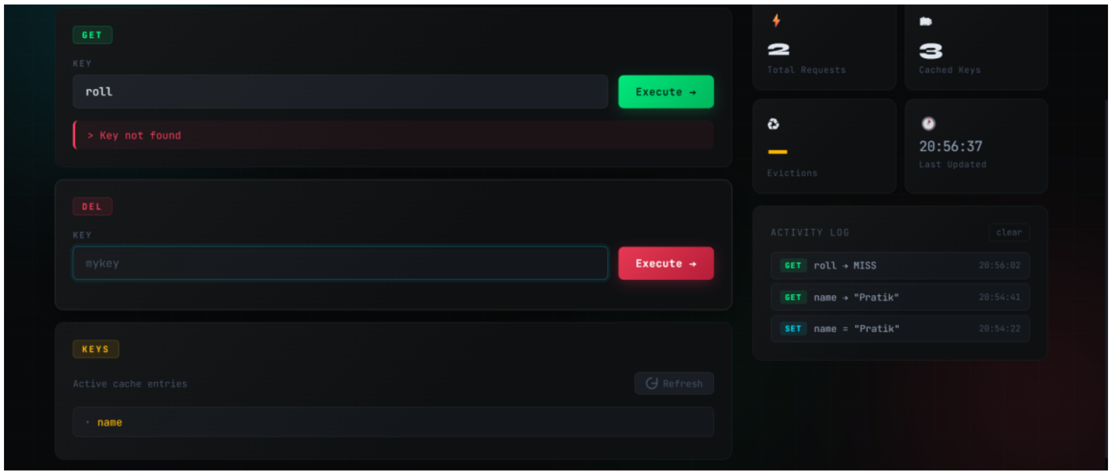
SET
KEY: name
VALUE: Pratik
TTL (SEC): ∞
Execute →
> OK

GET
KEY: roll
Execute →
> Key not found

DEL
KEY

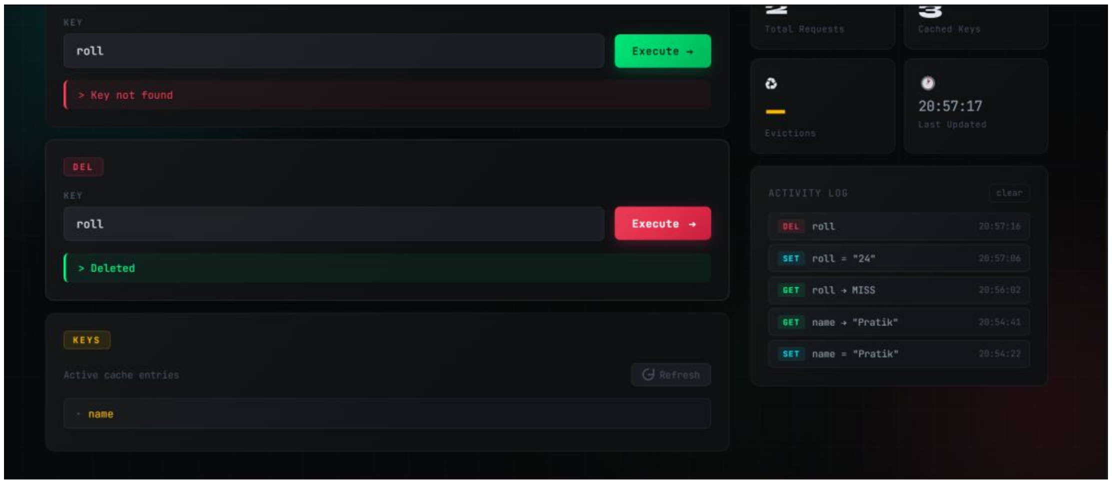
CACHE STATS AUTO ↺ 5S
HIT RATE
50.0%
● 1 hits ● – misses
2 Total Requests
3 Cached Keys
– Evictions
20:56:02 Last Updated
ACTIVITY LOG clear

