REPORT OF DECS PROJECT PHASE-II

M.Harshith-25m0834

1. GitHub Repository: https://github.com/harshith2302/KV-Server

2. Brief Description:

The Key-Value (KV) Store implemented as a C++ client-server application. supports **Create, Read, and Delete (CRUD)** operations over **HTTP**, using:

- **PostgreSQL** as the persistent database
- **In-memory LRU cache** for fast access
- **cpp-httplib** library for HTTP communication

The **Server** is a multithreaded C++ application that listens for HTTP requests. It manages data persistence by storing key-value pairs in a **PostgreSQL** database. To accelerate read operations, it uses an in-memory **LRU (Least Recently Used) Cache**. All database connections are managed by a custom **Connection Pool** to minimize latency.

The **Client** is a simple C++ command-line tool that provides a menu to interact with the server's API, allowing users to create, read, and delete key-value pairs.

3. System Architecture:

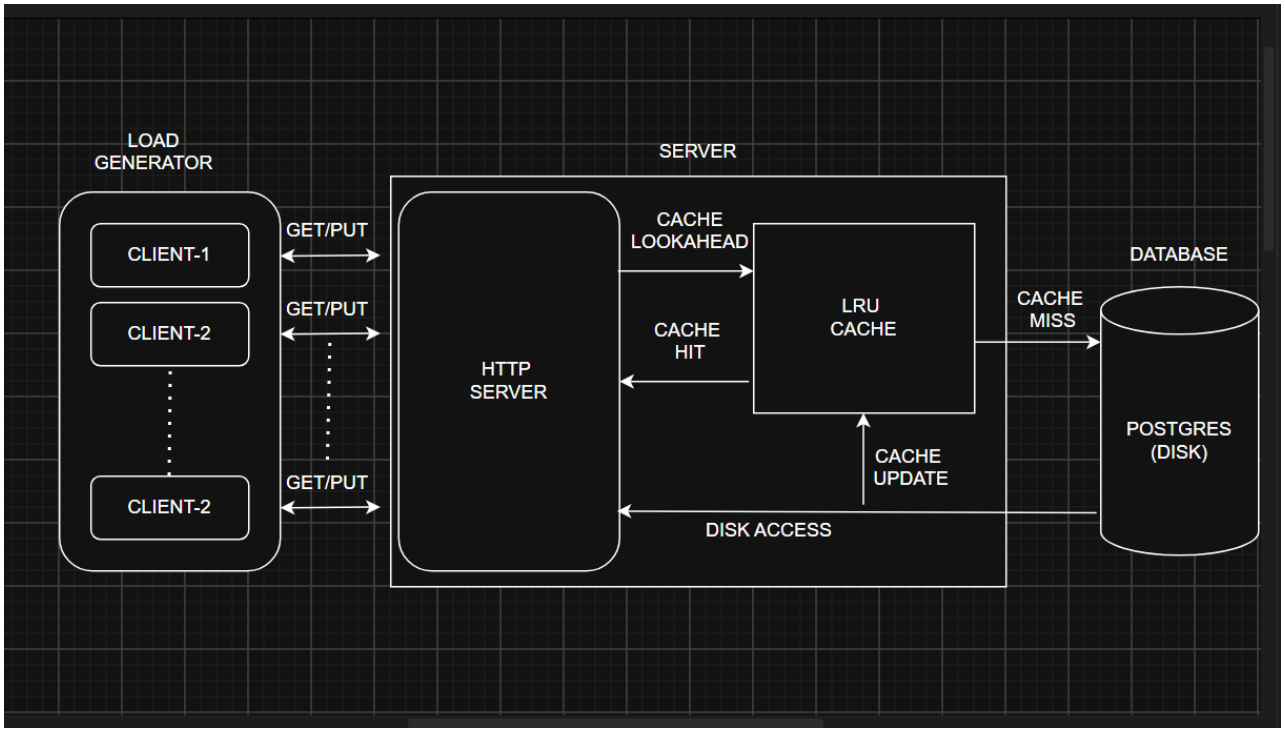


Fig : Overall flow of my system architecture

4. Explanation of Operations:

4.1 Read Request (/read):

1. The server first checks the LRUCache for the int key.
2. **Cache Hit:** If the key is found, the value is returned immediately. This is very fast.
3. **Cache Miss:** If the key is *not* in the cache, the server gets a connection from the DBPool, queries the PostgreSQL database, and (if found) stores the result in the cache before returning it to the client.

