Phase 2: Load Testing and Performance Evaluation

System Bottleneck Analysis

Department of Computer Engineering
Course Project Submission

Author: Monil Manish Desai

November 22, 2025

Contents

Chapter 1

Introduction

The reliability and scalability of distributed key-value stores depend heavily on their behavior under stress. This report details the Phase 2 performance evaluation and stress testing of the HTTP-based Key-Value store implemented in Phase 1.

The primary objective is to empirically identify specific hardware bottlenecks—Server CPU saturation, Network Bandwidth limits, and Disk I/O throttling—by systematically varying the load level. By pushing the system to its breaking point, we aim to characterize the "knee" of the performance curve, where throughput plateaus and latency begins to degrade exponentially.

Per the project requirements, we utilize a load generator to measure the following Key Performance Indicators (KPIs):

1. **Throughput:** The rate of successful requests processed per second (req/s). This metric helps identify the maximum capacity of the system.
2. **Latency:** The end-to-end time taken for a request to be served. We analyze how latency scales with concurrency to detect queue buildup.
3. **Resource Utilization:** The saturation level of the specific hardware constraint (CPU %, Bandwidth MB/s, or Disk IO %). Correlating utilization with throughput allows us to pinpoint the exact hardware component limiting performance.

Code Availability

The complete source code, benchmarking scripts, and raw data logs for this project are available on GitHub:

`https://github.com/monil2003/cs744-project`

Chapter 2

System Architecture Overview

The system architecture is identical to the Phase 1 implementation, designed for high concurrency and data persistence. The key components are summarized below.

2.1 Core Components

- **HTTP Server**:
  A multithreaded server built using `cpp-httplib`, configured with a thread pool of up to 200 worker threads. This ensures that blocking I/O operations (such as database writes) do not stall the acceptance of new incoming connections.

- **LRU Cache**:
  An in-memory Least Recently Used (LRU) cache holds the 10 most recently accessed keys. This layer absorbs repeated read traffic, significantly reducing the load on the database for "hot" keys.

- **PostgreSQL Backend**:
  The persistence layer consists of three distinct database instances isolated on specific CPU cores to prevent resource contention:

  - **DB1:** Handles Partition 1 (Pinned to CPU 0–1).
  - **DB2:** Handles Partition 2 (Pinned to CPU 2–3).
  - **DB3:** Acts as a logical replica (Pinned to CPU 15).

- **Replication**:
  Asynchronous logical replication is configured to ensure DB3 stays synchronized with updates from DB1 and DB2, providing data redundancy.

- **Key Partitioning**:
  A client-side partitioning logic routes write requests to either DB1 or DB2 based on a deterministic hash of the key. This sharding strategy allows the system to utilize multiple database cores simultaneously.