Fig 3.3: GET Operation



```
roll
> Key not found
mykey
· name
20:56:37
GET roll → MISS          20:56:02
GET name → "Pratik"      20:54:41
SET name = "Pratik"      20:54:22
```

Fig 3.4: Total Keys Display



```
roll
> Key not found
roll
> Deleted
· name
20:57:17
DEL roll                 20:57:16
SET roll = "24"          20:57:06
GET roll → MISS          20:56:02
GET name → "Pratik"      20:54:41
SET name = "Pratik"      20:54:22
```

Fig 3.5: Deletion Operation

3.4 System Diagrams

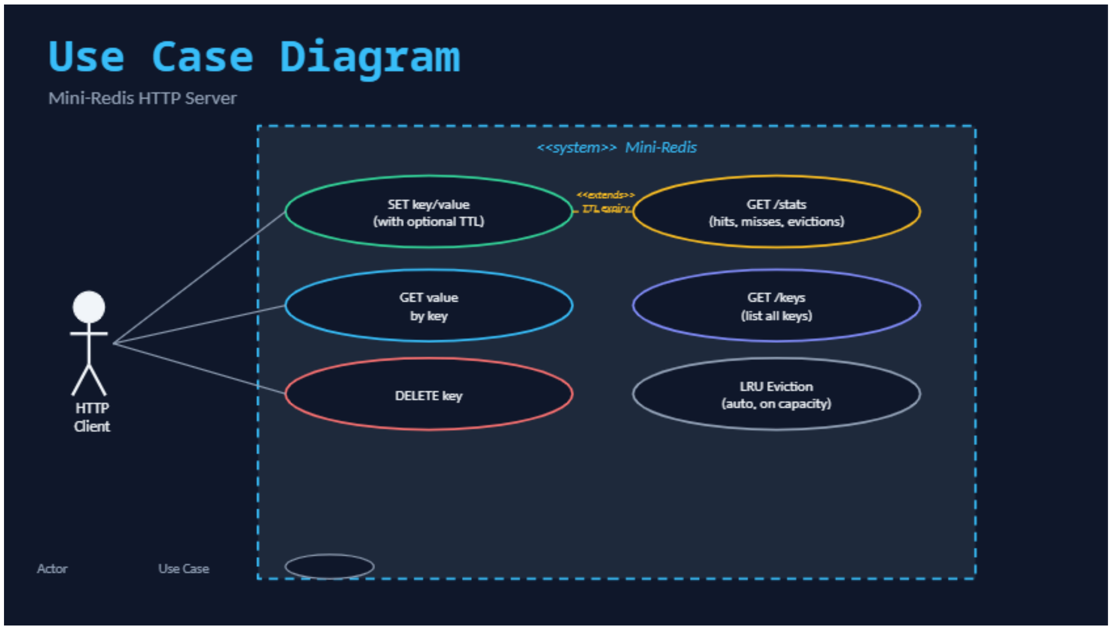


Fig 3.6: Use Case Diagram

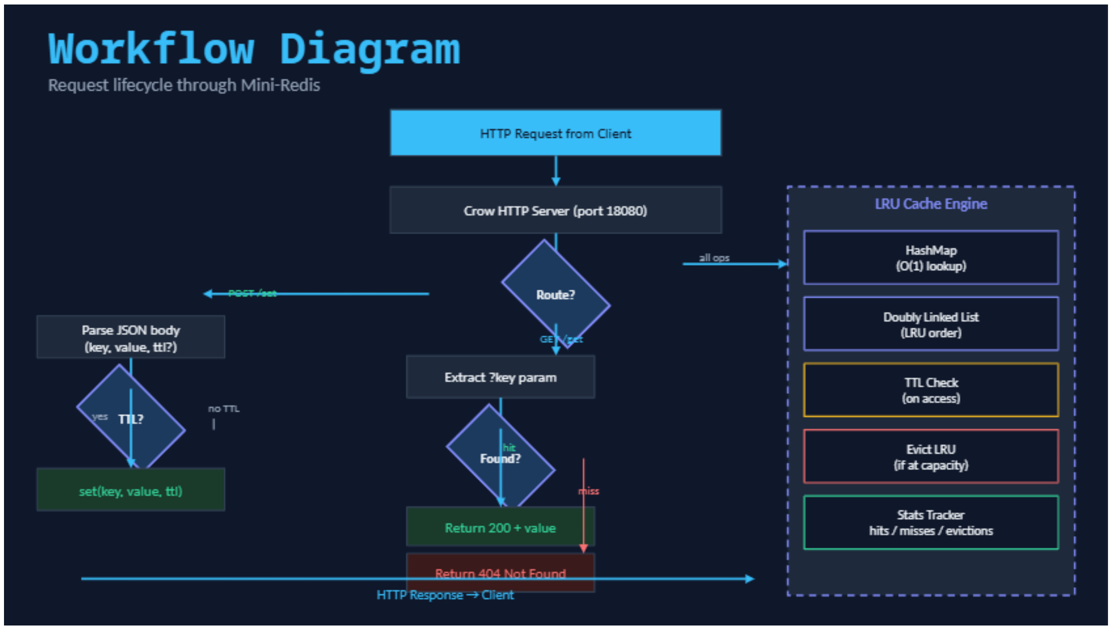


Fig 3.7: Workflow Diagram

Chapter 4 Discussion on the achievements

The major achievement of this project is the implementation of a fully functional LRU (Least Recently Used) Cache using C++. The system has been implemented correctly, with proper order being maintained and automatic deletion of the least recently used item when the maximum capacity is reached. The implementation has ensured time complexity of O(1) using a hash map and a doubly linked list.

Another major achievement is the integration of the LRU Cache with a RESTful backend using the Crow framework. This has enabled operations such as set, get, delete, and retrieving statistics using HTTP requests, simulating a backend system.

The implementation of a simple web-based frontend is another achievement of this project, as it enhances the system by enabling users to view the visualization of the cache system. The system has enabled users to see how recently used items are moved to the front of the cache and how older items are removed.

4.1 Features

The system includes several key features designed to provide a realistic, educational, and user-friendly trading experience:

- **LRU Cache Implementation**: Efficient storage with automatic eviction of least recently used items.
- **Set Key-Value**: Add key-value pairs with optional TTL (time-to-live).
- **Get Value**: Retrieve the value of a specific key.
