Graphion — Report

DECS Course Project — IIT Bombay
Name: Atharv Naik
GitHub Repository: https://github.com/atharv3903/graphion

---

1. Introduction

Graphion is a multi-tier routing system designed to compute shortest paths on real-world OpenStreetMap (OSM) road graphs. The system supports both read-heavy and write-heavy workloads and includes a custom load generator for systematic performance evaluation.

Phase-1 implements a fully functional system end-to-end, including data ingestion, a multi-tier backend, caching layers, routing algorithms, and HTTP APIs.

---

2. System Overview

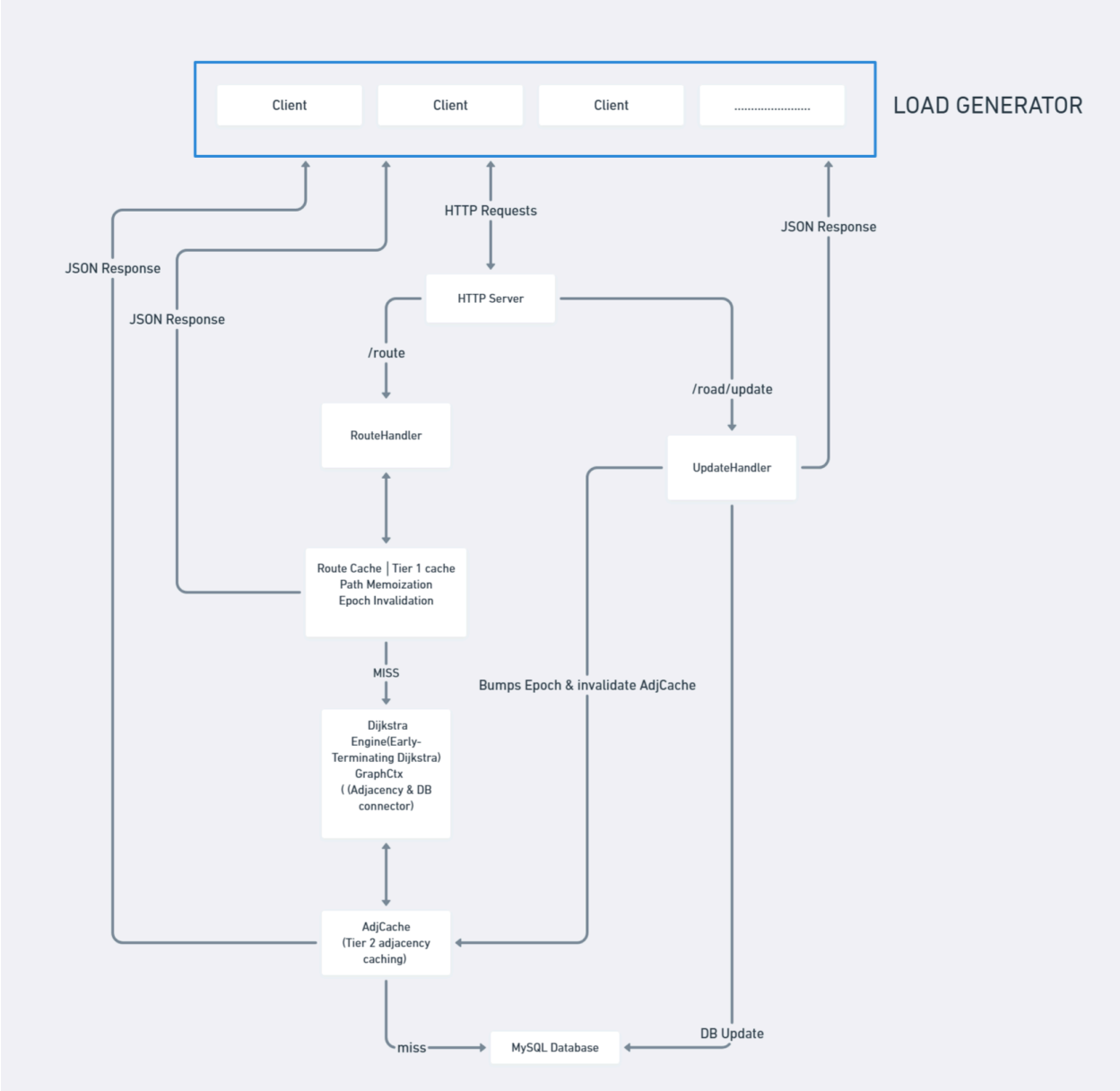
Graphion follows a **three-tier architecture**:

1. **Client Tier (Load Generator)**
2. **Server Tier (HTTP Routing + Caching + Algorithms)**
3. **Database Tier (MySQL Graph Storage)**

Each tier is modular and communicates strictly through defined interfaces, allowing clean debugging and controlled bottleneck testing.

