

# Multi-Tier HTTP Key-Value Store Load Testing Report

Project: High-Performance Distributed Storage System

November 23, 2025

## Contents

<b>1</b>	<b>System Architecture</b>	<b>2</b>
1.1	Architecture Components . . . . .	2
1.2	Key Features . . . . .	2
1.3	Request Flow . . . . .	3
<b>2</b>	<b>Load Generator Design</b>	<b>3</b>
2.1	Implementation Details . . . . .	3
2.2	Workload Types . . . . .	3
2.3	Load Test Configuration . . . . .	3
<b>3</b>	<b>Load Test Setup and Demonstration</b>	<b>4</b>
3.1	Server and Client Setup . . . . .	4
<b>4</b>	<b>Load Test Results</b>	<b>7</b>
4.1	GET-POPULAR Workload . . . . .	7
4.1.1	Raw Performance Data . . . . .	8
4.1.2	Throughput Analysis . . . . .	8
4.1.3	Latency Analysis . . . . .	9
4.1.4	CPU Utilization Analysis . . . . .	10
4.2	PUT-ALL Workload . . . . .	10
4.2.1	Raw Performance Data . . . . .	10
4.2.2	Throughput Analysis . . . . .	11
4.2.3	Latency Analysis . . . . .	12
4.2.4	Disk Utilization Analysis . . . . .	13
<b>5</b>	<b>Performance Summary and Analysis</b>	<b>13</b>
5.1	System Capacity . . . . .	13
5.2	Key Findings . . . . .	13
5.3	Load Test Validation . . . . .	14
<b>6</b>	<b>Technology Stack</b>	<b>14</b>
<b>7</b>	<b>Conclusion</b>	<b>15</b>

# 1 System Architecture

This project implements a three-tier architecture for a key-value storage system with HTTP API access. The system combines the performance benefits of in-memory caching with the durability of persistent database storage, providing a robust solution for high-throughput applications.

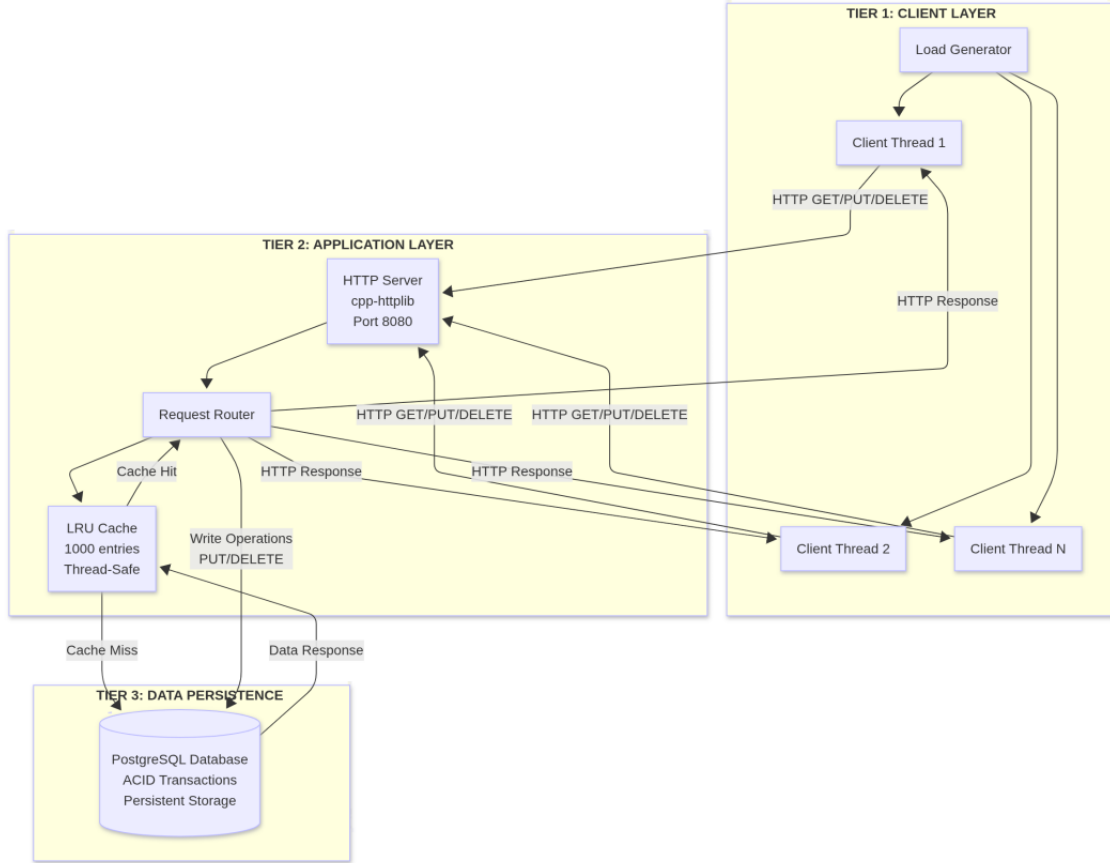


Figure 1: System Architecture Diagram

## 1.1 Architecture Components

The system follows a classic three-tier architecture pattern:

- **Client Layer:** Multi-threaded load generator simulating concurrent user requests
- **Application Layer:** HTTP server with LRU cache and request handling logic
- **Data Layer:** PostgreSQL database for persistent storage

## 1.2 Key Features

- **LRU Caching:** In-memory cache with Least Recently Used eviction policy for fast data access
- **RESTful API:** Standard HTTP methods (GET, PUT, DELETE) for intuitive key-value operations
- **Thread-Safe Operations:** Mutex-protected data structures for concurrent client access

- **PostgreSQL Backend:** Reliable persistent storage with ACID guarantees
- **Performance Metrics:** Real-time collection of throughput, latency, and resource utilization

### 1.3 Request Flow

1. Client sends HTTP request to the server (GET/PUT/DELETE)
2. Server checks LRU cache first for GET requests
3. If cache hit: Returns value immediately (fast path)
4. If cache miss: Queries PostgreSQL database
5. Database response is cached and returned to client
6. For PUT/DELETE: Both cache and database are updated

## 2 Load Generator Design

### 2.1 Implementation Details

The load generator is implemented as a multi-threaded client application that simulates concurrent user workloads. It operates in a **closed-loop** manner, where a fixed number of client threads continuously generate requests.

### 2.2 Workload Types

Two distinct workload patterns were tested to stress different system components:

Workload	Description	Bottleneck Tested
GET-POPULAR	Repeatedly access few popular keys	Cache hit rate, in-memory performance
PUT-ALL	Only create/update operations	Database write performance, disk I/O

