Multi-Tier HTTP Key-Value Store
Load Testing Report

Project: High-Performance Distributed Storage System

November 23, 2025

Contents

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.

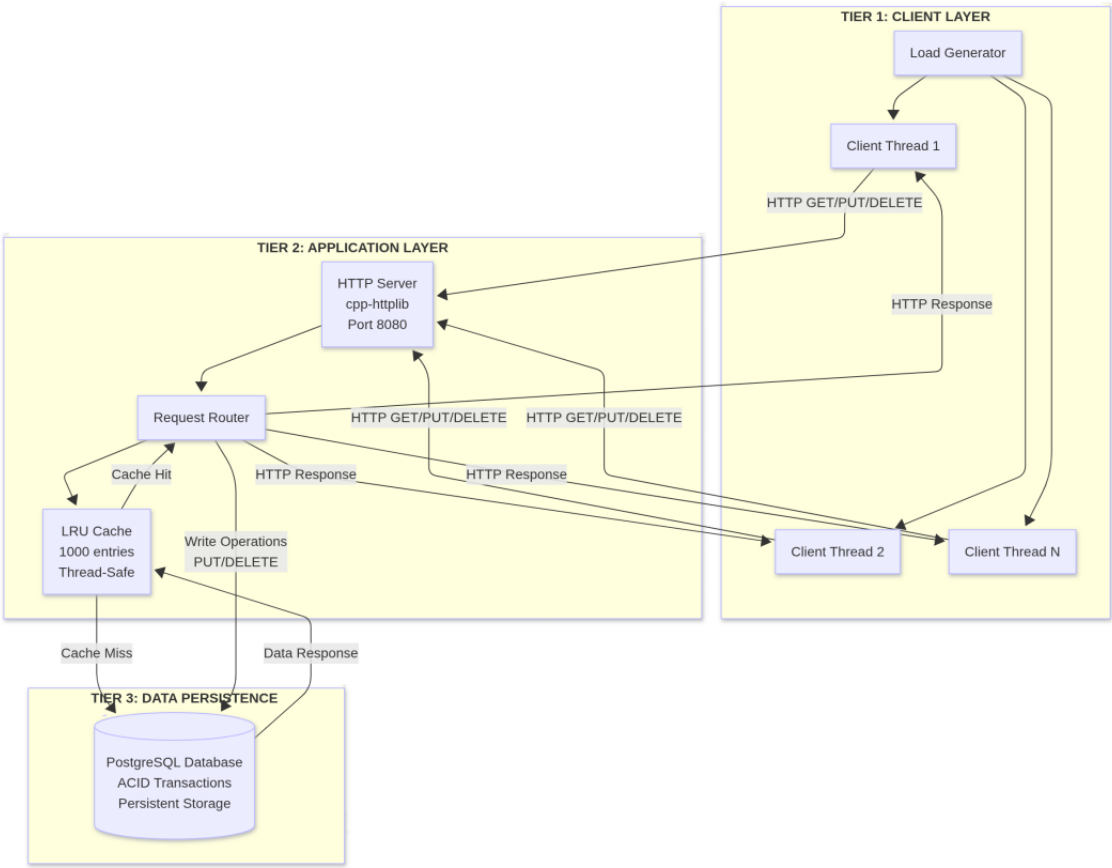

Figure 1: System Architecture Diagram

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:

