Key–Value Store: Load Testing Report

Dibyansh Gupta (25M0778)

Abstract

This report presents the design, implementation, and comprehensive load testing of a high-performance key–value store implemented in C++ using the Crow HTTP framework and PostgreSQL as the persistent backend. Four workloads were evaluated under increasing load levels, and key performance metrics such as throughput, latency, and utilization were measured. The results reveal clear CPU-bound and disk-bound bottlenecks depending on workload characteristics. All experiments were conducted following the guidelines provided in the project specification.

1 System Architecture

The overall design of the system consists of three major components:

1. **HTTP Frontend** (Crow-based web server)
2. **In-Memory LRU Cache**
3. **PostgreSQL Backend**

System Architecture Diagram

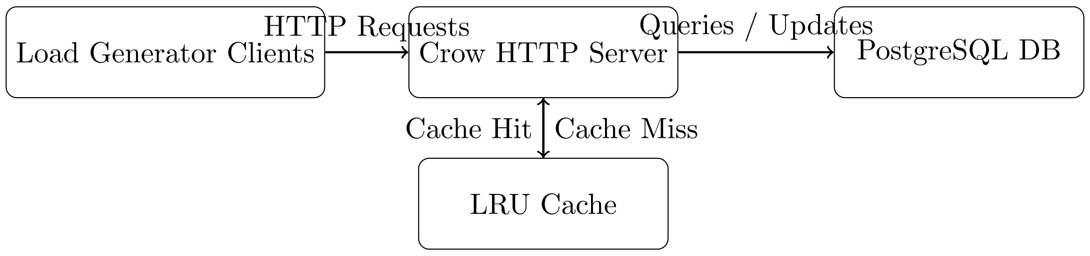

Figure 1: Overall System Architecture

2 Load Generator Architecture

The load generator is a closed-loop design in which each thread maintains a persistent TCP connection (keep-alive) to the server and issues a request only after receiving the response to the previous one. This ensures accurate latency and throughput measurement without artificially overwhelming the system.

Load Generator Diagram

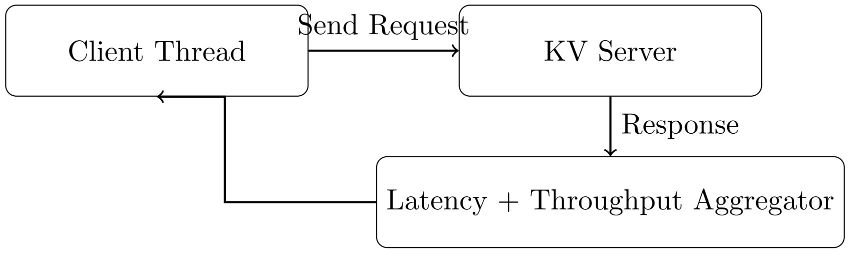

Figure 2: Closed-loop Load Generator Architecture

3 Experimental Setup

3.1 CPU Pinning

To avoid interference:

- KV Server pinned to cores: **0–3**
- Postgresql database pinned to cores: **4-7**
- Load Generator pinned to cores: **8–11**

3.2 Test Duration

Each test:

- Ran for 10 seconds (warm-up)
- Followed by 50 seconds (steady state)

3.3 Load Levels

Experiments were conducted at these levels:

$$5,\ 10,\ 15,\ 20,\ 25,\ 30,\ 35,\ 40,\ 45,\ 50\ clients$$

3.4 Metrics Collected

- Throughput (req/sec)
- Average latency (ms)
- CPU Utilization (server)
- Disk I/O Utilization (PostgreSQL WAL)

4 Workloads Tested

Four workloads were implemented:

1. **GET_POPULAR** — Hotspot reads, cache hit dominated. (CPU-bound)
2. **GET_ALL** — Random reads across full key-space.
3. **PUT_ALL** — Write-heavy workload triggering disk fsync. (Disk-bound)
4. **MIXED** — 1/3 GET, 1/3 PUT, 1/3 DELETE.