4.2 Write Request (/create):

1. The server gets a connection from the DBPool.
2. It executes an INSERT ... ON CONFLICT DO UPDATE command, which writes the data to the PostgreSQL database.
3. If the database write is successful, the server then **updates the LRUCache** with the new value. This ensures the cache stays consistent with the database.

4.3 Delete Request (/delete):

1. The server gets a connection from the DBPool.
2. It executes a DELETE command in the PostgreSQL database.
3. If the database delete is successful, the server **removes the key from the LRUCache**. This is crucial for preventing clients from reading "stale" data that no longer exists.

5. Load Testing Scripts using k6

MY project includes 4 main workload scripts

5.1 GET-only Workload

Sending key in query params

Used to test:

- Cache hit ratio
- Continuously reads the data form the server

5.2 PUT-only Workload

Sending key-value as query params

Used to test:

- Write throughput
- Continuously add the data into the server
- Cache update behavior

5.3 get-popular Workload

Sending key in query params

Used to test:

- How well the popular elements from the server we can retrive
- Cache invalidation impact

5.4 Mixed workload

Performs all hybrid kind data analysis

Used to test:

- When both kind of data request got asked how well it will run
- Cache behaviour

6. Bash Automation for High-Concurrency Experiments

Wrote automated benchmarking engine is written in Bash.

The script automatically:

Iterates over a list of VUs (1 → 1000)

Runs all workload types: GET, PUT, GET_POPULAR, MIXED

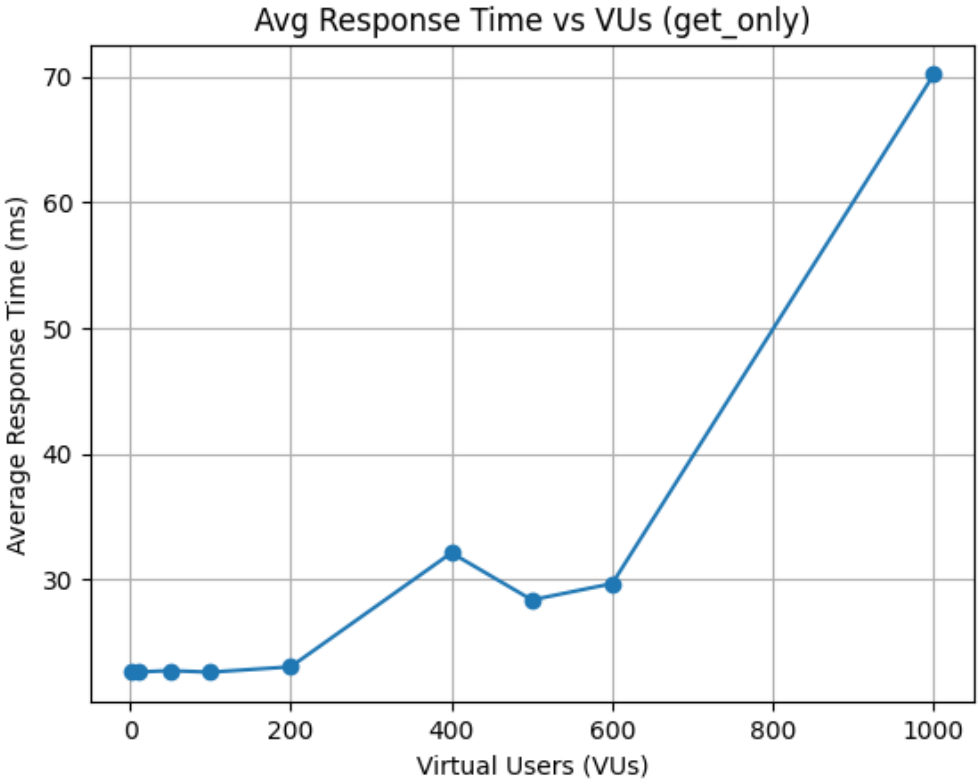
Extracts k6 output (throughput, avg, p50, p90, p95, p99)