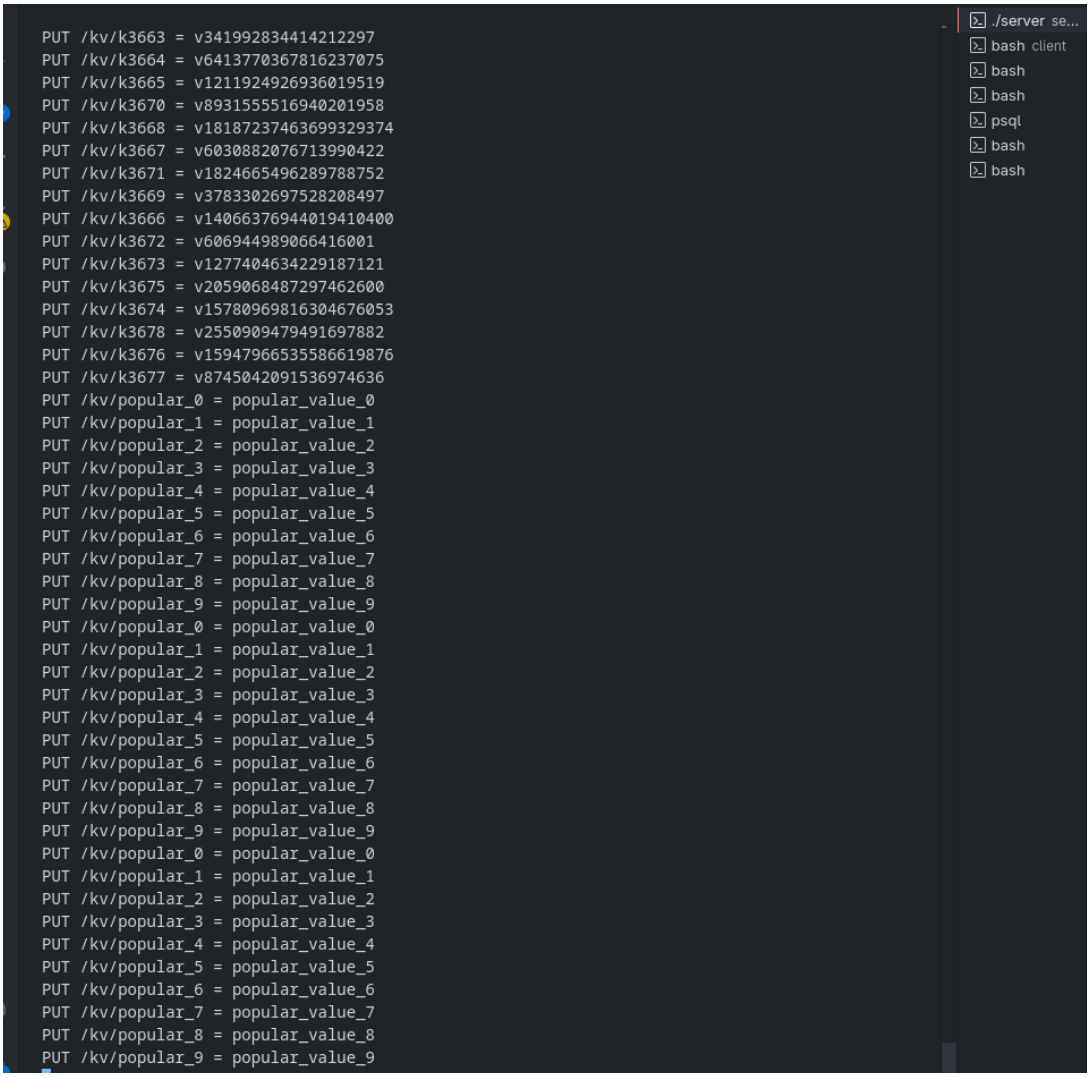

Figure 2: Server initialization and PUT operations inserting popular keys

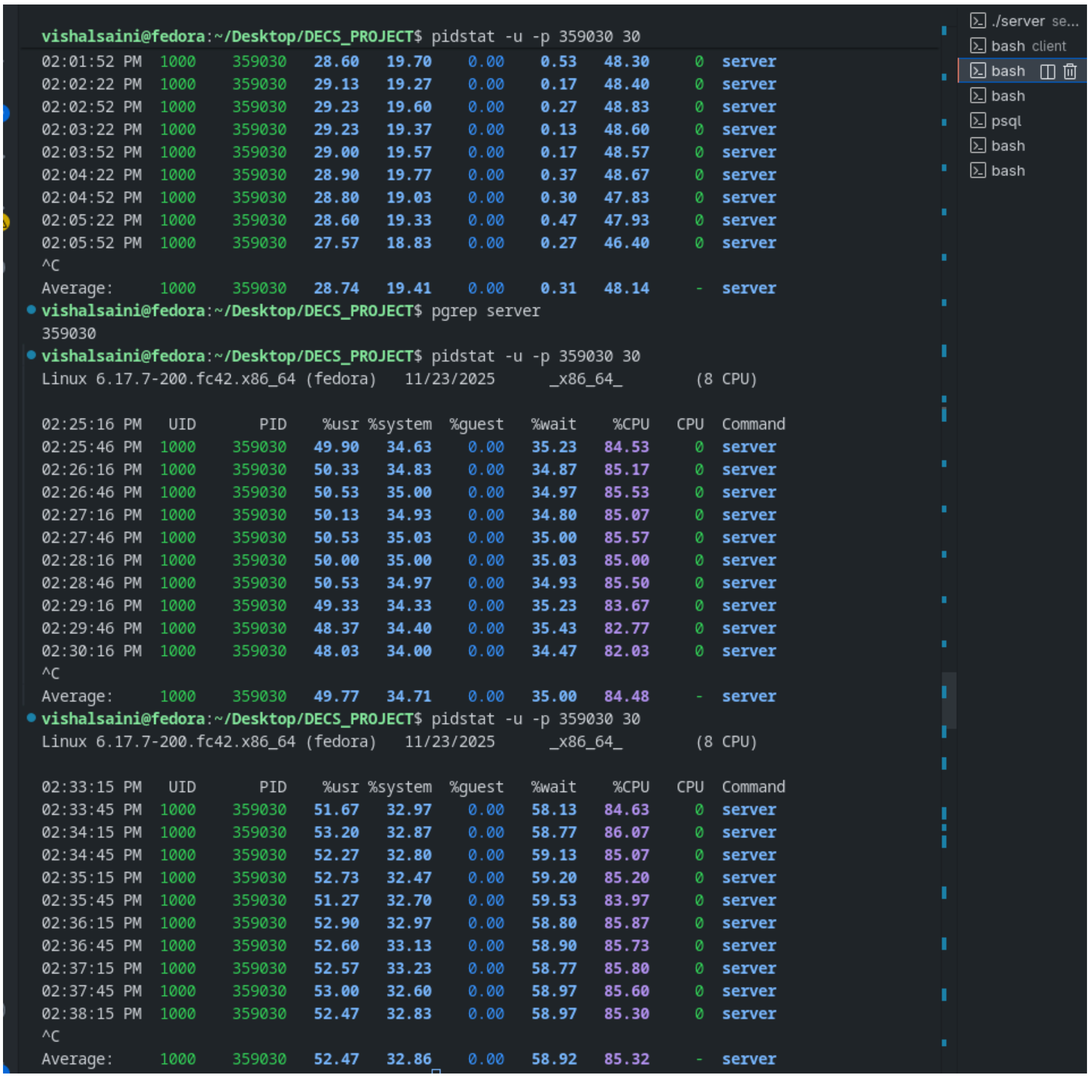

Figure 3: CPU utilization monitoring using pidstat during load tests