2.2 Architecture Diagram

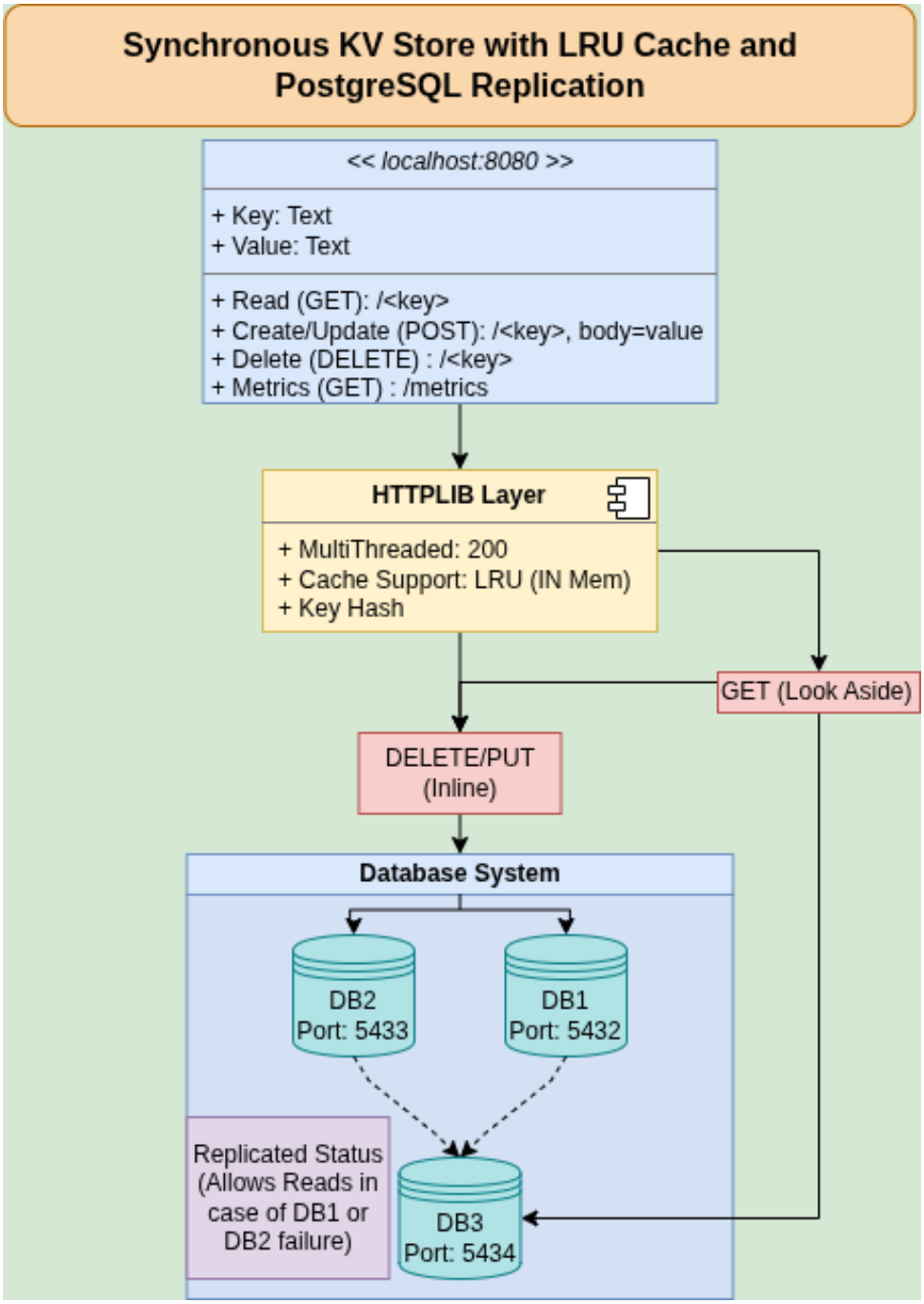

Figure 2.1: High-level architecture showing the interaction between the Multithreaded Server, LRU Cache, Sharded Primary DBs (DB1/DB2), and the Replica DB (DB3).

Chapter 3

Load Generator & Setup

3.1 Load Generator Details

We utilized a custom **Closed-Loop Load Generator** written in C++. Unlike open-loop generators that flood the server regardless of response capacity, this generator emulates real-world user behavior.

- **Closed-Loop Model:** The generator emulates a fixed number of concurrent users (threads). Each worker thread operates synchronously: it issues a request, waits for the full response, and then immediately issues the next request. This mechanism creates natural backpressure; if the server slows down, the client request rate automatically adjusts. This allows us to measure the system's maximum capacity without causing artificial queue explosions.
- **Usage Syntax:**

  ```
  ./loader <num_threads> <duration_second> <w_type>
  ```

- **Supported Workloads:**
  - `get_popular1`: Read-heavy (10 hot keys) → tests CPU/Cache.
  - `put_all`: Write-heavy (random keys) → tests I/O Bandwidth/Disk.
  - `put_all_big`: Large Payloads (20KB) → tests Memory/DRAM bandwidth.
  - `put_key_1`: Writes to a single key → tests Database Row-Level Locking.

3.2 Hardware Topology and Isolation

The test environment utilizes a hybrid CPU architecture comprising Performance (P) and Efficiency (E) cores. To prevent interference between the Client, Server, and Databases, we utilized strict **CPU Affinity (Pinning)**.

3.2.1 CPU Core Specifications

Core ID	Logical Threads	Core Type
0	0 – 1	P-core (High Performance)
2	2 – 3	P-core (High Performance)
4	4 – 5	P-core (High Performance)
6	6 – 7	P-core (High Performance)
8	8 – 9	P-core (High Performance)
10	10 – 11	P-core (High Performance)
12	12	E-core (Background Tasks)
13	13	E-core (Background Tasks)
14	14	E-core (Background Tasks)
15	15	E-core (Background Tasks)

Table 3.1: Host Machine CPU Topology

3.2.2 Database Configuration

Three PostgreSQL instances were deployed with a **500 MB memory limit** each to force disk I/O usage once the working set exceeds memory.