- **Delete Key**: Remove a key-value pair from the cache.
- **List Keys**: View all current keys in the cache.
- **Cache Statistics**: See total requests, hits, misses, evictions, and current cache size.
- **Web-based UI**: Simple frontend to interact with the cache in real-time.
- **Cross-platform Server**: Runs on Windows and Linux using Crow + Asio.
- **Multithreading Support**: Handles multiple requests concurrently.

Chapter 5 Conclusion and Recommendation

The project has been implemented successfully, reflecting the use of an LRU cache along with the core operations such as set, get, delete, and all keys, along with the use of a simple web interface for the purpose of interacting with the cache in real time. The project has been implemented while considering the key statistics associated with the cache, such as hits, misses, evictions, and total requests.

For the purpose of learning and for further use, the project can be considered a starting point for learning the concepts associated with the caching mechanisms used in the context of web applications, along with the development of the API for the purpose of visualization. For further learning and development, the project can be used for the purpose of creating complex applications, such as the use of persistent storage, along with the development of the REST API.

5.1 Limitations

- Cache **does not persist data**; all data is lost when the server stops.
- **TTL expiration** is basic; no automatic cleanup of expired keys.
- **Concurrency** is limited to Crow's multithreaded handling; no fine-grained locking.
- **Security** and authentication are not implemented; anyone can access the API.

5.2 Future Enhancement

- Add **persistent storage** so cached data survives server restarts.
- Implement **background TTL cleanup** for automatic expiration.
- Improve **concurrency control** with locks or atomic operations.
- Add **authentication and authorization** to secure the API.
- Enhance frontend with **real-time updates** and better visualization of cache stats.

