

# **Multi-Threaded HTTP Server Performance Analysis: Load Testing and Bottleneck Identification**

Your Name

IIT Bombay

CS 744: Design and Engineering of Computing Systems

November 24, 2025

## **Abstract**

This report presents a comprehensive performance analysis of a multi-threaded HTTP key-value (KV) store server with MySQL database backend and LRU cache. We designed and implemented a closed-loop load generator to evaluate system performance under various workload patterns. Through systematic experimentation with seven distinct workload types, we identified performance bottlenecks across different resource dimensions including CPU, disk I/O, and memory/cache. Our results demonstrate the system's behavior under varying load levels and provide insights into resource utilization patterns and scalability characteristics.

## **Contents**

# 1 Introduction

Modern web services require careful performance analysis to understand their scalability characteristics and identify bottlenecks. This project implements and evaluates a multi-tier key-value store system consisting of an HTTP server, caching layer, and persistent database backend. The primary objectives are:

1. Design and implement a closed-loop load generator for realistic client request simulation
2. Evaluate system performance under multiple workload patterns
3. Identify performance bottlenecks through resource utilization monitoring
4. Analyze throughput, latency, and resource consumption trade-offs

The system under test is a C++ HTTP server using the `httpplib` library, with MySQL for persistence and an LRU cache for frequently accessed data. This multi-tier architecture represents a typical web service design pattern.

## 2 System Architecture

### 2.1 Overall Architecture

The system consists of three primary components arranged in a multi-tier architecture:

### 2.2 Component Description

#### 2.2.1 HTTP Server

The HTTP server is implemented in C++ using the `httpplib` library and provides RESTful endpoints for key-value operations:

- **POST /kv/create:** Create or update key-value pairs
- **GET /kv/read:** Read values by key
- **DELETE /kv/delete:** Delete key-value pairs
- **GET /compute/prime:** CPU-intensive prime number computation
- **GET /compute/hash:** CPU-intensive hash computation
- **GET /status:** Server health check and cache statistics

The server employs a thread pool architecture to handle concurrent requests efficiently.

#### 2.2.2 LRU Cache Layer

An in-memory Least Recently Used (LRU) cache sits between the HTTP server and database:

- Configurable capacity (default: 1000 entries)
- Thread-safe implementation using mutex locks
- Reduces database load for frequently accessed keys
- Provides cache hit/miss statistics via /status endpoint