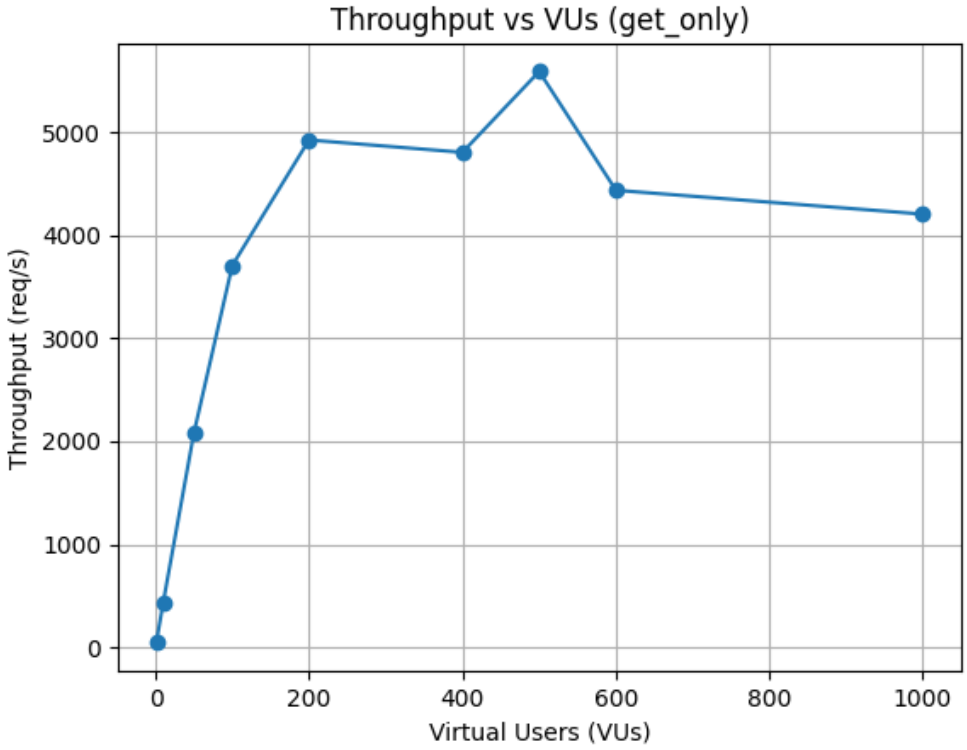
Writes results into separate CSV files

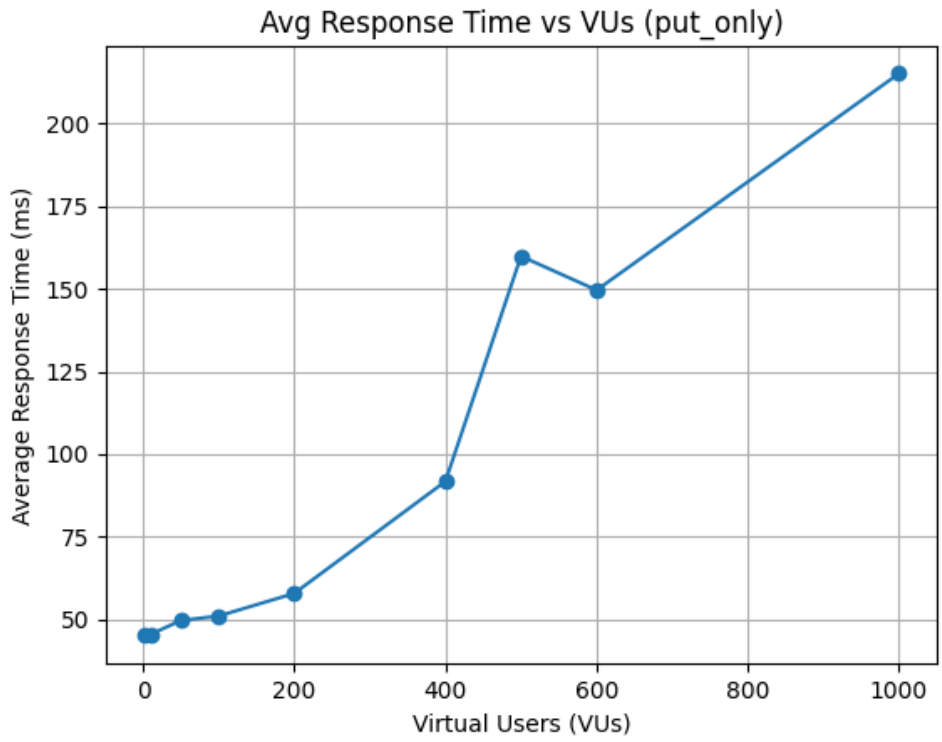
Example:

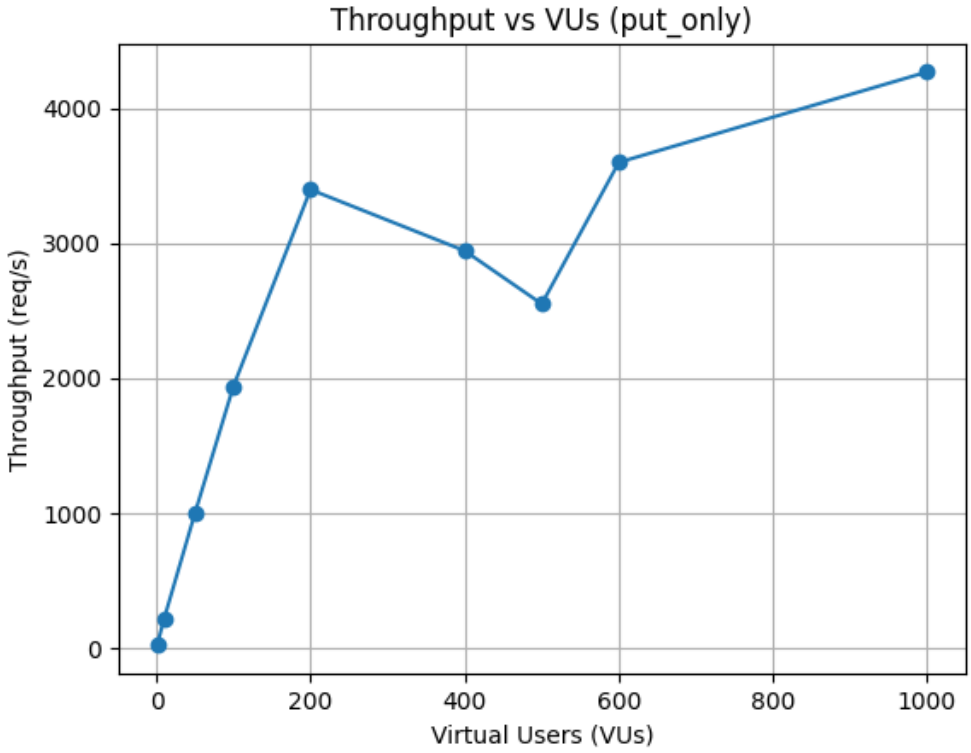
- results_get_only.csv
- results_put_only.csv
- results_get_popular.csv
- results_mixed.csv