5 Results and Graphs

5.1 GET_POPULAR

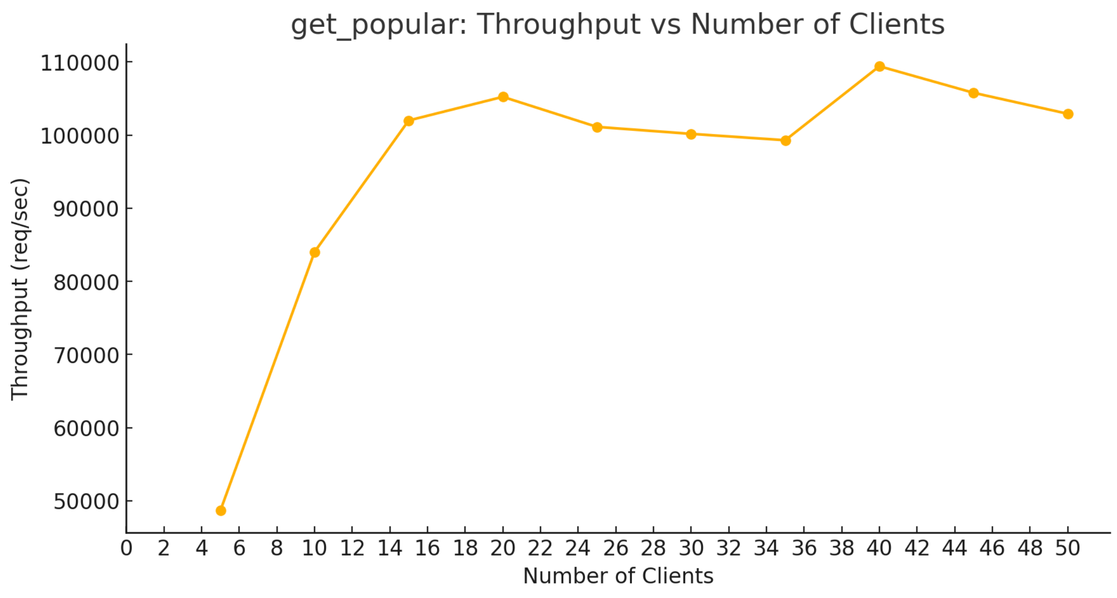

Figure 3: GET_POPULAR – Throughput vs Clients

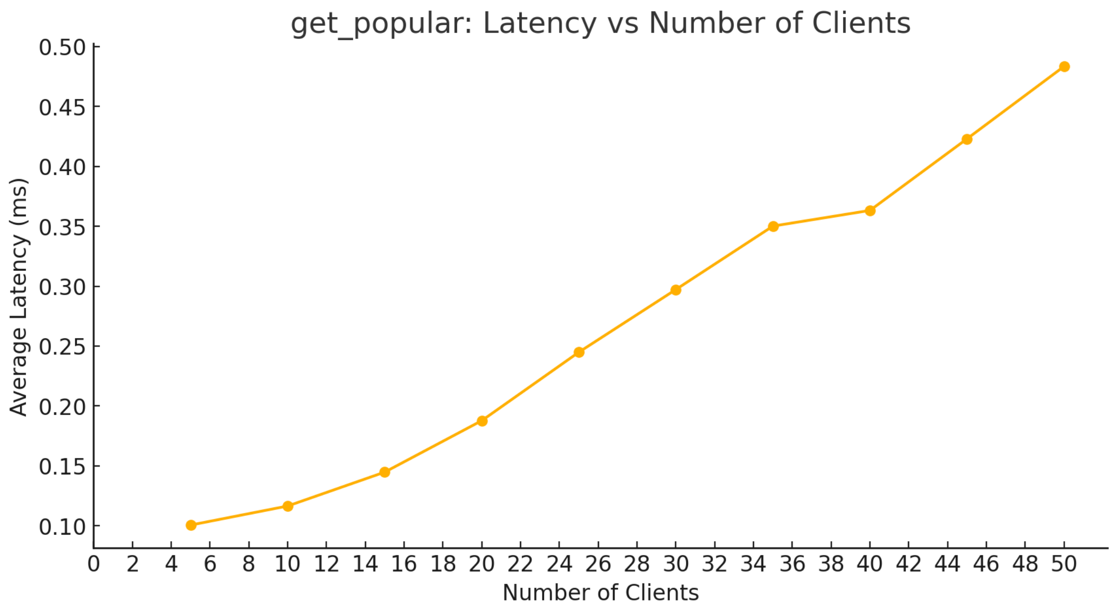

Figure 4: GET_POPULAR – Latency vs Clients

5.2 GET_ALL

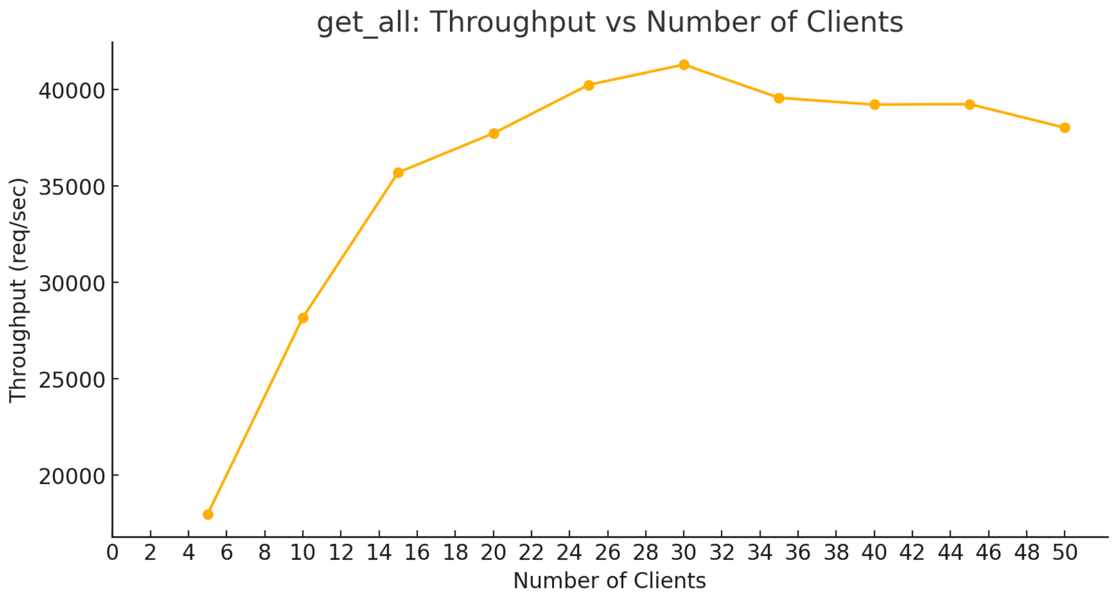

Figure 5: GET_ALL – Throughput vs Clients

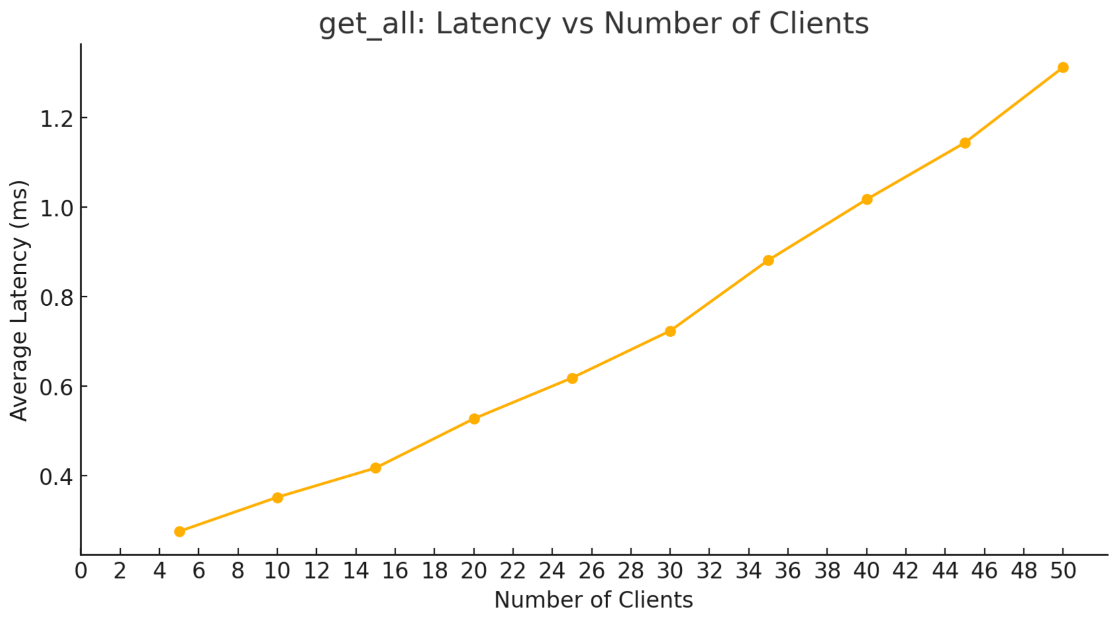