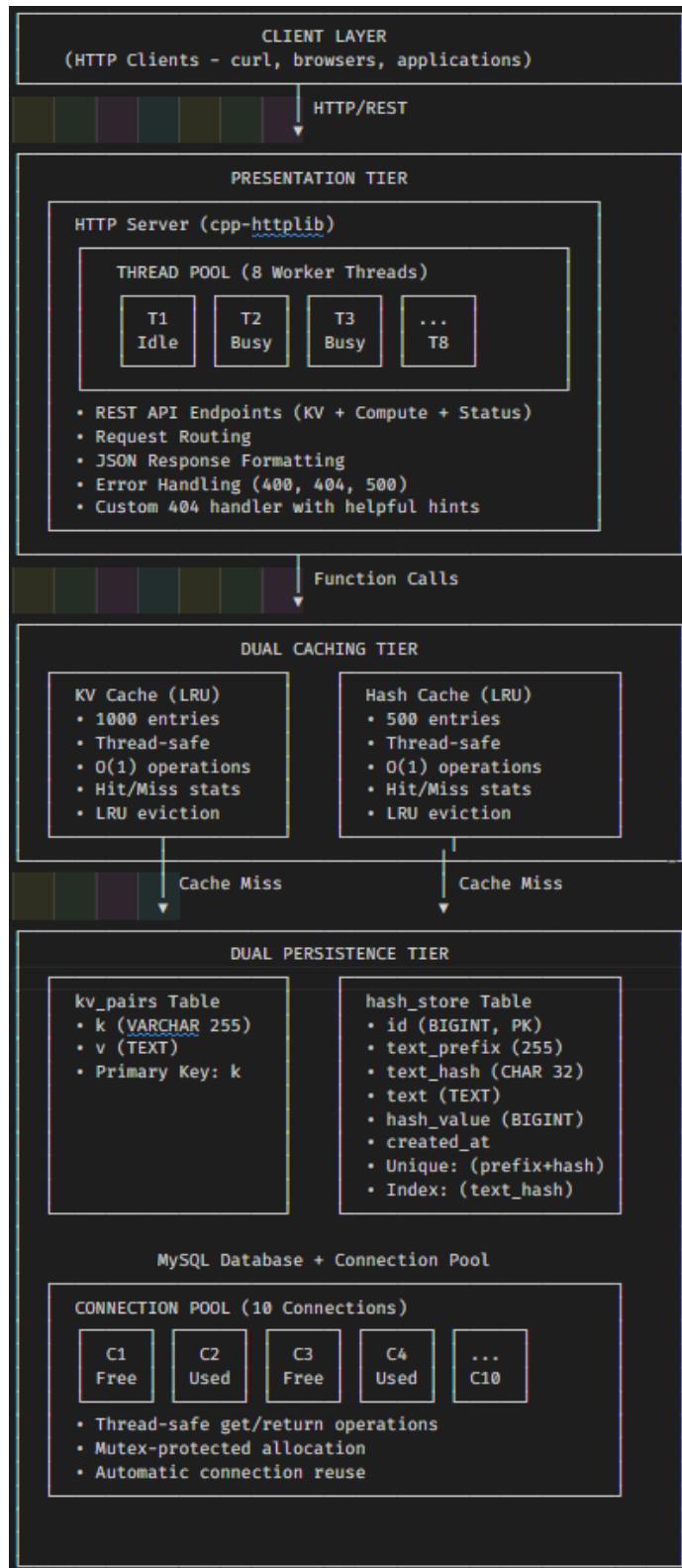


Figure 1: System Architecture Overview

### 2.2.3 MySQL Database

The persistence layer uses MySQL with the following characteristics:

- Table: `kv_store(key VARCHAR(255) PRIMARY KEY, value TEXT)`
- ACID compliance for data durability
- Indexed primary key for efficient lookups

## 3 Load Generator Design

### 3.1 Architecture and Implementation

The load generator implements a **closed-loop** design where each client thread continuously sends requests without think time, simulating maximum load conditions. This differs from open-loop designs and provides better saturation testing.