- **DB 1:** Pinned to CPU 0–1 (P-Core)
- **DB 2:** Pinned to CPU 2–3 (P-Core)
- **DB 3 (Replica):** Pinned to CPU 15 (E-Core)

3.3 Experimental Scenarios & Constraints

3.3.1 Scenario 1: CPU Bound (Get Popular)

This scenario isolates CPU processing power by using a read-heavy workload with a small cacheable dataset.

- **Workload Type:** `get_popular1` (10 keys)
- **Server Config:** 200 Threads, Cache Size 10.
- **Server Affinity:** CPU 4–5 (P-Cores)
- **Client Affinity:** CPU 6–14 (Mixed P/E Cores)
- **Goal:** saturate the specific cores assigned to the server to observe the throughput ceiling.

3.3.2 Scenario 2: Bandwidth Bound (Put All)

To simulate a network-constrained environment, we utilized Linux Control Groups (`cgroup v2`) to throttle Block I/O bandwidth.

- **Workload Type:** `put_all`
- **Server Config:** 200 Threads, Cache Size 10.
- **Server Affinity:** CPU 4–11 (P-Cores)
- **Client Affinity:** CPU 12–14 (E-Cores)
- **Applied Constraint:** $\approx$ 10.5 MB/s Limit.

Throttling Command:

```
echo "7:0 rbps=10485760 wbps=10485760" | sudo tee /sys/fs/cgroup/system.slice/io.max
```

Verification: A ‘dd‘ test confirmed a write speed of 9.7 MB/s after applying this constraint.

3.3.3 Scenario 3: Disk Utilization Bound (Put All - Max)

This scenario removes artificial throttling to test the physical limits of the storage device.

- **Workload Type:** `put_all`
- **Core Configuration:** Same as Scenario 2.
- **Applied Constraint:** Limits Removed (`max`).

Un-throttling Command:

```
echo "7:0 rbps=max wbps=max" | sudo tee /sys/fs/cgroup/system.slice/io.max
```

Verification: A ‘dd‘ test confirmed a write speed of 3.7 GB/s after removing the limit.

3.3.4 Additional Scenarios

- **Row-Level Locking:** Uses workload `put_key_1`. All threads write to the same key, forcing serialization at the Database level.
- **Memory/DRAM Bottleneck:** Uses workload `put_all_big`. Writes 20KB values to stress memory bandwidth during data transfer between the socket and the database buffers.

Chapter 4

Experimental Results

4.1 Workload 1: CPU Saturation (Get Popular)

In this read-heavy workload, the bottleneck is the Server CPU. We stress-tested the system from 1 thread up to 64 threads to observe behavior well past the saturation point.

4.1.1 Data Table

Threads	Throughput (req/s)	Latency (ms)	CPU Util	Notes
T-1	1219.89	1.26	70.0%	
T-2	1694.97	1.29	80.0%	
T-4	1897.22	1.66	92.7%	
T-8	1937.38	3.71	98.0%	Saturation Point
T-12	1941.73	5.75	98.4%	
T-16	1940.85	7.81	99.0%	
T-20	1941.48	9.86	99.2%	
T-32	1961.30	15.86	100.0%	
T-64	**1948.11**	**32.35**	**100.0%**	**Queue Build-up**

Table 4.1: Load 1 Results. Note that T-64 is excluded from graphs to maintain visual scale.

4.1.2 Observation on T-64

While included in the table, T-64 is excluded from the graphs below. At T-64, throughput remains flat compared to T-32, but latency doubles (15ms $\rightarrow$ 32ms). This confirms that once the CPU is fully saturated, adding more concurrency only adds to the queue length without improving performance.

4.1.3 Performance Analysis

Throughput plateaus at ≈ 1960 req/s. As shown in Figure 4.2, this directly correlates with the CPU hitting 99-100% utilization at T-8.

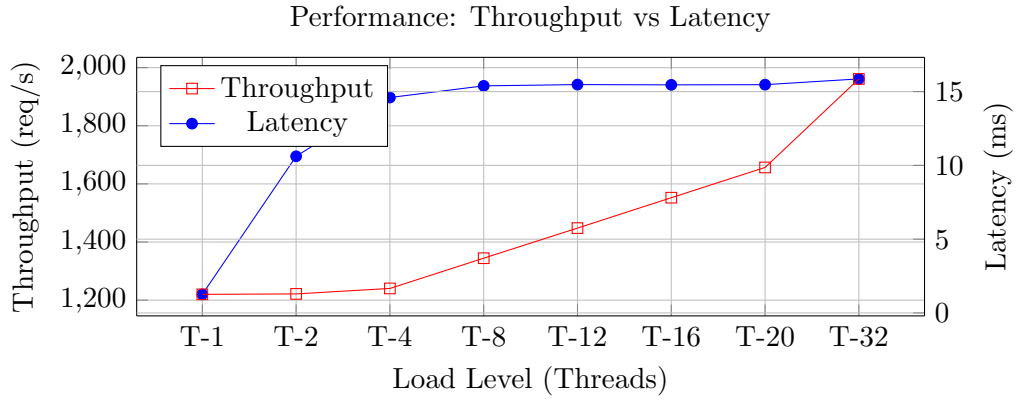

Figure 4.1: Performance: Latency spikes as system reaches capacity.