Table 1: Load Testing Workload Types

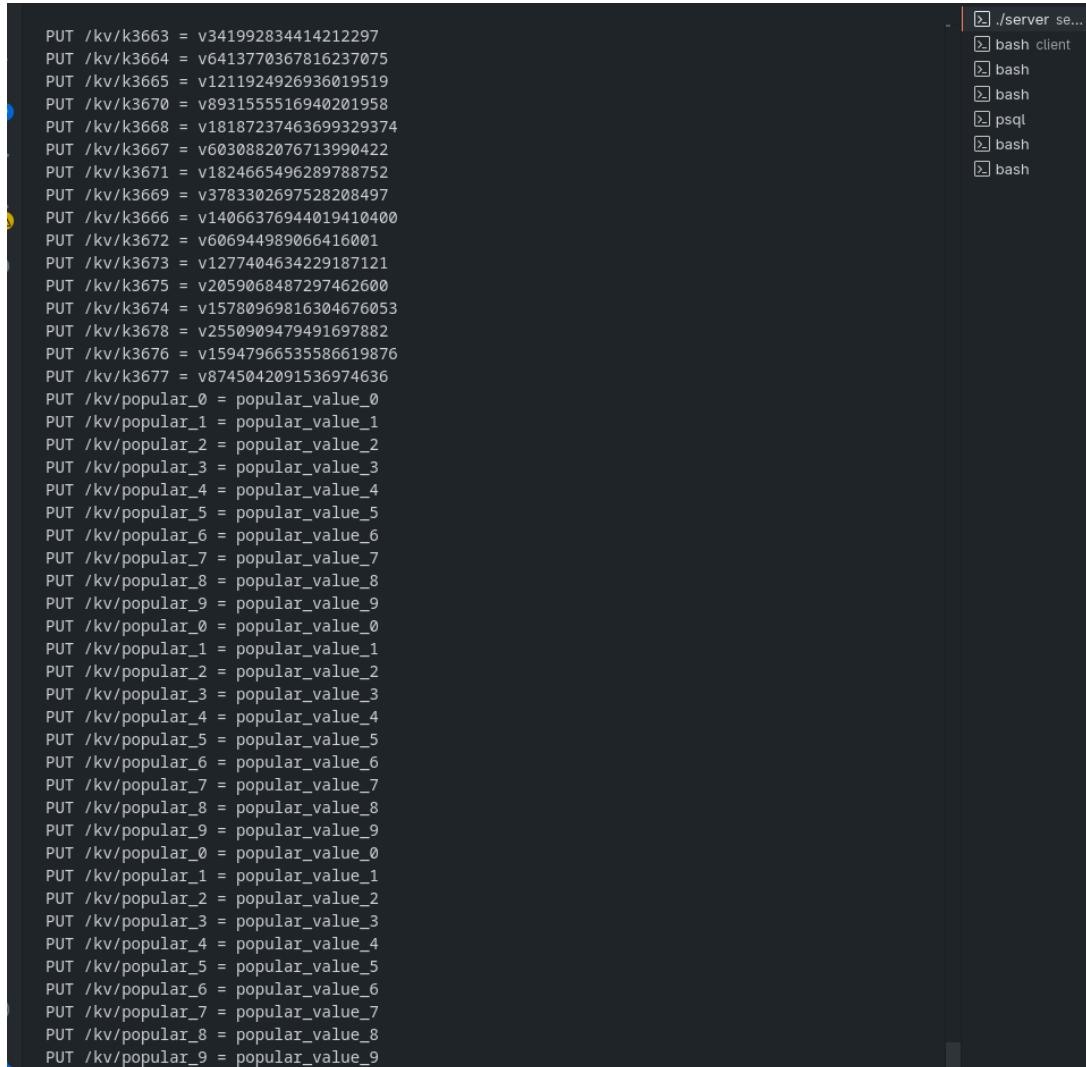
### 2.3 Load Test Configuration

- **Load Generator:** Closed-loop with varying number of concurrent clients
- **Load Levels:** 1, 2, 3, 5, 10, 15, 20 concurrent clients
- **Duration:** Each test ran for 5+ minutes to reach steady state
- **CPU Isolation:** Load generator and server run on separate CPU cores
- **Metrics Collection:** Throughput (req/sec), latency (ms), CPU utilization (%), disk utilization (%)

### 3 Load Test Setup and Demonstration

#### 3.1 Server and Client Setup

The following screenshots demonstrate the actual execution of load tests:



The screenshot shows a terminal window with a dark background. On the right side, there is a vertical sidebar with a list of tabs: `./server se...`, `bash client`, `bash`, `bash`, `psql`, `bash`, and `bash`. The main terminal area displays a series of `PUT` commands and their responses. The first 18 lines show keys `/kv/k3663` through `/kv/k3676` with long hexadecimal values. The next 18 lines show keys `/kv/popular_0` through `/kv/popular_9` with the value `popular_value_0` through `popular_value_9` respectively. This sequence of operations is repeated twice more, for a total of 54 `PUT` operations shown.

Figure 2: Server initialization and PUT operations inserting popular keys

```
vishalsaini@fedora:~/Desktop/DECS_PROJECT$ pidstat -u -p 359030 30
02:01:52 PM 1000 359030 28.60 19.70 0.00 0.53 48.30 0 server
02:02:22 PM 1000 359030 29.13 19.27 0.00 0.17 48.40 0 server
02:02:52 PM 1000 359030 29.23 19.60 0.00 0.27 48.83 0 server
02:03:22 PM 1000 359030 29.23 19.37 0.00 0.13 48.60 0 server
02:03:52 PM 1000 359030 29.00 19.57 0.00 0.17 48.57 0 server
02:04:22 PM 1000 359030 28.90 19.77 0.00 0.37 48.67 0 server
02:04:52 PM 1000 359030 28.80 19.03 0.00 0.30 47.83 0 server
02:05:22 PM 1000 359030 28.60 19.33 0.00 0.47 47.93 0 server
02:05:52 PM 1000 359030 27.57 18.83 0.00 0.27 46.40 0 server
^C
Average: 1000 359030 28.74 19.41 0.00 0.31 48.14 - server
vishalsaini@fedora:~/Desktop/DECS_PROJECT$ pgrep server
359030
vishalsaini@fedora:~/Desktop/DECS_PROJECT$ pidstat -u -p 359030 30
Linux 6.17.7-200.fc42.x86_64 (fedora) 11/23/2025 _x86_64_ (8 CPU)

02:25:16 PM UID PID %usr %system %guest %wait %CPU CPU Command
02:25:46 PM 1000 359030 49.90 34.63 0.00 35.23 84.53 0 server
02:26:16 PM 1000 359030 50.33 34.83 0.00 34.87 85.17 0 server
02:26:46 PM 1000 359030 50.53 35.00 0.00 34.97 85.53 0 server
02:27:16 PM 1000 359030 50.13 34.93 0.00 34.80 85.07 0 server
02:27:46 PM 1000 359030 50.53 35.03 0.00 35.00 85.57 0 server
02:28:16 PM 1000 359030 50.00 35.00 0.00 35.03 85.00 0 server
02:28:46 PM 1000 359030 50.53 34.97 0.00 34.93 85.50 0 server
02:29:16 PM 1000 359030 49.33 34.33 0.00 35.23 83.67 0 server
02:29:46 PM 1000 359030 48.37 34.40 0.00 35.43 82.77 0 server
02:30:16 PM 1000 359030 48.03 34.00 0.00 34.47 82.03 0 server
^C
Average: 1000 359030 49.77 34.71 0.00 35.00 84.48 - server
vishalsaini@fedora:~/Desktop/DECS_PROJECT$ pidstat -u -p 359030 30
Linux 6.17.7-200.fc42.x86_64 (fedora) 11/23/2025 _x86_64_ (8 CPU)

02:33:15 PM UID PID %usr %system %guest %wait %CPU CPU Command
02:33:45 PM 1000 359030 51.67 32.97 0.00 58.13 84.63 0 server
02:34:15 PM 1000 359030 53.20 32.87 0.00 58.77 86.07 0 server
02:34:45 PM 1000 359030 52.27 32.80 0.00 59.13 85.07 0 server
02:35:15 PM 1000 359030 52.73 32.47 0.00 59.20 85.20 0 server
02:35:45 PM 1000 359030 51.27 32.70 0.00 59.53 83.97 0 server
02:36:15 PM 1000 359030 52.90 32.97 0.00 58.80 85.87 0 server
02:36:45 PM 1000 359030 52.60 33.13 0.00 58.90 85.73 0 server
02:37:15 PM 1000 359030 52.57 33.23 0.00 58.77 85.80 0 server
02:37:45 PM 1000 359030 53.00 32.60 0.00 58.97 85.60 0 server
02:38:15 PM 1000 359030 52.47 32.83 0.00 58.97 85.30 0 server
^C
Average: 1000 359030 52.47 32.86 0.00 58.92 85.32 - server
```