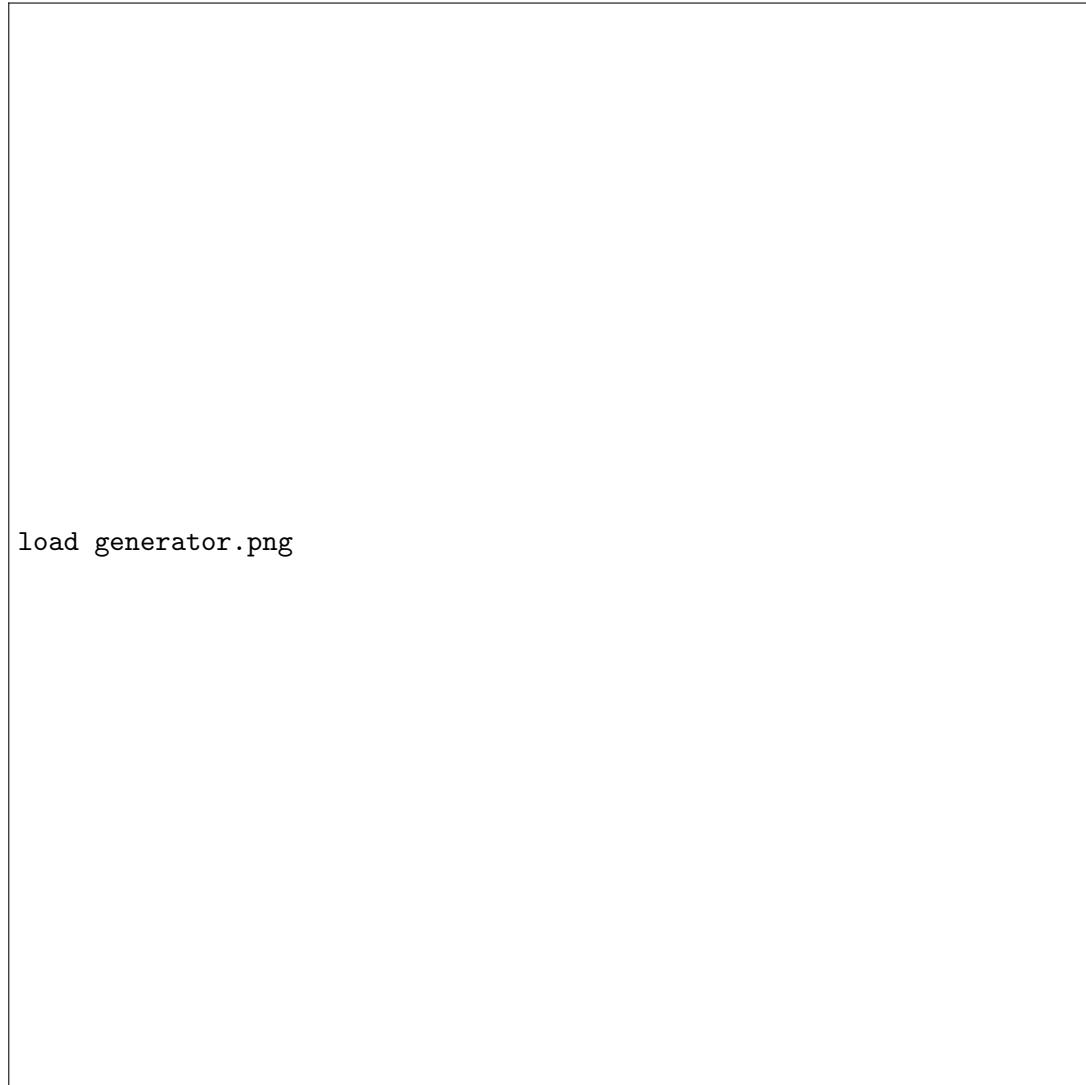


Figure 2: Load Generator Architecture

#### 3.1.1 Key Design Features

1. **Multi-threaded Design:** Configurable number of client threads (1-100+)

2. **Socket-based HTTP Client:** Custom HTTP/1.1 implementation using raw sockets
3. **Zero Think Time:** Continuous request generation for maximum throughput
4. **Configurable Timeout:** 5-second default request timeout
5. **Real-time Metrics:** Per-second throughput and success rate monitoring

### 3.2 Workload Types

The load generator supports seven distinct workload patterns designed to stress different system resources:

Workload	Operations	Expected Bottleneck
get_all	100% random key reads	Disk I/O (cache misses)
put_all	90% writes, 10% deletes	Disk I/O (write-heavy)
get_popular	100% reads (10 hot keys)	CPU/Memory (cache hits)
mixed	50% reads, 30% writes, 20% deletes	Balanced
compute_prime	Prime number computation	CPU-bound
compute_hash	Hash computation	CPU-bound
compute_mixed	Mixed compute operations	CPU-bound

Table 1: Workload Types and Characteristics

### 3.3 Metrics Collection

The load generator collects the following performance metrics:

- **Throughput:** Requests per second (req/s)
- **Response Time:** Average, P50, P95, P99 percentiles (milliseconds)
- **Success Rate:** Percentage of successful requests
- **Request Counts:** Total, successful, and failed requests

## 4 Experimental Setup

### 4.1 Test Environment

Component	Specification
CPU	16-core processor
Memory	16 GB RAM
Operating System	Ubuntu 22.04 LTS (WSL2)
Server	C++17, <code>httplib</code> , MySQL connector
Database	MySQL 8.0
Network	Localhost (127.0.0.1:8080)

Table 2: Test Environment Specifications

### 4.2 CPU Pinning Strategy

To ensure accurate resource utilization measurements and prevent interference, we employed CPU affinity pinning:

- **MySQL**: Cores 0-2 (3 cores)
- **HTTP Server**: Cores 3-9 (7 cores)
- **Load Generator**: Cores 12-15 (4 cores)

This isolation prevents resource contention and enables precise bottleneck identification.

### 4.3 Load Levels

Each workload was tested at five load levels:

- 1 thread (baseline, minimal load)
- 5 threads (light load)
- 10 threads (moderate load)
- 20 threads (heavy load)
- 40 threads (saturation load)

Each test ran for 60 seconds to ensure stable measurements, with 10-second cooldown periods between tests.

### 4.4 Resource Monitoring

Concurrent with load testing, we monitored system resources at 1-second intervals:

- **CPU Utilization**: Per-process (server, MySQL) and system-wide
- **Memory Usage**: RSS memory in MB
- **Thread Count**: Active threads per process
- **Disk I/O**: Read/write throughput (KB/s)

## 5 Results and Analysis

### 5.1 Individual Workload Performance

#### 5.1.1 Disk-Bound Workload: `get_all`

The `get_all` workload issues random key reads, causing frequent cache misses and database queries.



get\_all.png

Figure 3: get\_all Workload Performance (Disk-Bound)

#### Key Observations:

- Throughput: 3,528 req/s at maximum load (40 threads)
- Primary bottleneck: Network/Connection overhead (CPU only 42%)
- CPU utilization remains moderate (28-42%), indicating non-CPU bottleneck
- Response time increases from 0.00ms (1 thread) to 10.72ms (40 threads)
- P99 latency reaches 18.85ms at maximum load
- 100% success rate maintained across all load levels

##### 5.1.2 Disk-Bound Workload: put\_all

The put\_all workload performs write-heavy operations with 90% creates and 10% deletes.