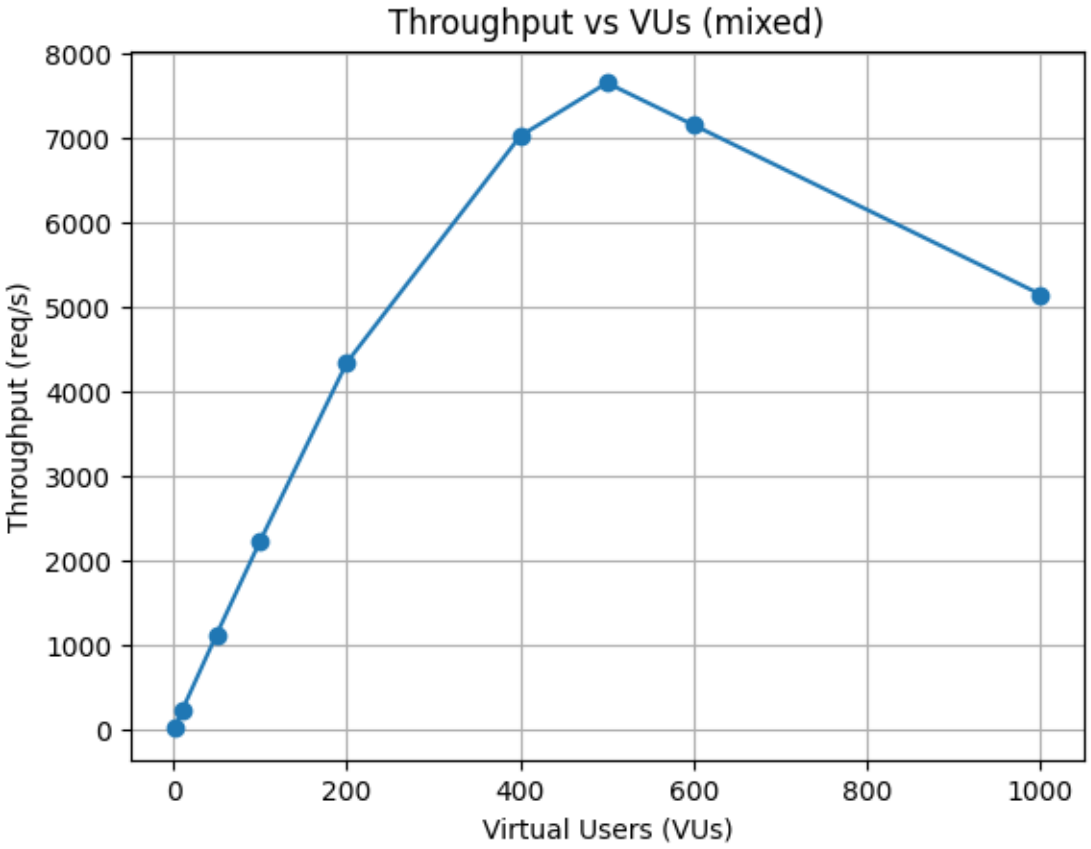
7. Graphs:

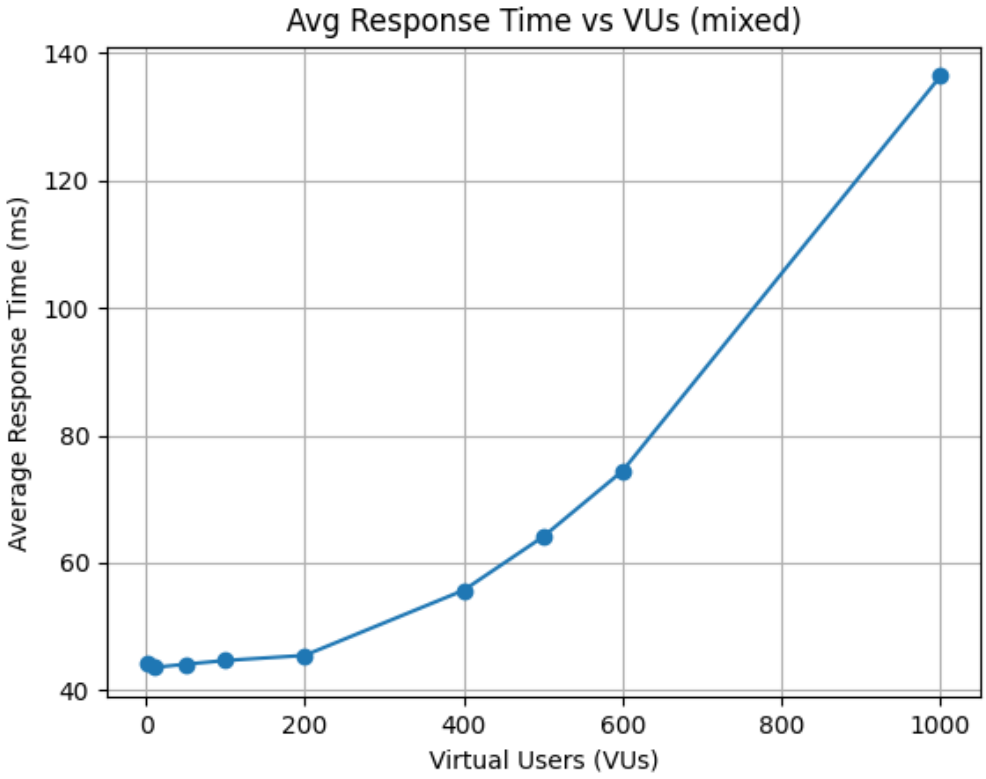

Avg Response Time vs VUs (get_only)
Average Response Time (ms)
70
60
50
40
30
0
200
400
600
800
1000
Virtual Users (VUs)

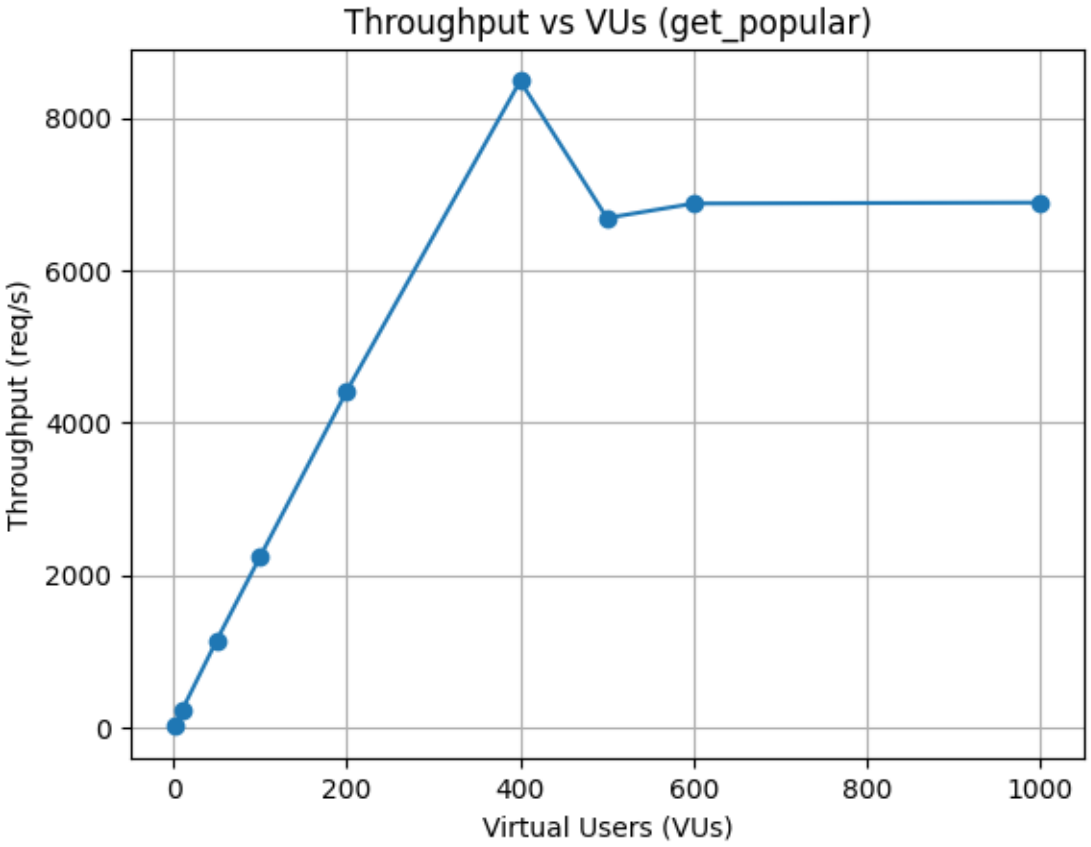

Throughput vs VUs (get_only)
Throughput (req/s)
5000
4000
3000
2000
1000
0
0
200
400
600
800
1000
Virtual Users (VUs)