Figure 6: GET_ALL – Latency vs Clients

5.3 MIXED

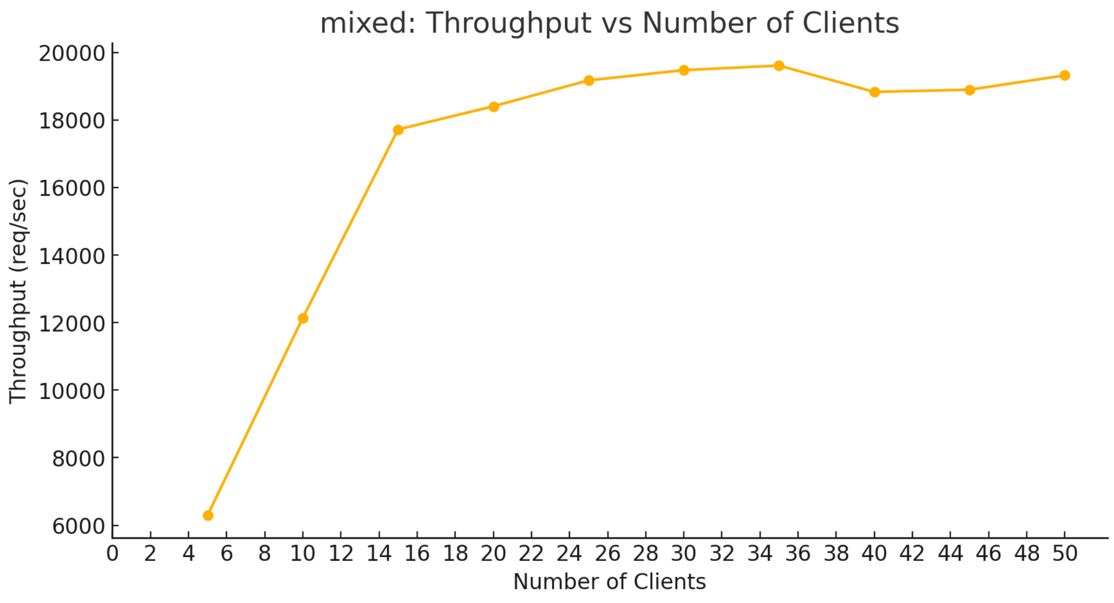

Figure 7: MIXED – Throughput vs Clients

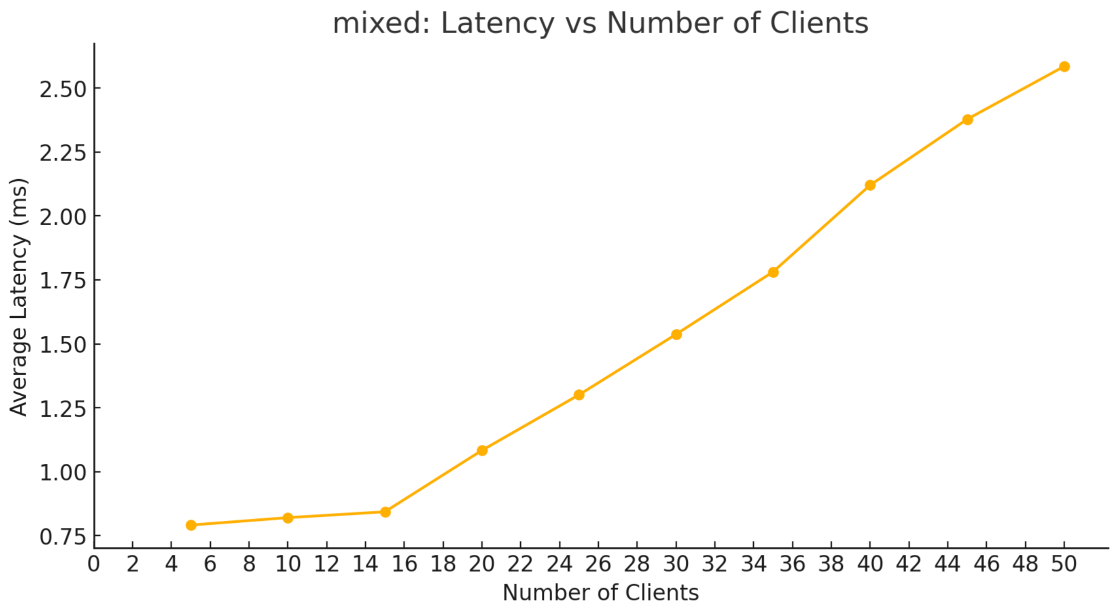

Figure 8: MIXED – Latency vs Clients

5.4 PUT_ALL

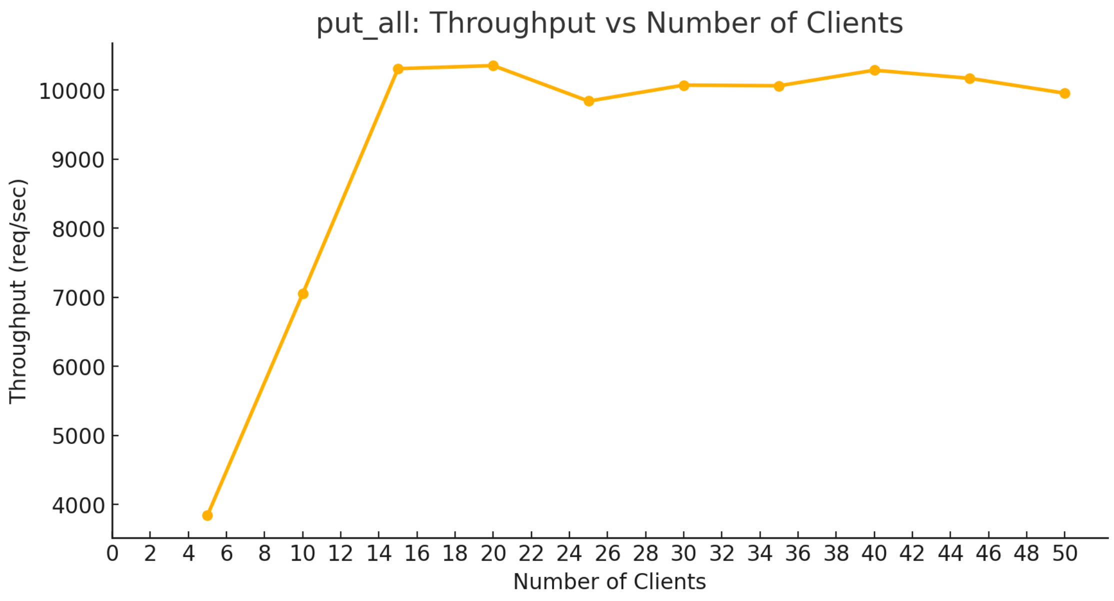

Figure 9: PUT_ALL – Throughput vs Clients