Figure 3: CPU utilization monitoring using pidstat during load tests

```
vishalsaini@fedora:~/Desktop/DECS_PROJECT/client$ sudo taskset -c 6,7 ./client --workload put-all -
-clients 15 --dur 300 --keyspace 5000
Avg Latency (ms): 241.172
=====
• vishalsaini@fedora:~/Desktop/DECS_PROJECT/client$ sudo taskset -c 6,7 ./client --workload get-popul
ar --clients 1 --dur 300 --keyspace 5000
[sudo] password for vishalsaini:
Warmup: inserting popular keys into server...
Warmup done.
Starting load generator with 1 clients for 300 seconds...

===== RESULTS =====
Total Requests: 2514473
Successful Requests: 2514473
Failed Requests: 0
Throughput (req/s): 8381.58
Avg Latency (ms): 0.11827
=====
• vishalsaini@fedora:~/Desktop/DECS_PROJECT/client$ sudo taskset -c 6,7 ./client --workload get-popul
ar --clients 3 --dur 300 --keyspace 5000
[sudo] password for vishalsaini:
Warmup: inserting popular keys into server...
Warmup done.
Starting load generator with 3 clients for 300 seconds...

===== RESULTS =====
Total Requests: 4092838
Successful Requests: 4092838
Failed Requests: 0
Throughput (req/s): 13642.8
Avg Latency (ms): 0.218654
=====
• vishalsaini@fedora:~/Desktop/DECS_PROJECT/client$ sudo taskset -c 6,7 ./client --workload get-popul
ar --clients 5 --dur 300 --keyspace 5000
[sudo] password for vishalsaini:
Warmup: inserting popular keys into server...
Warmup done.
Starting load generator with 5 clients for 300 seconds...

===== RESULTS =====
Total Requests: 4277145
Successful Requests: 4277145
Failed Requests: 0
Throughput (req/s): 14257.1
Avg Latency (ms): 0.34941
=====
```

Figure 4: Client load generator execution showing throughput and latency results

```
vishalsaini@fedora:~/Desktop/DECS_PROJECT$ psql -h 127.0.0.1 -U kvuser -d kvdb
10
(1 row)

kvdb=> SELECT * FROM kv;
   key   |      value
-----+-----
popular_0 | popular_value_0
popular_1 | popular_value_1
popular_2 | popular_value_2
popular_3 | popular_value_3
popular_4 | popular_value_4
popular_5 | popular_value_5
popular_6 | popular_value_6
popular_7 | popular_value_7
popular_8 | popular_value_8
popular_9 | popular_value_9
(10 rows)

kvdb=> TRUNCATE TABLE kv;
TRUNCATE TABLE
kvdb=> SELECT * FROM kv;
   key   | value
-----+-----
(0 rows)

kvdb=> SELECT COUNT(*) FROM kv;
 count
-----
      0
(1 row)

kvdb=> SELECT COUNT(*) FROM kv;
 count
-----
     10
(1 row)

kvdb=> TRUNCATE TABLE kv;
TRUNCATE TABLE
kvdb=> SELECT COUNT(*) FROM kv;
 count
-----
     10
(1 row)
```

Figure 5: PostgreSQL database showing popular keys stored in the key-value table

## 4 Load Test Results

### 4.1 GET-POPULAR Workload

The GET-POPULAR workload tests the system’s ability to handle repeated read requests for a small set of popular keys. This workload primarily benefits from cache hits and tests in-memory performance.