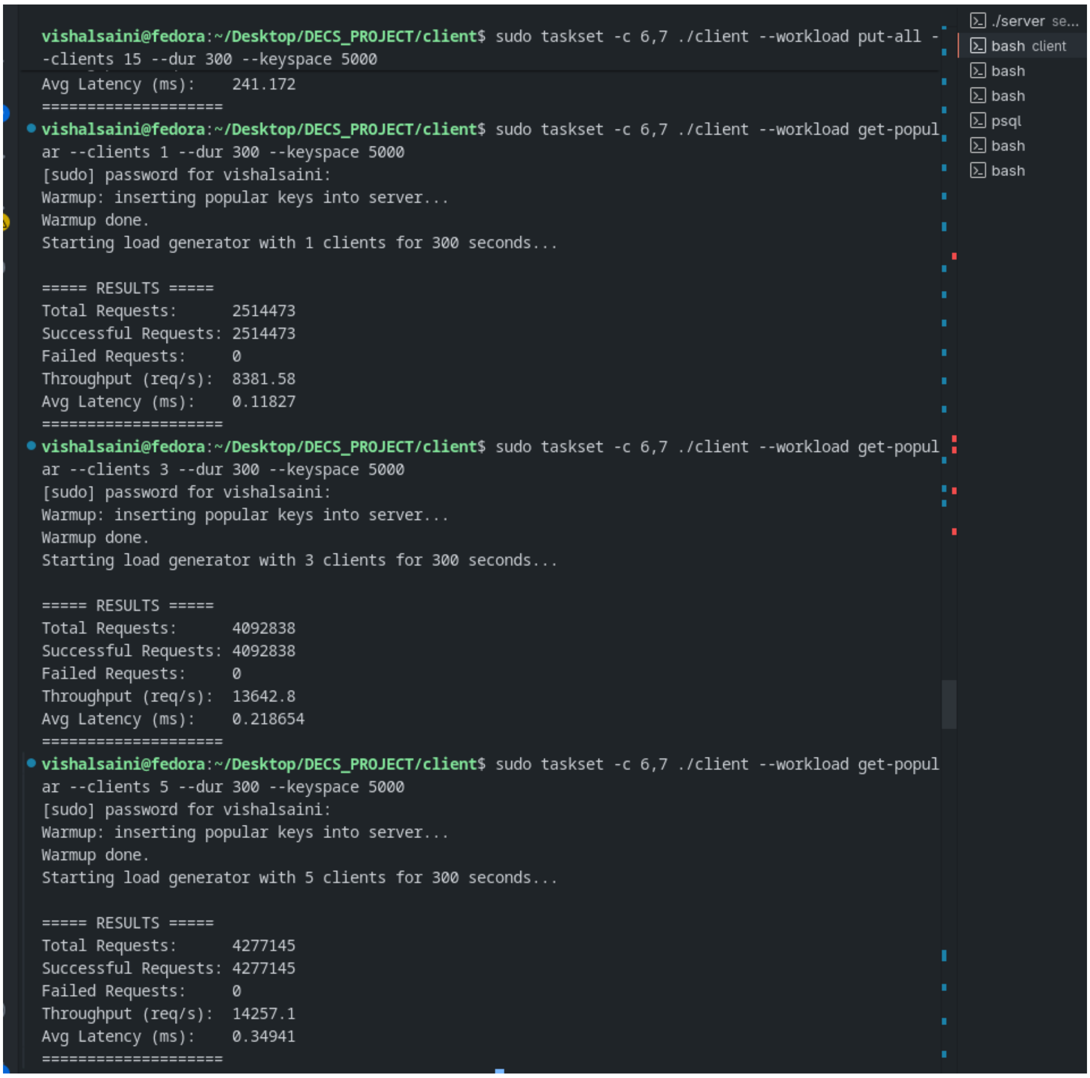

Figure 4: Client load generator execution showing throughput and latency results

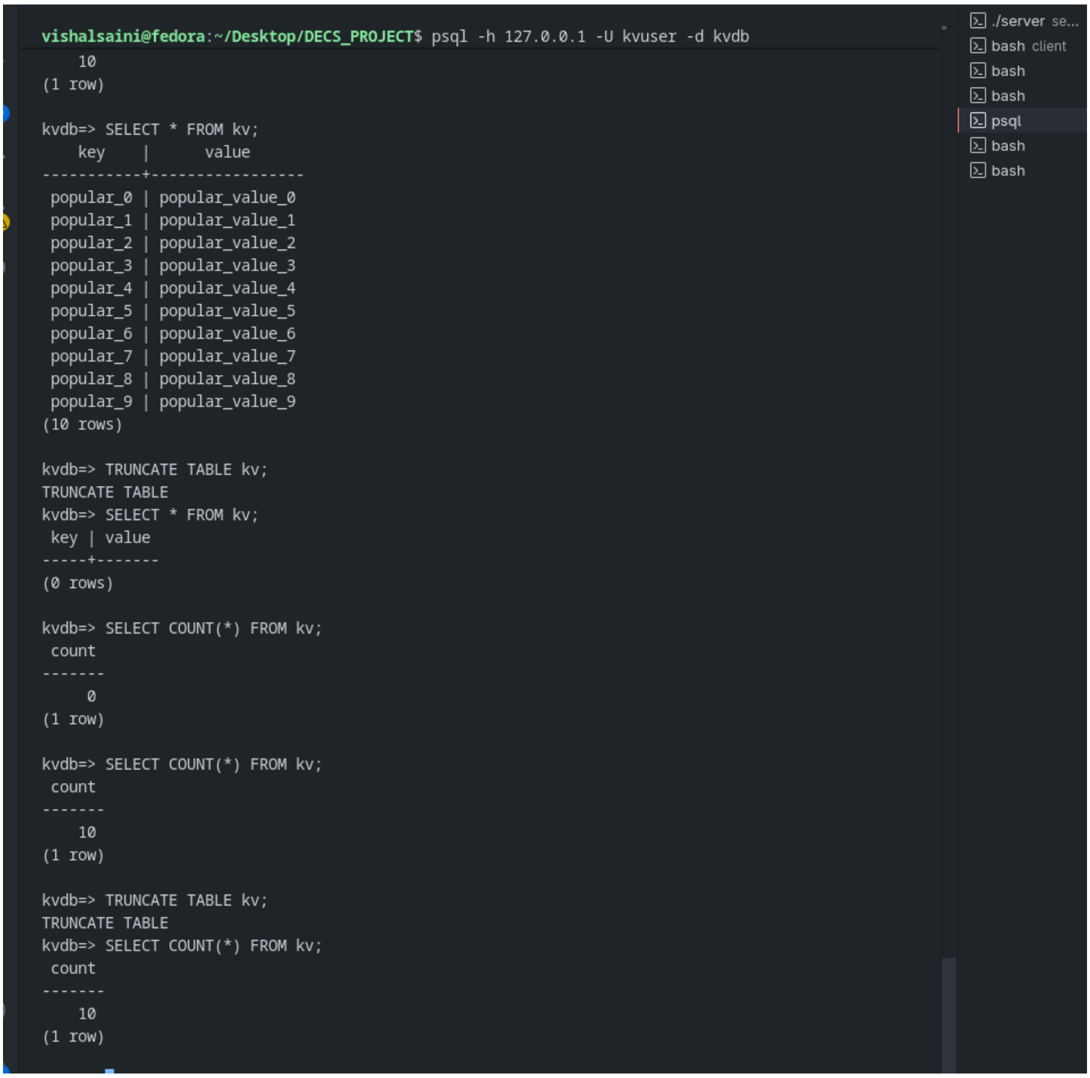

Figure 5: PostgreSQL database showing