HTTP-based KV server with in-mem Cache

This project implements a simple HTTP key-value server with a custom in-memory cache, a PostgreSQL database, and a multithreaded closed-loop load generator for performance experiments. The server supports create/read/delete operations and evaluates throughput and latency under configurable load level and workload.
The main goal of this project is to exercise different bottlenecks (CPU vs DB), measure throughput and response time under configurable client load, and evaluate system behavior (CPU, cache hit rate, IO).

Github repo: *https://github.com/s1ddh3sh/CS744-project*

1 System Architecture

The system consists of four main components:

- **Server:** Multi-threaded HTTP server (`server.cpp`) built with `httplib`, exposing the `/create`, `/get`, `/delete`, `/preload`, `/clear`, and `/cache-stats` endpoints(Figure 1). Manages coordination between cache and database access.
- **In-memory Cache:** Thread-safe, bounded in-memory key-value store (`kv-store.c` / `kv-store.h`) implementing a first-come, first-served (FCFS) eviction policy.
- **Database Layer:** Database connectivity layer (`db.c` / `db.h`) built on `libpq` for interfacing with a PostgreSQL database.
- **Load Generator:** Closed-loop client program (`loadgen.cpp`) that generates deterministic workloads with configurable load levels and measures throughput, response times, and performance bottlenecks in the server, CPU, and disk I/O.

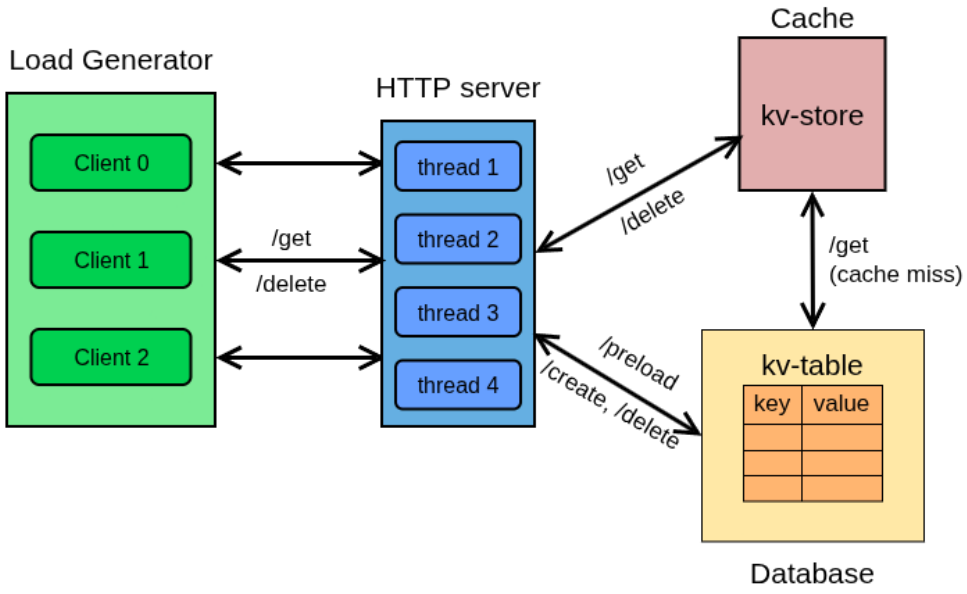

Figure 1: System architecture

2 Load Generator

- **Closed-loop model:** Each worker thread generates requests independently; it waits for the response before generating the next request.
- **Supported workloads implemented:**
  - `get_popular`: Random GET requests concentrated on a small hotset of 1000 popular keys → high CPU utilization (CPU-bound, CPU-heavy).
  - `put_all`: 50% PUT + 50% DELETE requests that exercise DB writes (DB-bound, persistence heavy).
- **Metrics collected at load generator:**
  - Successful / failed requests, cache hits/hit rate.
  - Total latency (used to compute average response time)
  - Throughput = successful requests / test duration (sec)
- **Load generator functions:**
  - `do_post_create()`: Sends a POST request (`/create`) to create key-value pair, measures request latency, and sets success flag.

- `do_get()`: Sends a GET request (`/get`) to fetch value for a given key.
- `do_delete()`: Sends a DELETE (`/delete`) request to delete a kv-pair from DB.
- `prepopulate_data()`: Clears server state (`/clear`) and preloads keys (`/preload`) required by read workloads, so workload start from a known state.
- `fetch_cache_stats()`: Calls server endpoint (`/cache-stats`) to retrieve cache metrics (hits,misses, hit rate) for reporting.
- `client_thread()`: Core per-client loop, uses (`/preload`) seeds deterministic PRNG, opens persistent `httplib::Client`, generates workload-specific requests (closed-loop), records latency and success/failure into shared atomic `GlobalStats`.
- `main(argc, argv)`: Parses command-line parameters (threads, duration, workload), calls `prepopulate_data`, spawns N `client_thread` threads, joins them after duration, aggregates totals, computes throughput and average response time, and prints results.