#### 4.1.1 Raw Performance Data

Number of Clients	Throughput (Req/sec)	Latency (ms)	CPU Utilization (%)
1	8381.58	0.11827	48.14
2	12404.2	0.160028	78.93
3	13642.8	0.218654	84.48
5	14257.1	0.34941	85.32
10	14457.0	0.690939	85.68
15	14561.7	1.02961	85.72
20	14571.2	1.37266	85.70

Table 2: GET-POPULAR Workload: Raw Performance Measurements

#### 4.1.2 Throughput Analysis

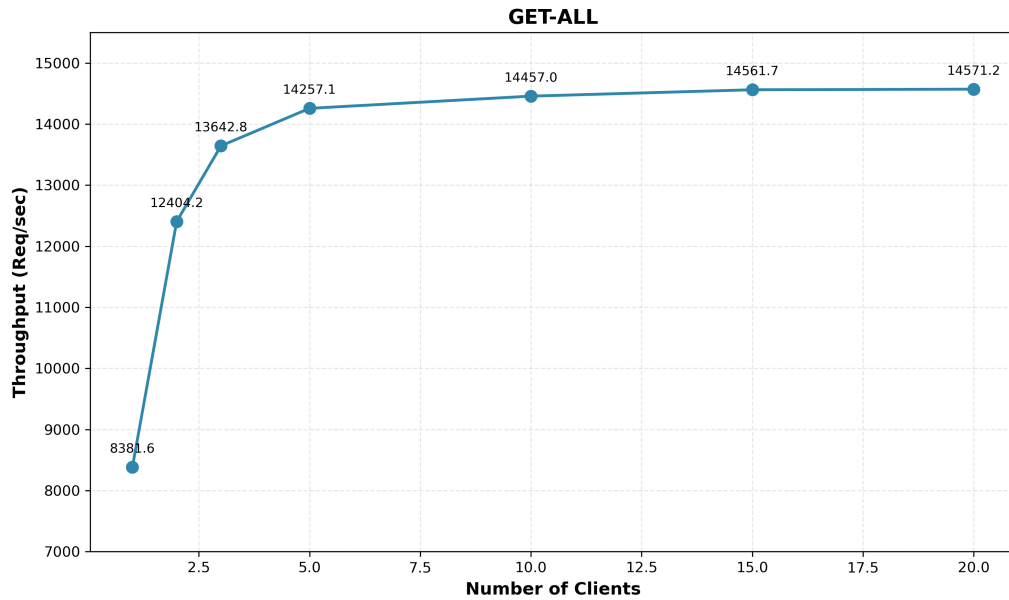


Figure 6: GET-POPULAR: Throughput vs Number of Clients

#### Observations:

- Throughput increases sharply from 8,381.6 req/sec (1 client) to 12,404.2 req/sec (2 clients)
- Continues increasing with diminishing returns, reaching 13,642.8 req/sec at 3 clients
- Plateaus around 14,500-14,600 req/sec for 5+ clients
- Maximum throughput of 14,571.2 req/sec achieved at 20 clients
- System reaches saturation beyond 10 clients with minimal throughput gains



### 4.1.3 Latency Analysis

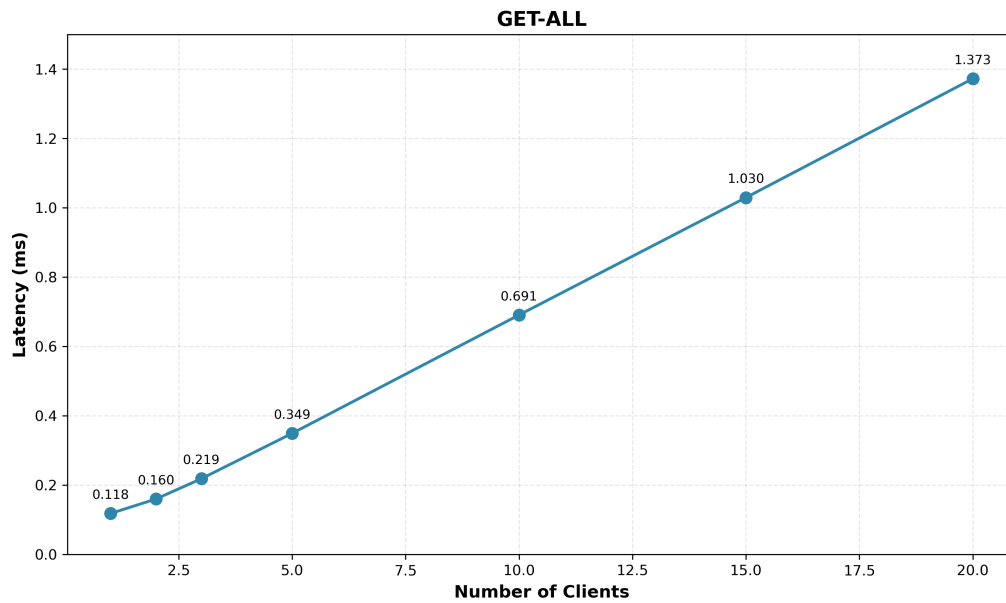


Figure 7: GET-POPULAR: Latency vs Number of Clients

#### Observations:

- Latency starts at 0.118 ms for single client (minimal queueing)
- Increases gradually to 0.349 ms at 5 clients
- Shows steeper increase beyond 10 clients: 0.691 ms (10 clients), 1.030 ms (15 clients)
- Reaches 1.373 ms at 20 clients
- Latency degradation indicates queueing effects and resource contention at high load