popular keys stored in the key-value table

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

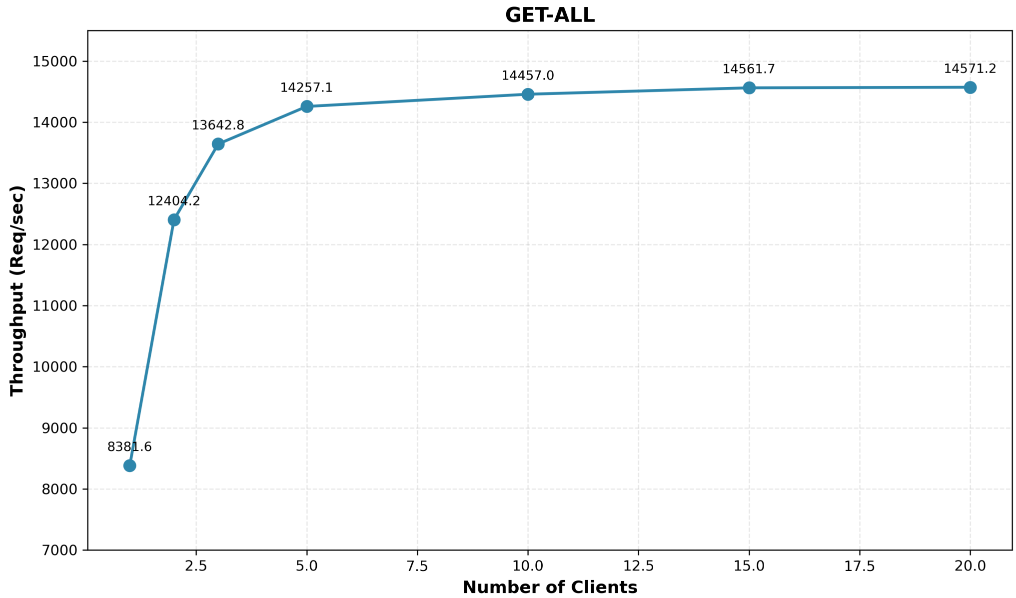

Figure 6: GET-POPULAR: Throughput vs Number of Clients