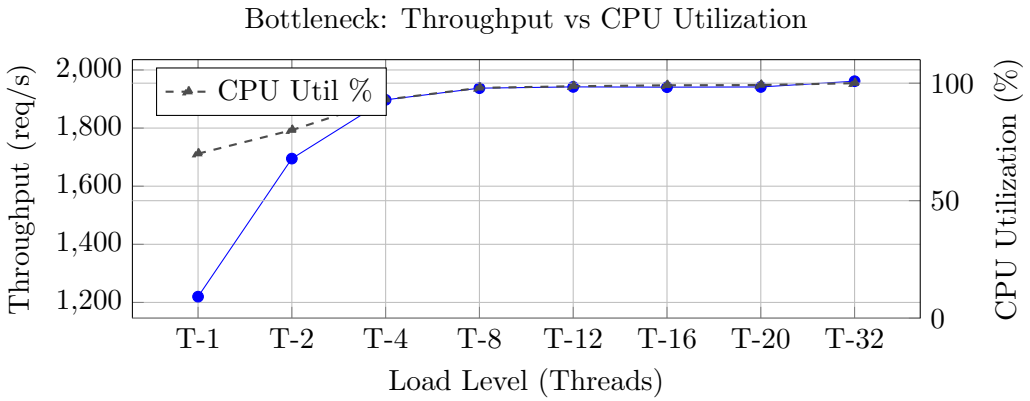

Figure 4.2: Bottleneck: Throughput stops growing exactly when CPU hits 100%.

4.2 Workload 2: Bandwidth Saturation (Put All)

Here, we limited the IO bandwidth to 10.5 MB/s using `cgroups` to simulate a restricted network environment.

4.2.1 Data Table

Threads	Thrpt (req/s)	Latency (ms)	IO Speed	IO Util	Status
T-1	533.03	1.13	4.87 mbps	30.3%	
T-2	991.23	1.28	9.33 mbps	50.0%	
T-3	1038.27	2.22	9.95 mbps	65.0%	
T-4	1045.64	3.32	10.20 mbps	55.0%	BW Cap
T-5	1039.70	4.30	10.50 mbps	50.0%	BW Cap
T-6	1057.90	5.16	10.50 mbps	40.0%	
T-8	1048.94	7.13	10.50 mbps	37.0%	
T-10	**1212.1**	**7.75**	**10.50 mbps**	**38.0%**	**System Failure**

Table 4.2: Load 2 Results. T-10 resulted in a critical failure.

4.2.2 Analysis of T-10 Failure

At T-10, the benchmark reported a throughput of **1212.1 req/s** and a latency of **7.75 ms** with a ∼23-25% error rate. While these numbers nominally appear "better" than the T-8 results, they represent a critical system collapse. This data point is excluded from the graphs to avoid misinterpretation, as the metrics are artificially inflated by failure.

The specific causes are:

1. **Queue Timeout & Saturation:** The 23-25% error rate indicates that roughly 1 in 4 requests failed entirely (connection reset or timeout). The network buffer, strictly throttled to 10.5 MB/s, became full and physically could not accept new packets.

2. **Artificial Throughput Increase (Request Dumping):** The apparent increase in throughput (from 1048 to 1212 req/s) is deceptive. This phenomenon occurs because the server or load balancer is "fast-failing" requests. Rejecting a request takes significantly less time and bandwidth than processing a full payload. Therefore, the system processed a higher volume of *failures*, creating a mirage of improved performance when, in fact, it was dumping requests to survive the backlog.

4.2.3 Performance Analysis

As seen in Figure 4.4, the system hits the "Bandwidth Wall" at T-4. The IO Speed flatlines at 10.5 MB/s. Consequently, throughput cannot exceed $\approx$ 1050 req/s, causing latency to rise linearly.