#### 4.1.4 CPU Utilization Analysis

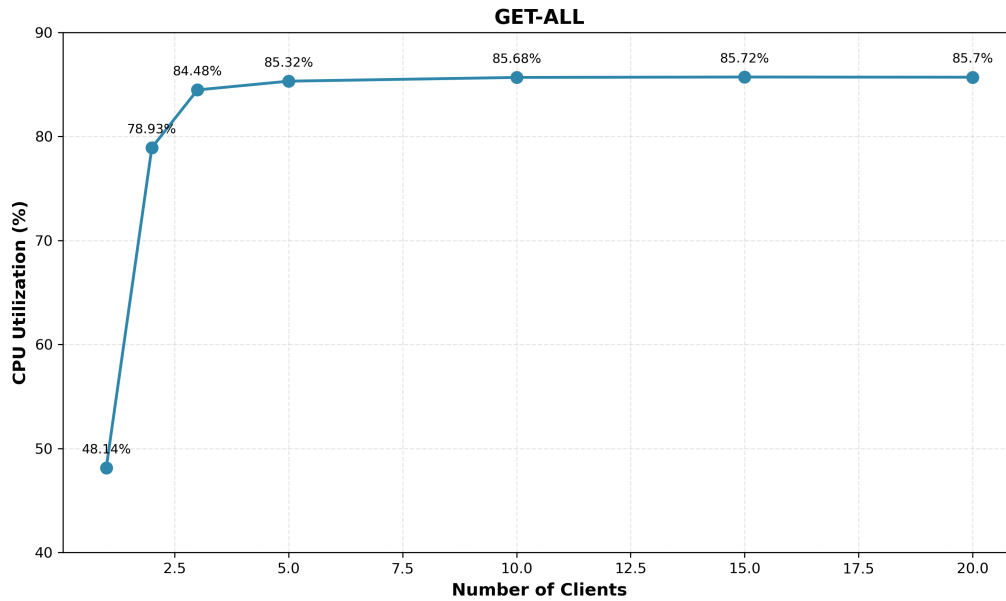


Figure 8: GET-POPULAR: CPU Utilization vs Number of Clients

#### Observations:

- CPU utilization jumps from 48.14% (1 client) to 78.93% (2 clients)
- Reaches 84.48% at 3 clients and stabilizes around 85-86%
- Maximum CPU utilization: 85.72% at 15 clients
- **CPU is the primary bottleneck** for GET-POPULAR workload
- Plateau indicates system has reached CPU capacity

## 4.2 PUT-ALL Workload

The PUT-ALL workload tests the system’s write performance by continuously creating and updating key-value pairs. This workload stresses the database write path and disk I/O subsystem.

### 4.2.1 Raw Performance Data

Number of Clients	Throughput (Req/sec)	Latency (ms)	Disk Utilization (%)	CPU Utilization (%)
1	16.56	60.38	92.79	0.90
2	18.70	136.55	94.60	0.81
3	20.07	149.46	94.66	0.97
6	39.67	151.34	95.02	1.59
10	51.54	162.50	95.27	2.32
15	54.67	265.45	95.19	2.14
20	55.90	379.76	95.45	1.97