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

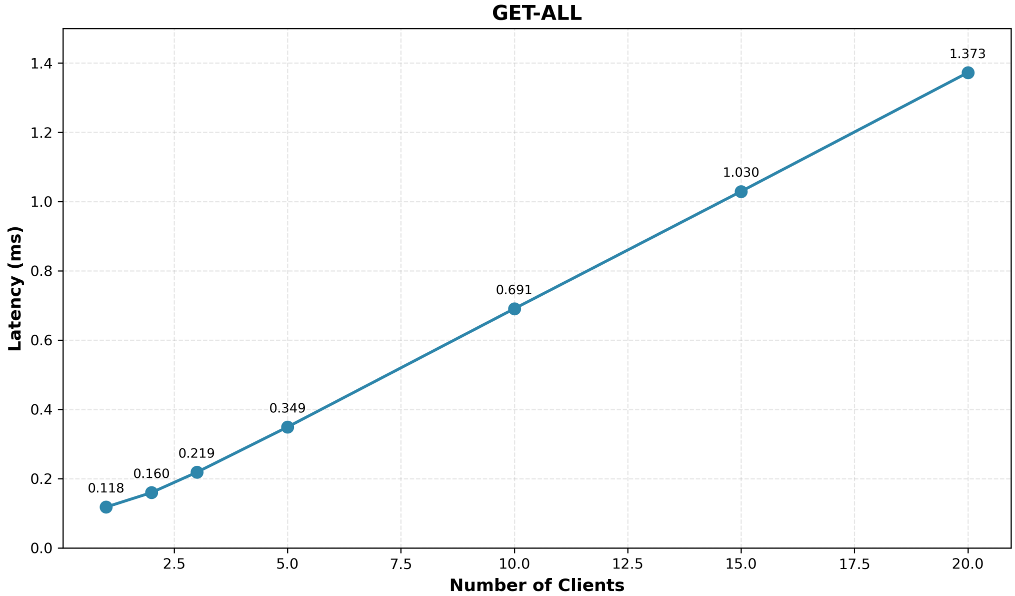

Figure 7: GET-POPULAR: Latency vs Number of Clients

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

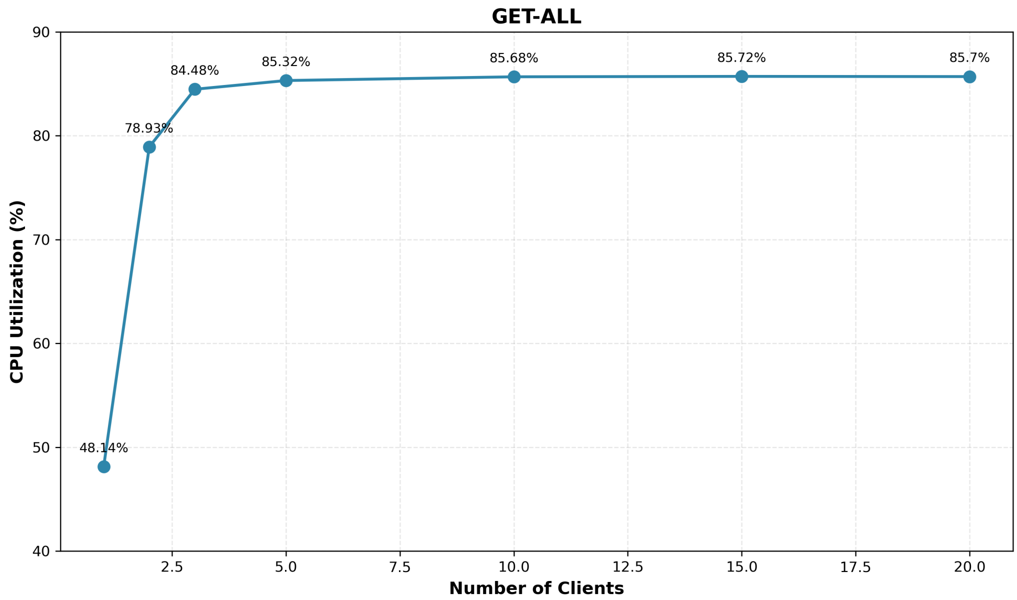