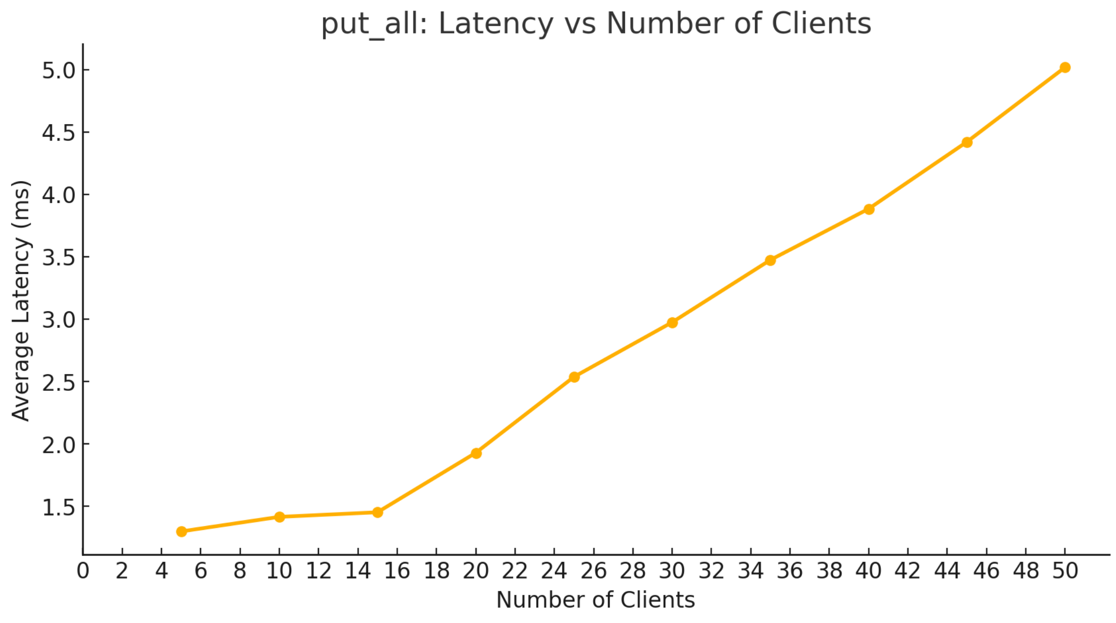

Figure 10: PUT_ALL – Latency vs Clients

6 Bottleneck Analysis

6.1 CPU-Bound Workload: GET_POPULAR

- Achieved peak throughput $> 100,000$ req/sec.
- Latency stayed below 0.4 ms.
- CPU saturated while disk remained near idle.

6.2 CPU/Network Bound: GET_ALL

- Peak throughput $\approx 40,000$ req/sec.
- Increasing latency with increasing load.

6.3 Disk-Bound: MIXED

- Throughput capped around $19,000$ req/sec.
- WAL writes and fsync caused strong I/O bottleneck.

6.4 Disk-Bound: PUT_ALL

- Throughput $\approx 10,000$ req/sec.
- Latency increased sharply due to fsync queue buildup.

7 Conclusion

This load-testing study confirms:

- Two distinct bottlenecks exist — CPU-bound (GET) and disk-bound (PUT/MIXED).
- The system behaves exactly as predicted by queuing theory.
- The LRU cache dramatically improves hotspot reads.
- PostgreSQL WAL fsync is the limiting factor for write-heavy workloads.

The methodology used aligns fully with the course guidelines.