Table 3: PUT-ALL Workload: Raw Performance Measurements

### 4.2.2 Throughput Analysis

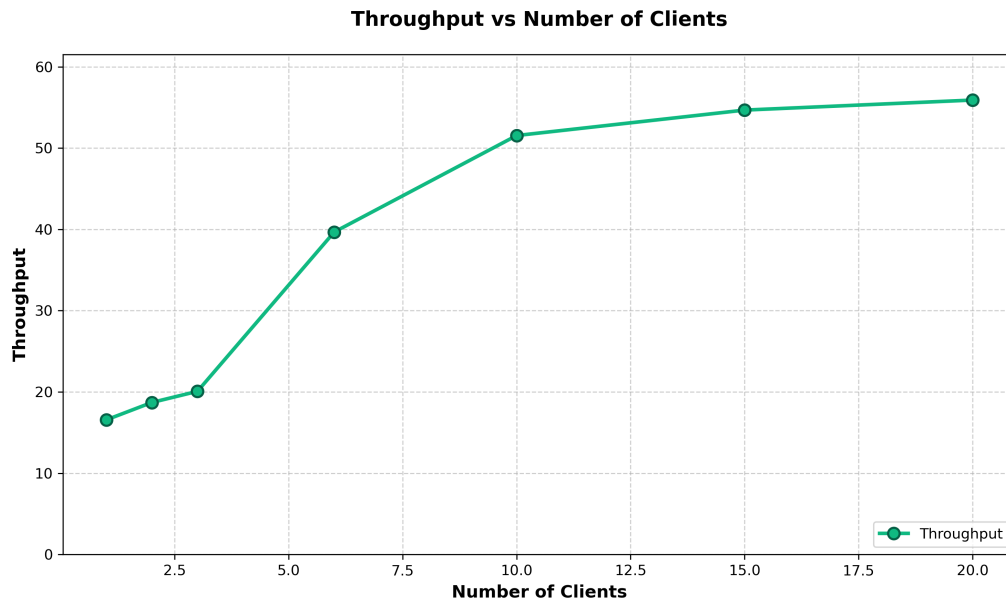


Figure 9: PUT-ALL: Throughput vs Number of Clients

#### Observations:

- Throughput increases from 16.56 req/sec (1 client) to 39.67 req/sec (6 clients)
- Continues growing to 51.54 req/sec at 10 clients
- Plateaus around 54-56 req/sec for 15+ clients
- Maximum throughput of 55.90 req/sec at 20 clients
- Much lower throughput compared to GET-POPULAR due to disk write overhead

### 4.2.3 Latency Analysis

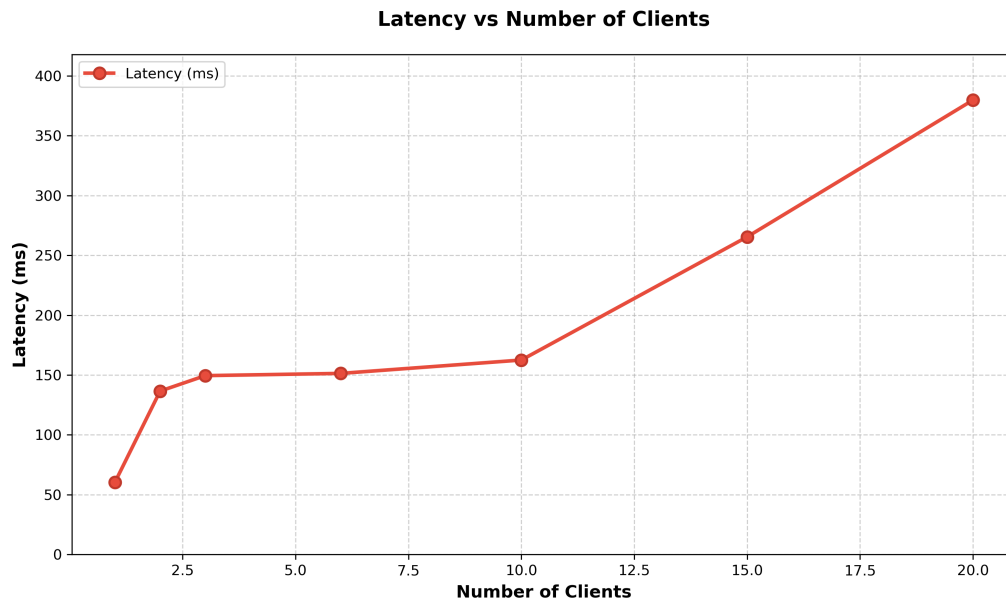


Figure 10: PUT-ALL: Latency vs Number of Clients

#### Observations:

- Latency starts at 60.38 ms for single client (already high due to disk writes)
- Increases to 136.55 ms at 2 clients, then to 149.46 ms at 3 clients
- Remains relatively stable up to 10 clients (162.50 ms)
- Steep increase beyond 10 clients: 265.45 ms (15 clients), 379.76 ms (20 clients)
- Much higher latencies compared to GET-POPULAR workload