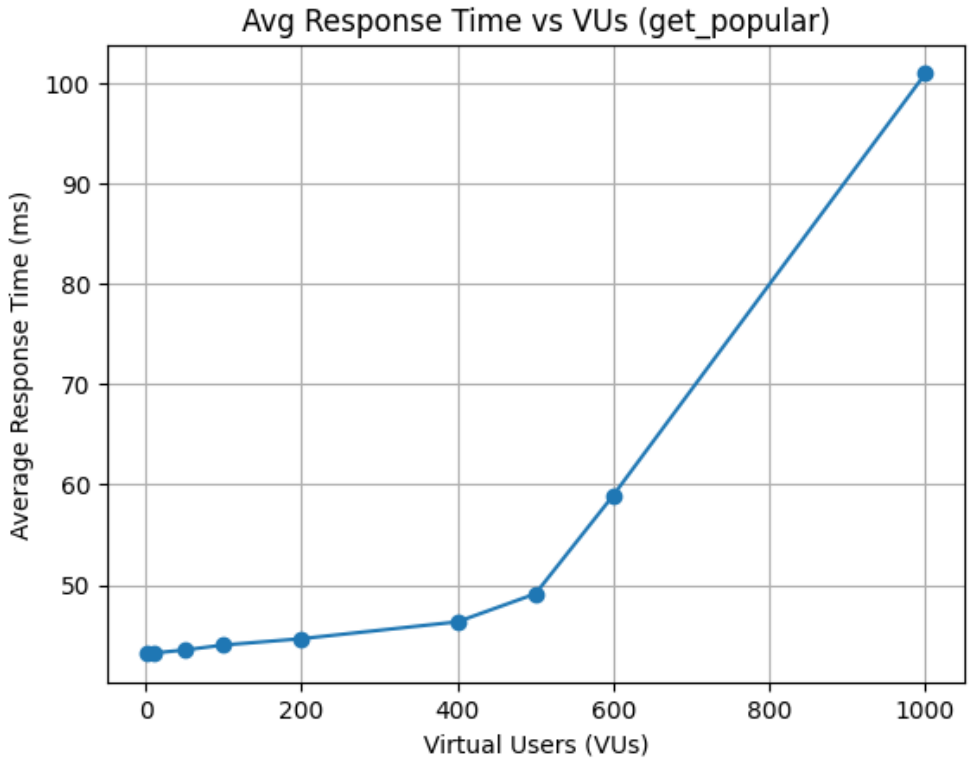

Avg Response Time vs VUs (put_only)
Average Response Time (ms)
200
175
150
125
100
75
50
0
200
400
600
800
1000
Virtual Users (VUs)


Throughput vs VUs (put_only)
Throughput (req/s)
4000
3000
2000
1000
0
0
200
400
600
800
1000
Virtual Users (VUs)


Throughput vs VUs (mixed)
Throughput (req/s)
Virtual Users (VUs)
0
1000
2000
3000
4000
5000
6000
7000
8000
0
200
400
600
800
1000


Avg Response Time vs VUs (mixed)
Average Response Time (ms)
Virtual Users (VUs)
40
60
80
100
120
140
0
200
400
600
800
1000


Throughput vs VUs (get_popular)
Throughput (req/s)
0
2000
4000
6000
8000
0
200
400
600
800
1000
Virtual Users (VUs)


Avg Response Time vs VUs (get_popular)
Average Response Time (ms)
50
60
70
80
90
100
0
200
400
600
800
1000
Virtual Users (VUs)