3 Load Test Setup

- **Software:**
  - Server: C++ / `httplib`, (`server.cpp`)
  - In-memory cache: C (`kv-store.c / kv-store.h`)
  - Database layer: `libpq` database connector
  - Load generator: C++ (`loadgen.cpp`)
- **Database:** PostgreSQL running locally (ensure connection info is correct in `db.c`).
- **Experiment parameters (configurable in script):**
  - **Core affinity:** Server runs on core 1, loadgen runs on cores 2-7.
  - **Threads:** Number of concurrent client threads (range: 1-16)
  - **Duration:** Length of each run in seconds (300 s)
  - **Workloads:** Run each workload (`get_popular`, `put_all`) separately
- Refer README.md for individual execution of server and loadgen using taskset.

- **Monitoring & outputs:**
  - Load generator produces results into /results, later converted to `results.csv` (columns: Threads, Throughput, Response time, Cache hit rate, CPU util, DB util) for plotting.
  - System monitoring captured with `mpstat` (server core utilization) and `iostat` (disk IO utilization)
  - Results stored under `results/{workload}/{threads}` for later plotting

4 Load Test Results

Following are the test results for the given workloads with relevant plots deriving the observations and inference.

4.1 `get_popular` Workload

Table 1: Load Test Results: `get_popular`

Threads	Throughput	Response Time	Cache hits	CPU util	DB util
1	7248.76	0.136424	99.954015	25.99	1.32
2	12139.4	0.162961	99.972541	51.96	1.40
4	16887.3	0.234606	99.980202	82.83	1.50
5	17711.3	0.279985	99.981104	89.02	1.24
6	18343.9	0.324659	99.981756	91.64	1.19
8	18573.1	0.427467	99.981898	92.09	1.28
10	18731.8	0.530498	99.982055	92.28	1.23
12	18716.5	0.637187	99.982035	92.46	1.25
14	18754.8	0.742602	99.981994	92.68	1.18
16	18718.3	0.850319	99.981842	92.82	1.15

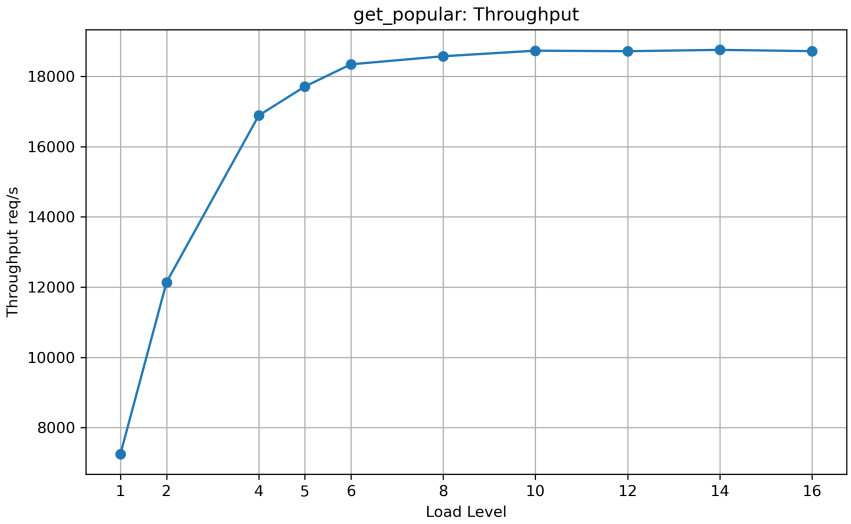

Figure 2: Throughput vs Load Level for `get_popular`

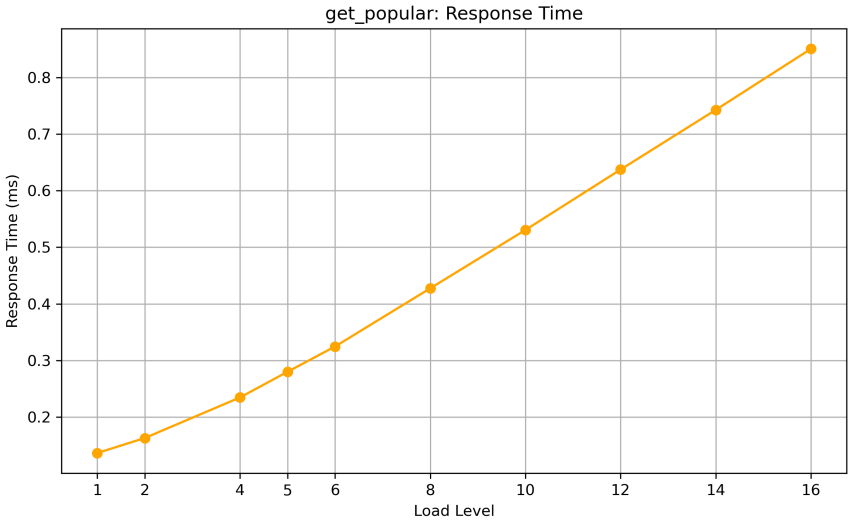

Figure 3: Response Time vs Load Level for `get_popular`

Observation:

- Server cores reach full saturation at roughly the same load level, producing similar curves in both the throughput and CPU utilization graphs[2][4. Cache hit rates remain extremely high (99.98%; Table 1).
- Throughput scales with load level initially but saturates due to server bottleneck.

- Average response time grows gradually beyond 8–10 threads due to thread contention and scheduling overhead.
- Disk I/O(DB util) remain very low(Table 1), confirming minimal database access.

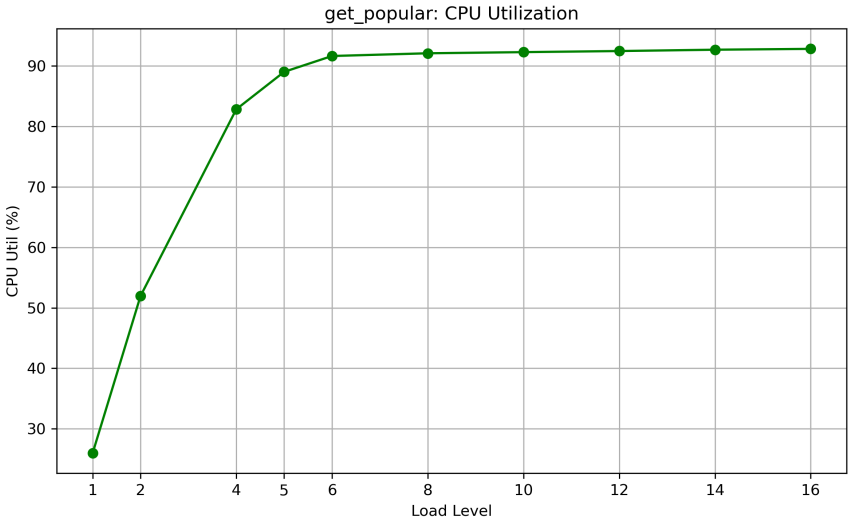

Figure 4: CPU Utilization for `get_popular`

Inference:

- Most requests are served from in-memory cache, making this workload CPU bound[4] due to parsing, serialization, and mutex operations.
- Overall, the system demonstrates effective caching: high throughput, low latency, predictable performance *i.e server saturation from load-level=8 and onwards*, which corresponds to the plateau observed in throughput.
- The high server CPU utilization (90–93%) indicates that the primary bottleneck lies in server-side processing rather than disk I/O.

4.2 `put_all` Workload

Table 2: Load Test Results: `put_all`

Threads	Throughput	Response Time	Cache hits	CPU util	DB util
1	150.251	3.49711	0	11.13	74.97
2	163.152	7.16411	0	6.36	84.74
4	160.396	15.5995	0	6.23	85.21
5	160.066	19.8891	0	6.06	85.04
6	157.135	24.6374	0	6.10	85.44
8	157.092	33.1279	0	6.01	85.49
10	159.157	40.5184	0	6.14	85.03
12	157.826	49.6997	0	6.20	85.31
14	159.144	57.6408	0	6.02	85.26
16	159.480	65.4264	0	6.23	85.19

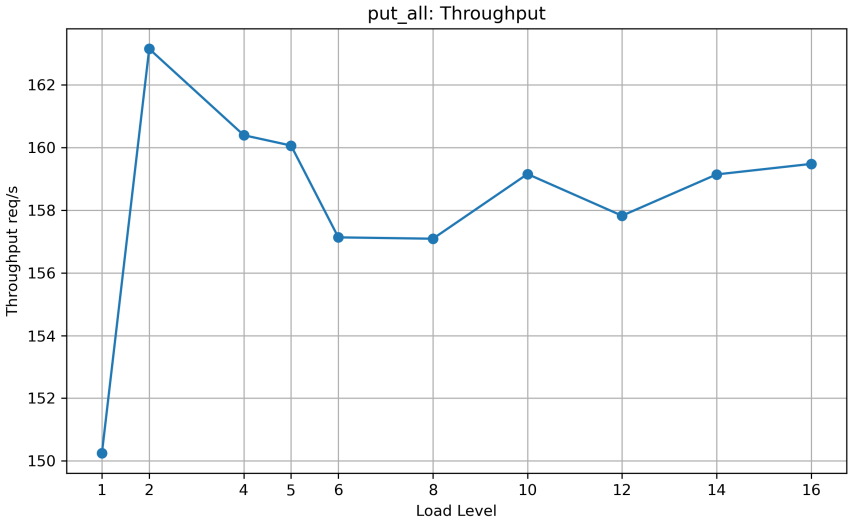

Figure 5: Throughput vs Load Level for `put_all` workload

Observation:

- The workload is DB-bound during write operations: database CPU and I/O activity remain high, while server-side CPU usage stays low, around 6–11% (Table 2).
- The Disk I/O saturates early, even at low load levels and stays consistently high throughout the test, peaking at roughly 86%[7].

- Throughput shows an early rise from load level 1 to 2, but then stabilizes once the disk becomes saturated. Beyond this point (load-level ≥ 5), throughput stays within a narrow band of 150–160 req/s [5].
- Response times scale almost linearly with load, rising from 4 ms at low load to over 65 ms at the highest load.

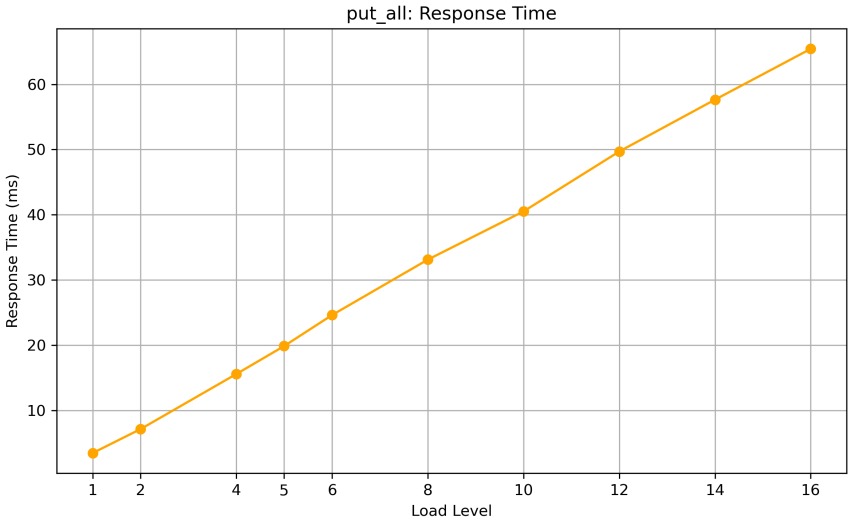

Figure 6: Response Time vs Load Level for `put_all` workload

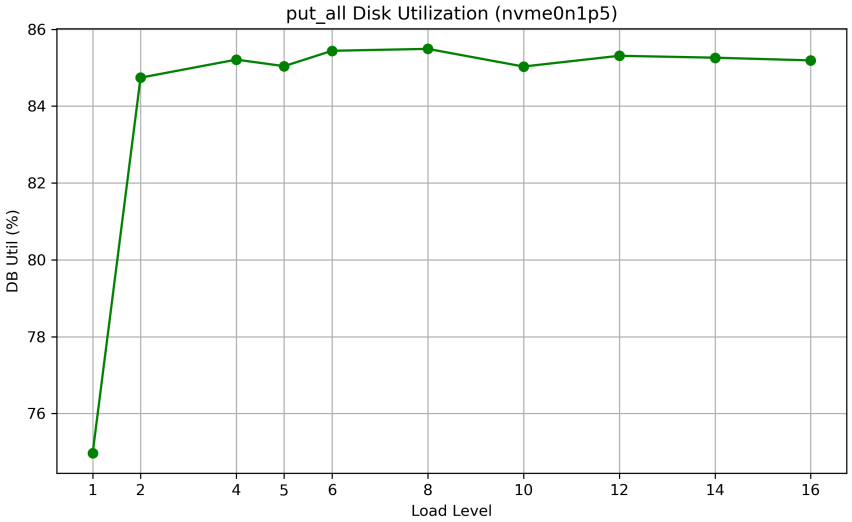

Figure 7: Disk utilization for `put_all` workload

Inference:

- The results indicate that database write operations are the dominant bottleneck during put_all load testing.
- The throughput plateau suggests the system is hitting a resource ceiling, most likely at the database layer. Even as the load increases, the system cannot process writes faster.
- Steady increase in response time[6] suggests the latency is not due to the increasing load level, but due to the piling of requests as DB writes are slower.
- Early DB utilization shows it is working near its resource limit very early.Additional load increases latency rather than resource usage.