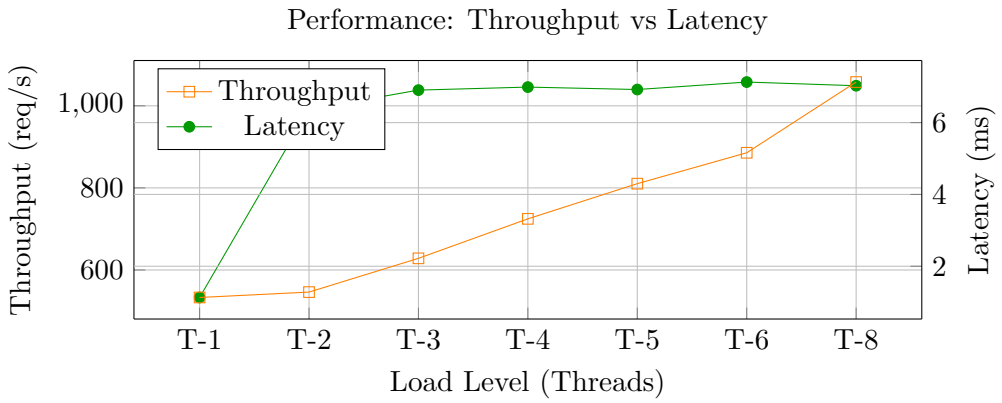

Figure 4.3: Performance: Throughput is capped by the network pipe.

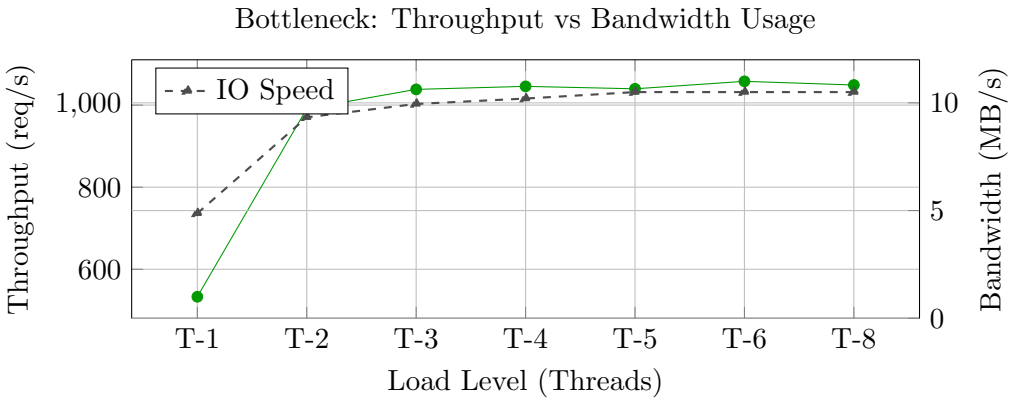

Figure 4.4: Bottleneck: Bandwidth usage hits the 10.5 MB/s limit at T-5.

4.3 Workload 3: Disk Utilization (Put All)

This workload tests the physical limits of the disk without artificial bandwidth throttling.

4.3.1 Data Table

Threads	Thrpt (req/s)	Latency (ms)	Disk Util	Notes
T-1	360.37	2.35	30.0%	
T-2	932.59	1.42	60.0%	
T-4	1571.28	2.10	93.0%	Near Saturation
T-8	2556.27	2.63	99.1%	Max Throughput
T-16	**2442.65**	**6.04**	**98.7%**	**Thrashing**

Table 4.3: Load 3 Results. Throughput degrades after T-8.

4.3.2 Performance Analysis

Figure 4.6 shows Disk Utilization reaching 99.1% at T-8. Unlike CPU or Bandwidth saturation where throughput simply flattens, Disk saturation causes **Thrashing**. At T-16, throughput drops (from 2556 to 2442 req/s) and latency doubles. This occurs because the OS and disk controller spend more cycles managing the I/O queue than actually writing data.