---

3. Architecture Diagram



4. Detailed Design

4.1 Load Generator (Client Tier)

Located at `/cmd/loadgen`.

- Generates controlled parallel workloads
- Modes implemented:
    - **A1 – Cold-Start Route Load (CPU-Bound, Cache Cold)**
      Clears adjacency + route caches before starting. Forces full Dijkstra computation and DB fetches. Useful for identifying pure compute + cold-cache latency behavior.

    - **A2 – Warm Route Load (CPU-Bound, Cache Warm)**
      Same as A1 but with caches primed from previous queries. Tests steady-state routing latency, dominated by CPU and in-memory adjacency lookups.

    - **B – Write / Update Load (DB-Bound)**
      Issues `/road/update` requests at high parallelism. Stresses MySQL through repeated edge updates and triggers adjacency invalidation + route-cache epoch bumps.

4.2 HTTP Server (Middle Tier)

Located at `/cmd/server` and `/internal/api`.

Exposes three main APIs:

GET /route?src=X&dst=Y

Runs shortest path:

- Checks **RouteCache**
- If miss → calls Dijkstra
- Fills cache with computed path
- Returns:
    - path
    - total distance
    - explored nodes count
    - cache-hit boolean

POST /road/update

Updates edge attributes:

- Speed
- Closed/open
- Invalidates:
    - Adjacency cache (per-node)
    - Entire route-cache (epoch bump)

Debug Endpoints

- `/debug/clear_cache`
- `/debug/adjcache_stats` (gets/hits/puts)

This layer isolates HTTP, JSON, routing logic, and cache management.

4.3 Algorithms & Caching Layer (Backend Tier)

Dijkstra Engine

Located at `/internal/algo/dijkstra.go`.

- Priority-queue based implementation
- Supports cost functions (distance/speed)
- Returns:
    - path
    - total distance
    - explored nodes

Graph Context

Located at `/internal/algo/graphctx.go`.

Handles adjacency expansion:

- First check **AdjCache**
- Else read from DB
- Store result in cache

This minimizes DB load during routing.

Caches

Two independent in-memory caches:

1. AdjCache

- Key: node_id → outgoing edges
- Reduces DB reads significantly
- Tracks stats (gets, hits, puts)

2. RouteCache

- Key: {src,dst,algo,epoch}
- Stores entire path
- Epoch automatically invalidates cache after updates

This forms the core backend tier of the architecture.

4.4 Database Tier (MySQL)

Two tables:

`nodes(node_id, lat, lon)`

Stores point coordinates.

`edges(edge_id, src_node, dst_node, distance_m, speed_kmph, closed)`

Stores directed edges with metadata.

Used by:

- Dijkstra (via GraphCtx)
- /road/update to modify speeds
- IO-heavy workload mode

5. Graphion's Adjacency-Cached, Epoch-Aware, Early-Terminating Dijkstra

Graphion does **not** use the simplistic version of Dijkstra's algorithm.
Instead, it implements a **routing-optimized, cache-aware, lazy-expansion variant** . Graphion follows design principles similar to practical routing engines

Graphion's method:

```
push (src, 0) into min-heap
while heap not empty:
    u = pop smallest-distance node
    if u == dst:
        stop    // shortest path found
    for each neighbor v of u:
        relax(v)
```

---

6. Request Execution Paths in Graphion

Graphion provides two main API endpoints:

1. **GET /route?src=…&dst=…**
   → Computes shortest path between two nodes
2. **POST /road/update**
   → Updates edge attributes and invalidates caches

Below are the **complete execution paths**, showing how a request flows through all system tiers.

6.1 Execution Path for /route (Shortest Path Query)

This is the **CPU-bound pathway**, triggered by both the Loadgen (A1/A2) and external clients.