References

Crow C++ microframework. (n.d.). *Crowcpp.org*. Retrieved February 21, 2026, from https://crowcpp.org/

GeeksforGeeks. (n.d.). *LRU cache implementation*. Retrieved February 15, 2026, from https://www.geeksforgeeks.org/lru-cache-implementation/

Boost.Asio. (n.d.). *Think-Async.com: Asio C++ library*. Retrieved February 22, 2026, from https://think-async.com/Asio/

MDN Web Docs. (n.d.). *HTTP overview*. Mozilla. Retrieved February 18, 2026, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Crow C++ microframework. (n.d.). *JSON handling in Crow*. Retrieved February 19, 2026, from https://crowcpp.org/master/tutorials/json/

Wikipedia contributors. (n.d.). *Cache replacement policies*. In *Wikipedia*. Retrieved February 15, 2026, from https://en.wikipedia.org/wiki/Cache_replacement_policies

APPENDIX

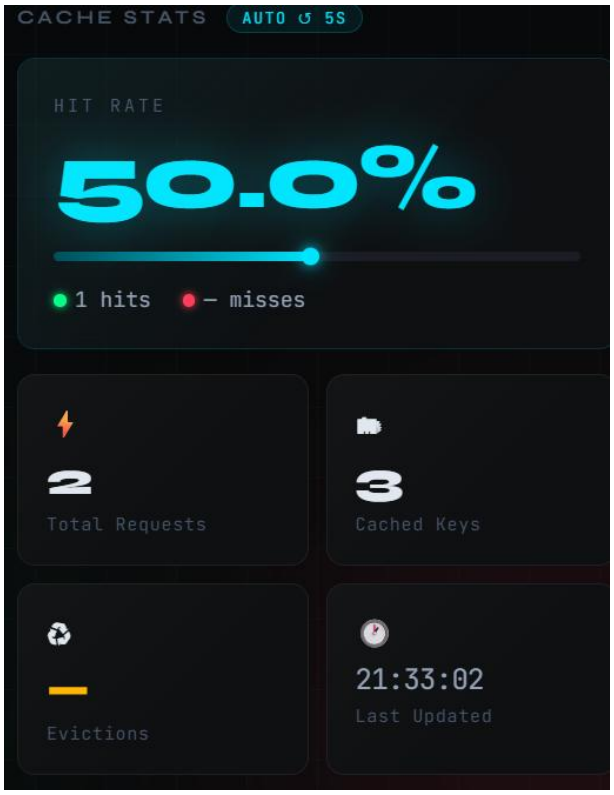
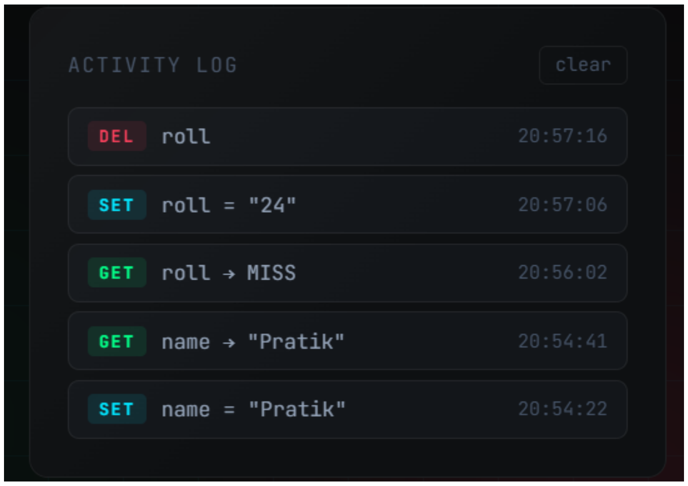


Fig A1: Stat Retrieval Legend



```
ACTIVITY LOG                                clear
DEL  roll                                   20:57:16
SET  roll = "24"                            20:57:06
GET  roll → MISS                            20:56:02
GET  name → "Pratik"                        20:54:41
SET  name = "Pratik"                        20:54:22
```

Fig A2: Activity Log