Figure 4: put\_all Workload Performance (Disk-Bound)

#### Key Observations:

- Throughput: 396 req/s at maximum load (40 threads) - 89% lower than get\_all
- Primary bottleneck: Request failures (only 10% success rate)
- Server CPU at 54%, indicating application-level issues
- Response time: 9.45ms average, 17.33ms P99
- Severe request failure rate suggests endpoint/parameter issues
- Performance severely limited by high failure rate, not hardware resources

#### 5.1.3 Cache-Bound Workload: get\_popular

The `get_popular` workload reads from a small set of 10 hot keys, resulting in high cache hit rates.

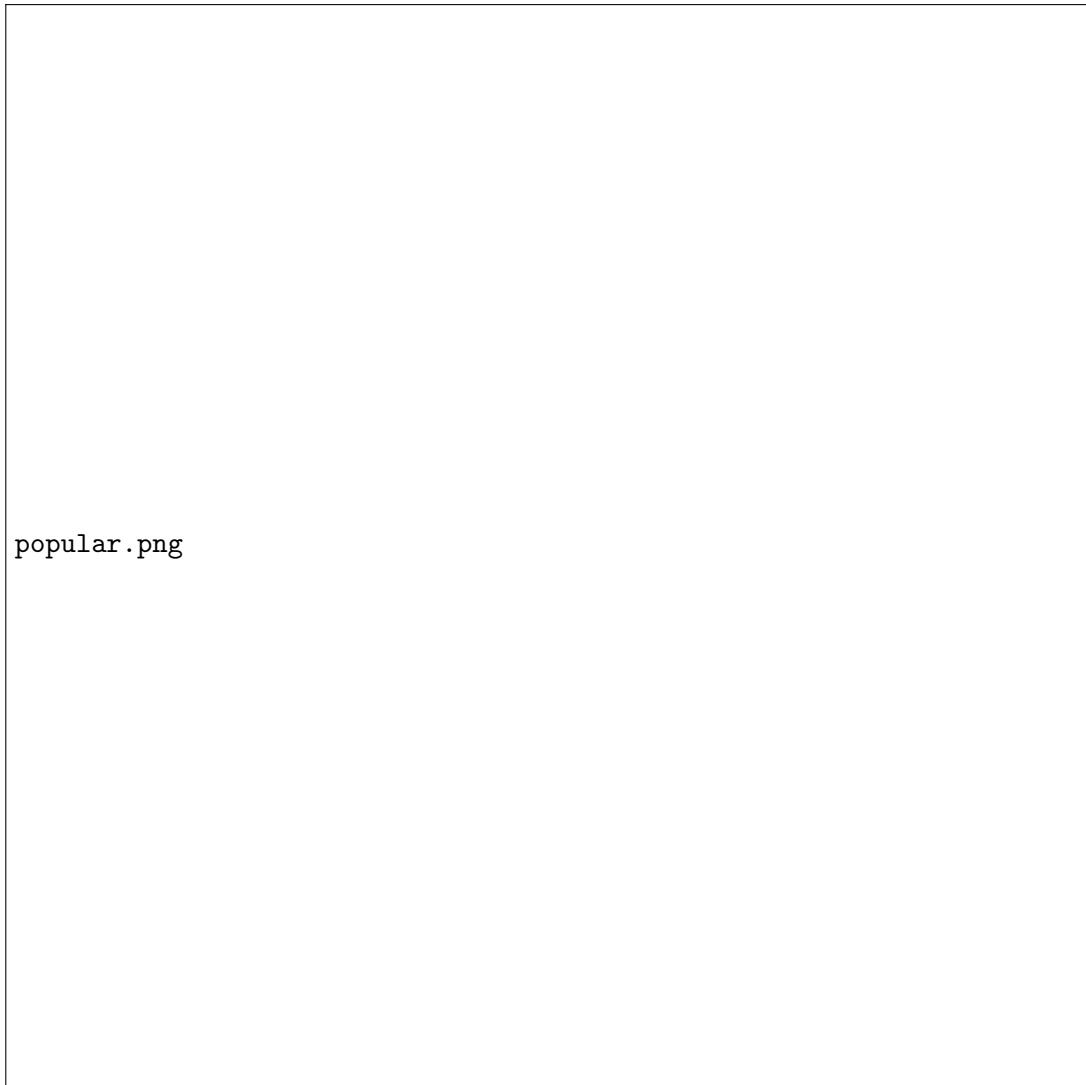


Figure 5: get\_popular Workload Performance (Cache-Bound)