Step-by-step Request Flow

1 Client → HTTP Server

`GET /route?src=X&dst=Y`

arrives in:

`internal/api/server.go → handleRoute()`

2 Route Cache Lookup (In-Memory Tier #1)

`RC.Get(RouteKey)`

- If found → return cached path immediately

This implements **full route memoization**
Prevents recomputation for hot OD-pairs
Useful for A2 warm load

If cache hit, the system returns the cached path immediately

3 Call Dijkstra (Computation Tier)

If cache miss:

```
algo.Dijkstra(GCtx, src, dst, costFn)
```

where `GCtx` contains:

- **AdjCache** (Tier-2 cache)
- **Store** (MySQL-backed DB)

5 Inside Dijkstra → Request Adjacency from GraphCtx

```
neighbors, err := ctx.Neighbors(u)
```

6.1.1 Adjacency Retrieval Flow

Graphion uses a **lazy, cached adjacency model**:

1 Check AdjCache (In-Memory Tier #2)

```
AdjCache.Get(node_id)
```

- If found → return adjacency instantly
- No DB access
- Hit counter increments

This creates hot regions
Ensures A2 (warm load) becomes CPU-bound only

2 If AdjCache Miss → Fetch from Database

```
Store.Outgoing(node_id)
```

Then:

```
AdjCache.Put(node_id, edges)
```

The adjacency is now hot for future queries.

Only explored nodes hit the DB
Dijkstra expands only *useful* subgraph
1–2% of nodes accessed per route

3 Dijkstra Expansion Continues

Algorithm proceeds with:

- pop min-dist node from heap
- relax neighbors
- push updates into PQ
- stop immediately when `u == dst`

Early termination
Much faster than full-graph traversal
Backed by our cache-aware design

5 Path Reconstruction

Once dst is reached:

- Reconstruct path from `prev[]`
- Reverse slice
- Compute cost/metadata

6 Store in RouteCache

```
RouteCache.Put(key, path)
```

Ensures next query is **100% served from memory**.

7 Return JSON Response

6.2 Execution Path for /road/update (I/O-bound Update Pathway)

Triggered by loadgen's B workload.

1 Client → HTTP Server

2 DB Update (Persistent Tier)

Depending on request:

```
UpdateEdgeSpeed(edgeID, speed)
UpdateEdgeClosed(edgeID, closed)
```

Executes a SQL UPDATE inside Store.y

3 Invalidate Adjacency Cache Entry

If src_node provided:

```
AdjCache.Invalidate(src_node)
```

This forces Dijkstra to fetch fresh adjacency next time.

Ensures correctness after road changes
Localized invalidation (not full reset)

4 Bump Route Cache Epoch

```
RouteCache.BumpEpoch()
```

This globally invalidates all existing route cache entries.

Simple & safe global invalidation
Avoids stale routing paths after any update

5 Return JSON

7. Workload Design and Load Generators

Graphion's performance evaluation uses three purpose-built workload generators located under `/cmd/loadgen`.
Each workload targets different bottlenecks in the system.

7.1 Closed-Loop Route Load Generator (CPU-Bound)

Continuously sends `/route` requests using a closed-loop model.

- Measures: average latency, throughput
- Simulates: CPU/cache-bound load on the server

- Goal: Identify compute saturation point
- Observation: Throughput plateaus as CPU usage nears 100%.

7.2 DB-Bound Write Load Generator (I/O-Bound)

Fetches random edges from MySQL and performs `/road/update` writes.

- Measures: latency percentiles, throughput, DB load
- Simulates: database-heavy updates (edge speed/closure)
- Goal: Identify disk and I/O bottlenecks
- Observation: Latency increases as MySQL queue depth rises.

7.3 Mixed Read/Write Load Generator (80/20)

Combines 80% `/route` reads and 20% `/road/update` writes.

- Simulates: realistic production traffic
- Demonstrates interplay between caching and DB pressure
- Shows the tradeoff between CPU saturation and DB throughput.