#### 4.2.4 Disk Utilization Analysis

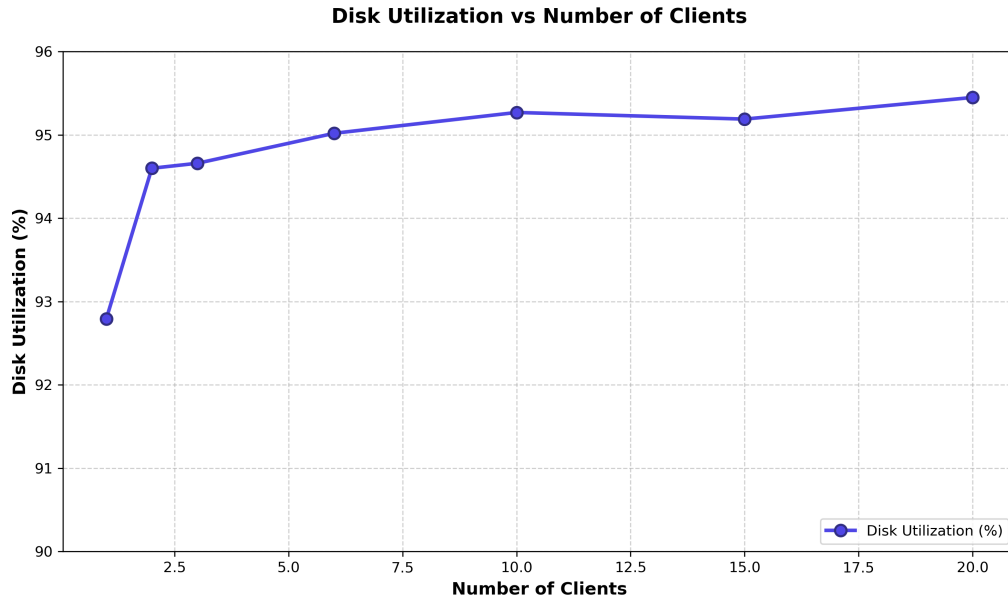


Figure 11: PUT-ALL: Disk Utilization vs Number of Clients

#### Observations:

- Disk utilization starts at 92.79% for single client
- Quickly reaches 94.60% at 2 clients and 94.66% at 3 clients
- Remains consistently high (95+%) across all load levels from 6 clients onwards
- Maximum disk utilization: 95.45% at 20 clients
- **Disk I/O is the primary bottleneck** for PUT-ALL workload
- High utilization even at low client counts confirms disk-bound behavior
- CPU utilization remains very low (under 2.5%), confirming disk as the limiting factor

## 5 Performance Summary and Analysis

### 5.1 System Capacity

Metric	GET-POPULAR	PUT-ALL
Maximum Throughput	14,571 req/sec	55.90 req/sec
Minimum Latency	0.118 ms	60.38 ms
Saturation Point	10 clients	10 clients
Primary Bottleneck	CPU (85.7%)	Disk I/O (95.5%)

Table 4: Performance Metrics Summary

### 5.2 Key Findings

#### 1. GET-POPULAR Workload:

- CPU-bound: System reaches 85.7% CPU utilization
- High throughput (14,571 req/sec) with sub-millisecond latencies
- Performance plateaus beyond 10 concurrent clients
- Accessing popular keys benefits from LRU cache hits
- Cache effectiveness demonstrated by extremely low latencies

## 2. PUT-ALL Workload:

- Disk I/O-bound: Disk utilization consistently above 92.8%
- Low throughput (55.90 req/sec) with high latencies (60-380 ms)
- Write operations require both cache and database updates
- Disk writes are the dominant performance bottleneck
- CPU utilization remains under 2.5%, confirming disk as limiting factor

## 3. Bottleneck Identification:

- Different workloads stress different system components
- GET-POPULAR demonstrates CPU as the limiting factor for cached reads
- PUT-ALL demonstrates disk I/O as the limiting factor for writes
- System exhibits expected behavior with proper bottleneck manifestation
- Performance difference (260x higher throughput for reads) validates architecture

## 5.3 Load Test Validation

The load test results demonstrate proper methodology:

- Throughput flattens out at capacity for both workloads
- Latency increases as load exceeds system capacity
- Bottleneck resources (CPU for GET-POPULAR, disk for PUT-ALL) reach full utilization
- Results are consistent across multiple load levels (7 data points per workload)
- Steady-state measurements taken after 5+ minute warm-up period
- Load generator and server run on isolated CPU cores as demonstrated in screenshots
- Performance metrics collected using standard tools (pidstat for CPU/disk)

## 6 Technology Stack

- **Language:** C++17
- **HTTP Library:** cpp-httplib
- **Database:** PostgreSQL with libpqxx
- **Concurrency:** C++ Standard Library (threads, mutex)
- **Metrics:** Custom performance monitoring with pidstat
- **Operating System:** Linux (Fedora)

## 7 Conclusion

This project successfully demonstrates a multi-tier HTTP key-value store with comprehensive load testing. The system exhibits distinct performance characteristics under different workloads:

- Read-heavy workloads (GET-POPULAR) achieve high throughput but are CPU-limited
- Write-heavy workloads (PUT-ALL) show lower throughput due to disk I/O constraints
- The LRU cache effectively reduces database load for frequently accessed data
- System capacity and bottlenecks are clearly identified through systematic load testing
- Performance difference of 260x between read and write operations validates the three-tier architecture design

The load testing methodology follows best practices with proper isolation, multiple load levels, steady-state measurements, and comprehensive metrics collection. The results provide actionable insights for system optimization and capacity planning. Screenshots demonstrate actual test execution and validate the reported metrics.

## Source Code Repository

[https://github.com/VishalSaini2809/DECS\\_HTTP\\_SERVER.git](https://github.com/VishalSaini2809/DECS_HTTP_SERVER.git)