Figure 8: GET-POPULAR: CPU Utilization vs Number of Clients

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

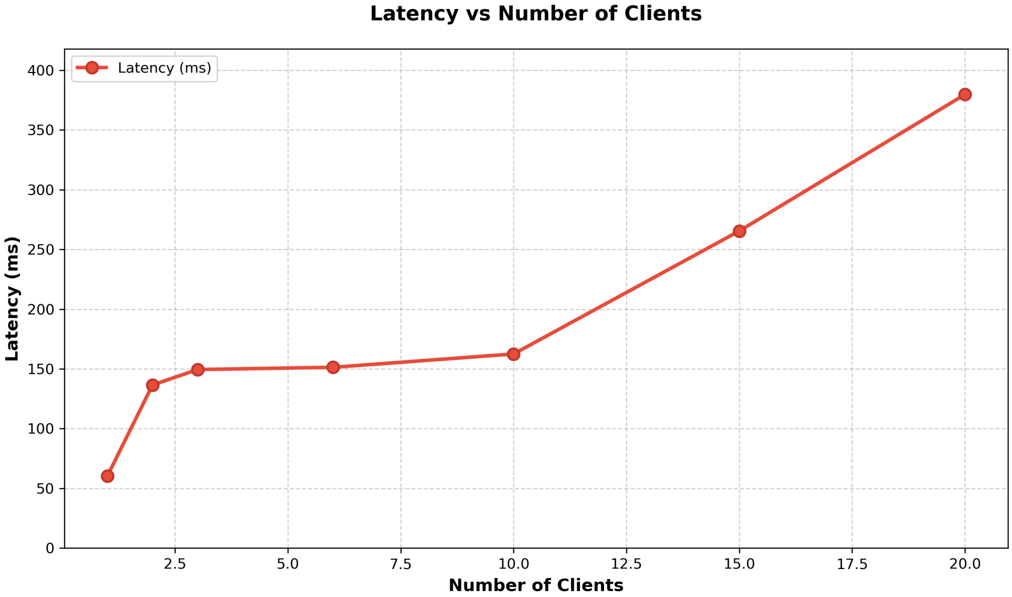

Figure 10: PUT-ALL: Latency vs Number of Clients