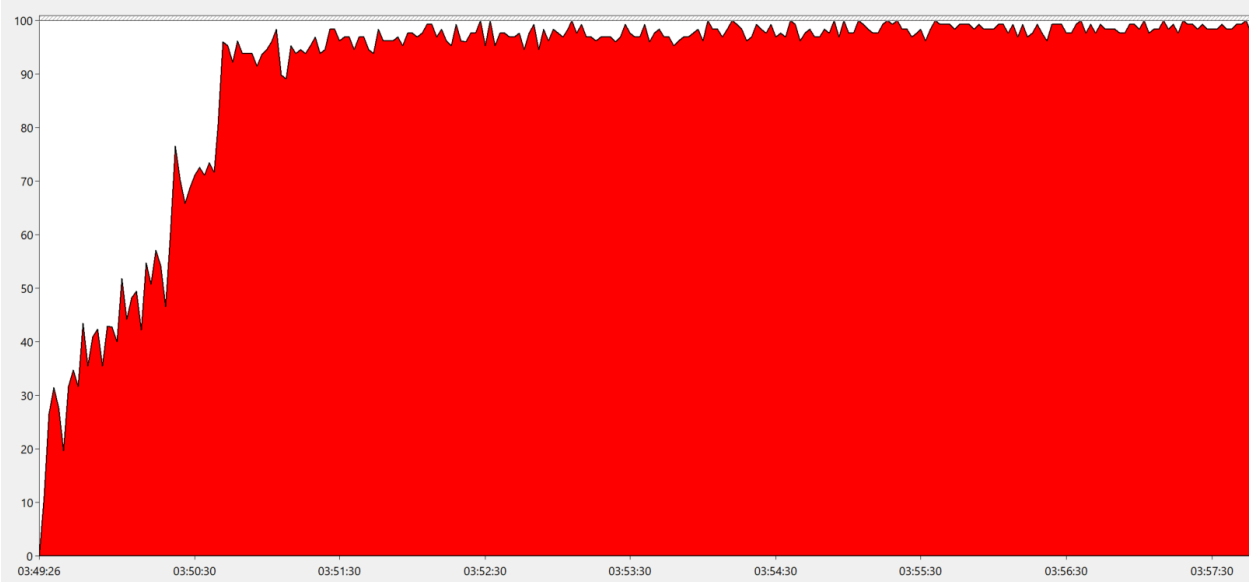
---

8. Performance Evaluation and Key Findings

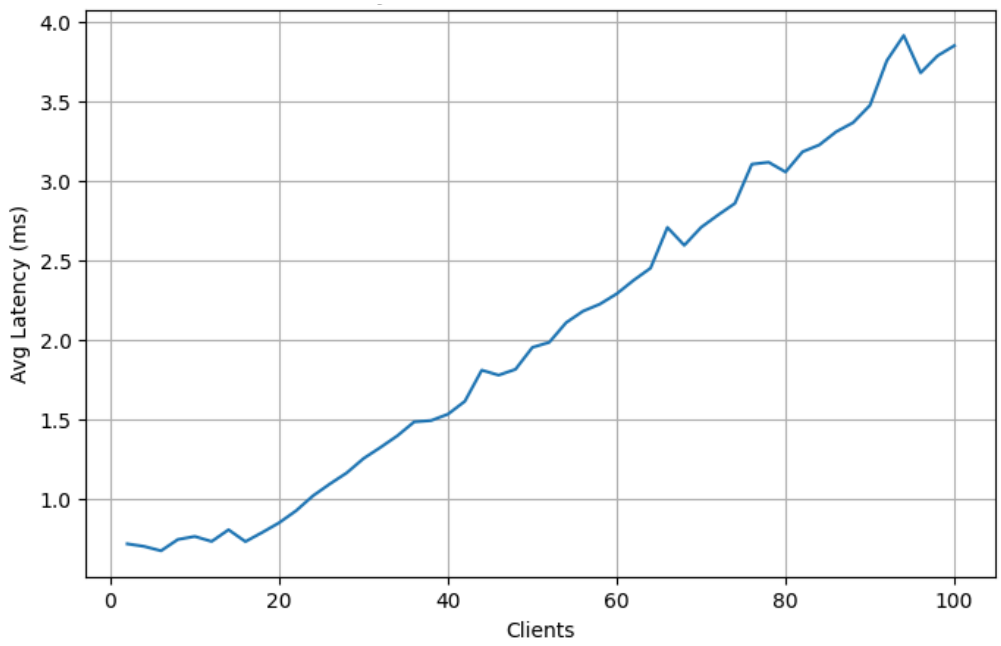
8.1 CPU-Bound Workload Results

Graphs:

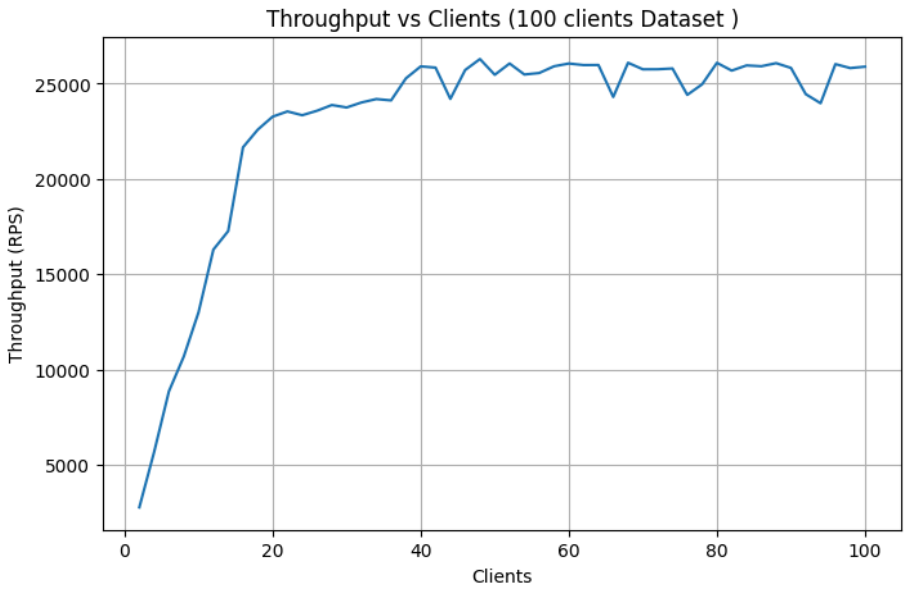
1. CPU Utilization (%) vs Time — shows core saturation