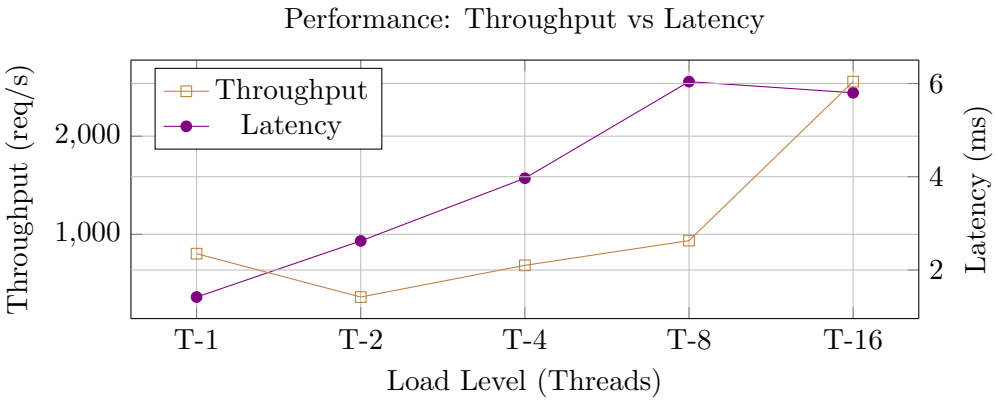

Figure 4.5: Performance: Thrashing observed after T-8 (Throughput drops).

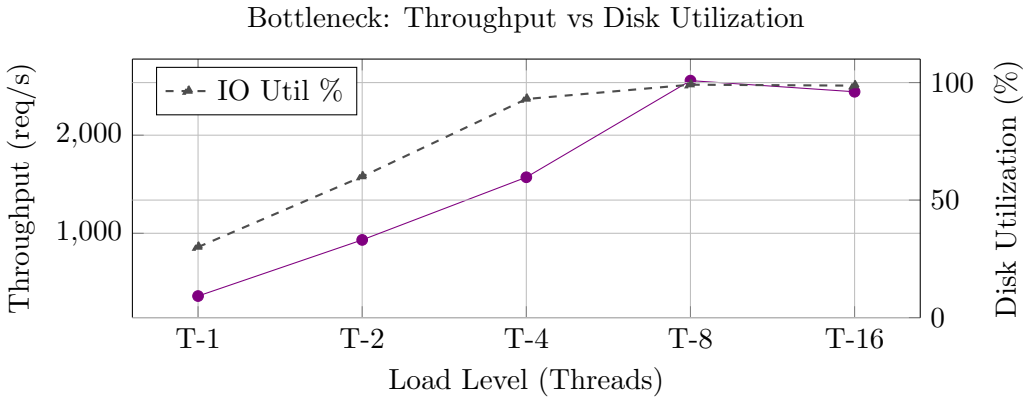

Figure 4.6: Bottleneck: Disk reaches ~100% utilization at T-8.

Chapter 5

Conclusion

This Phase 2 evaluation successfully characterized the performance envelope of the HTTP-based Key-Value store under various resource-constrained scenarios. By systematically isolating the CPU, Network, and Disk subsystems, we confirmed that the system's scalability is strictly bound by the scarcest hardware resource.

The experimental results validate fundamental queueing theory principles and demonstrate three distinct bottleneck signatures:

5.1 Summary of Bottleneck Behaviors

- **CPU Saturation (Compute Bound):** The "Get Popular" workload demonstrated a classic compute-bound profile. Throughput exhibited linear growth proportional to the thread count until the CPU utilization reached 100%. Beyond this saturation point, throughput plateaued completely, while latency transitioned from stable to exponential growth. This confirms that once processor cycles are exhausted, additional concurrency only serves to lengthen the request queue without adding processing capacity.
- **Bandwidth Saturation (Network Bound):** The throttled write tests revealed the "hard wall" nature of network limits. Unlike CPU saturation, where performance degrades gracefully via latency, bandwidth limits create an immediate throughput ceiling (observed at $\approx$ 10.5 MB/s). The critical failure observed at T-10 highlights a dangerous edge case: when buffers overflow, the system may report misleadingly high throughput due to "fast-failing" requests (connection resets), masking the true service unavailability.
- **Disk Saturation (I/O Bound):** The full-utilization write tests exposed the physical limitations of persistent storage. We observed the phenomenon of **thrashing**, where increasing concurrency beyond the saturation point (T-8) actually caused a *regression* in throughput (from 2556 down to 2442 req/s). This indicates that at high contention levels, the overhead of managing the I/O queue and disk controller context switching outweighs the benefit of parallel writes.

5.2 Final Remarks

The results emphasize the importance of identifying the "knee" of the performance curve. For this specific architecture:

1. **Read Scalability** is limited by core count; moving to a machine with more P-Cores would linearly improve read performance.
2. **Write Scalability** is currently bound by physical disk IOPS. Future optimizations, such as batching writes or implementing an asynchronous Write-Ahead Log (WAL), would be required to push beyond the current limits.

Overall, the load generator and system architecture performed predictably, providing a clear roadmap for future capacity planning and optimization.