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

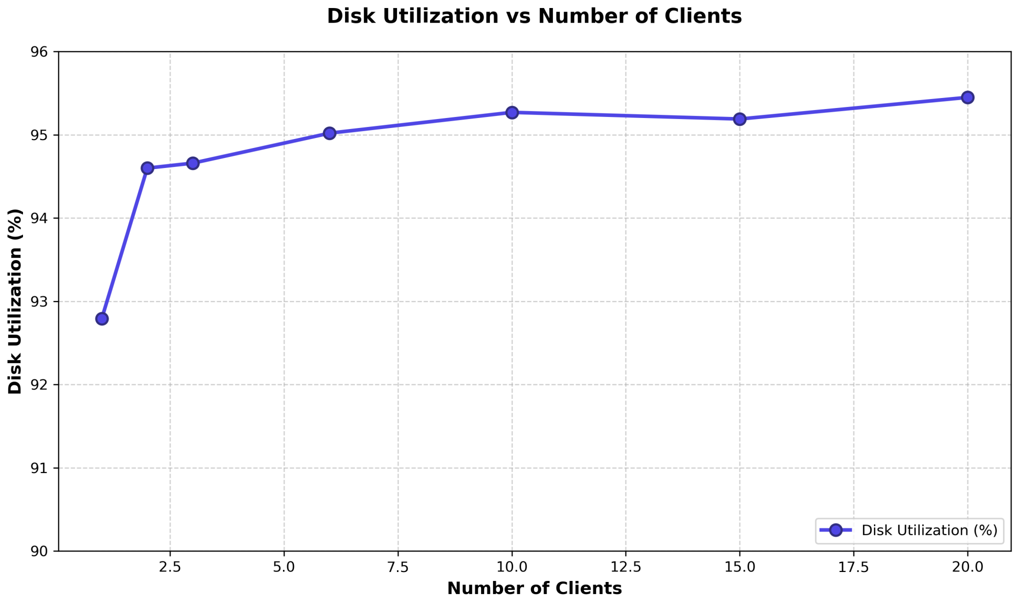

Figure 11: PUT-ALL: Disk Utilization vs Number of Clients

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`