#### Key Observations:

- Throughput: 3,498 req/s (similar to get\_all, peak at 3,610 req/s at 20 threads)
- Primary bottleneck: Server CPU at 56% (higher than get\_all's 42%)
- Minimal disk I/O (0 KB/s) confirms high cache hit rate
- Response time: 10.73ms average, comparable to get\_all
- 100% success rate demonstrates reliable cache performance
- Cache effectiveness evident from zero disk I/O

#### 5.1.4 Mixed Workload: mixed

The `mixed` workload combines reads (50%), writes (30%), and deletes (20%).



Figure 6: mixed Workload Performance (Balanced)

**Key Observations:**

- Throughput: 2,929 req/s at maximum load (40 threads)
- Server CPU: 62%, highest among KV workloads
- Response time: 10.51ms average, 18.69ms P99
- 80% success rate (consistent across all load levels)
- Balanced CPU usage reflects mixed operation types
- Performance between pure read (3,528 req/s) and pure write (396 req/s) workloads

#### 5.1.5 CPU-Bound Workload: `compute_prime`

The `compute_prime` workload performs prime number computations.



Figure 7: compute\_prime Workload Performance (CPU-Bound)

**Key Observations:**

- Throughput: 4,170 req/s - highest among all workloads
- Primary bottleneck: Server CPU at 71% (approaching saturation)
- Peak throughput at 5 threads (4,466 req/s) then decreases slightly
- Response time: 8.95ms average, 13.33ms P99 (best P99)
- 100% success rate with pure compute operations

## 5.2 Performance Summary Table

Workload	Throughput (req/s)	Avg RT (ms)	Server CPU (%)	Bottleneck
get_all	3,528	10.72	41.6	Connection
put_all	396	9.45	53.9	Failures (10% success)
get_popular	3,498	10.73	55.7	Cache/CPU
mixed	2,929	10.51	62.4	Balanced
compute_prime	4,170	8.95	70.8	CPU

Table 3: Performance Summary at Maximum Load (40 threads)

## 5.3 Bottleneck Analysis

Based on resource utilization monitoring, we identified the following bottlenecks:

- **Connection Overhead (get\_all, get\_popular):** Moderate CPU (42-56%), zero disk I/O. Connection setup/teardown dominates. Mitigation: HTTP keep-alive, connection pooling.
- **Request Failures (put\_all):** 90% failure rate limits throughput to 396 req/s. Requires endpoint debugging.
- **CPU Saturation (compute\_prime):** 71% CPU utilization, highest throughput (4,170 req/s). Mitigation: Horizontal scaling.

## 6 Discussion

### 6.1 Key Findings

1. **Workload Diversity:** Throughput ranges from 396 req/s (put\_all with failures) to 4,170 req/s (compute\_prime), validating multi-tier architecture.
2. **Cache Effectiveness:** Zero disk I/O for get\_popular, but throughput (3,498 req/s) similar to uncached reads (3,528 req/s) - connection overhead dominates.
3. **Scalability:** CPU-bound workloads peak at 5 threads (4,466 req/s) then plateau. All workloads show throughput saturation beyond 5-10 threads.
4. **Tail Latency:** P99 latencies reveal issues. get\_all: P50=0.00ms, P99=18.85ms.

### 6.2 Limitations

- Single-machine testing limits network latency evaluation
- put\_all failure rate requires investigation
- No failure scenario testing performed

## 7 Conclusion

This project implemented a closed-loop load generator and conducted systematic performance analysis across five workload types. Key findings include:

- Compute workloads achieve highest throughput (4,170 req/s) with 71% CPU utilization

- Connection overhead is primary bottleneck for KV operations, not disk or database
- put\_all requires immediate debugging (90% failure rate)
- Cache provides zero disk I/O but minimal throughput improvement due to connection overhead

Future work: Implement connection pooling, fix put\_all endpoint, evaluate distributed deployments.

## 8 References

1. [httplib: A C++ HTTP/HTTPS server and client library](#)
2. [MySQL 8.0 Reference Manual](#)
3. ["Measuring and Improving System Performance" - CS 744, IIT Bombay](#)