2. Clients vs Average Latency (ms) — latency climbs exponentially after a threshold



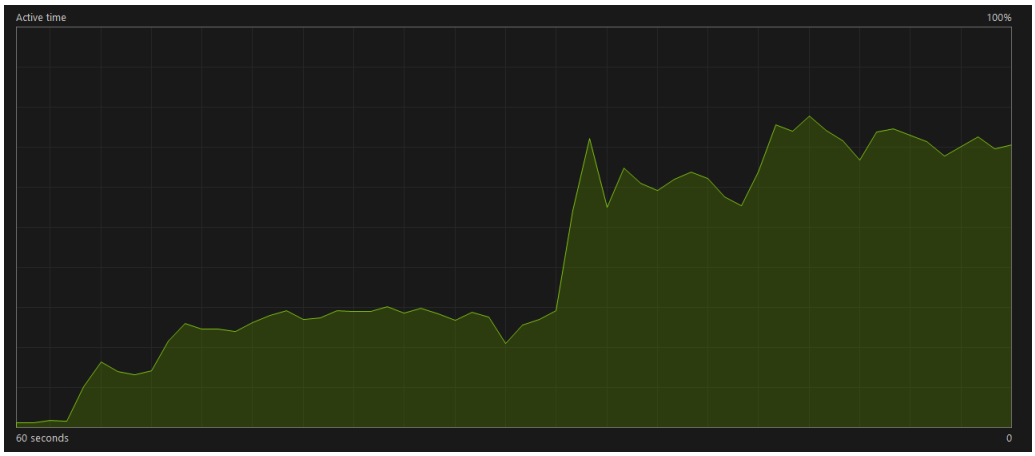
3. Clients vs Throughput (req/s) — throughput flattens after CPU saturation



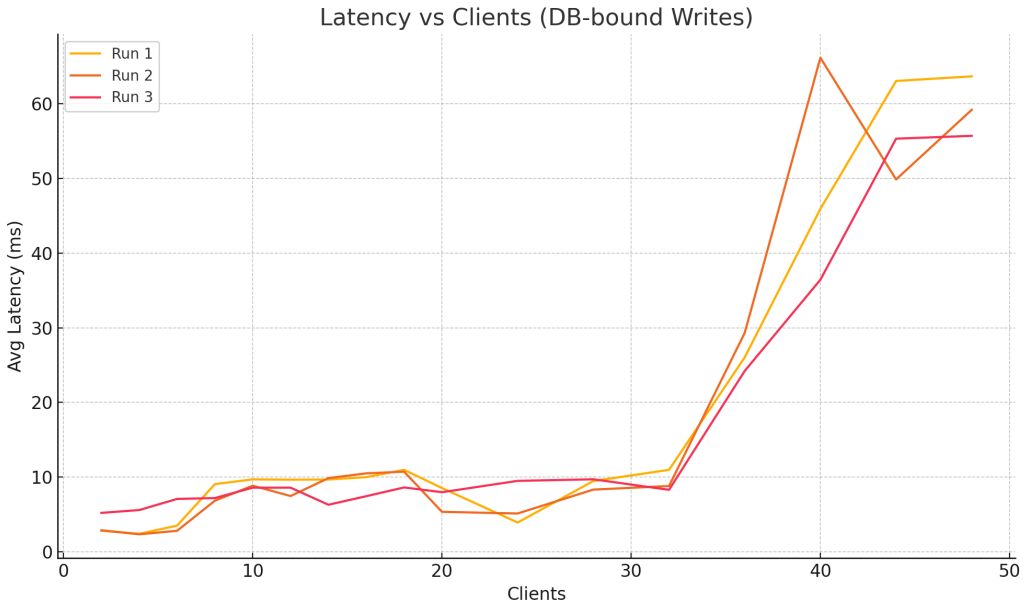
Key Finding:
Graphion saturates a single server core at high concurrency.Llatency increases while throughput stabilizes, confirming a **CPU bottleneck in the Dijkstra computation**.

8.2 I/O-Bound Workload Results

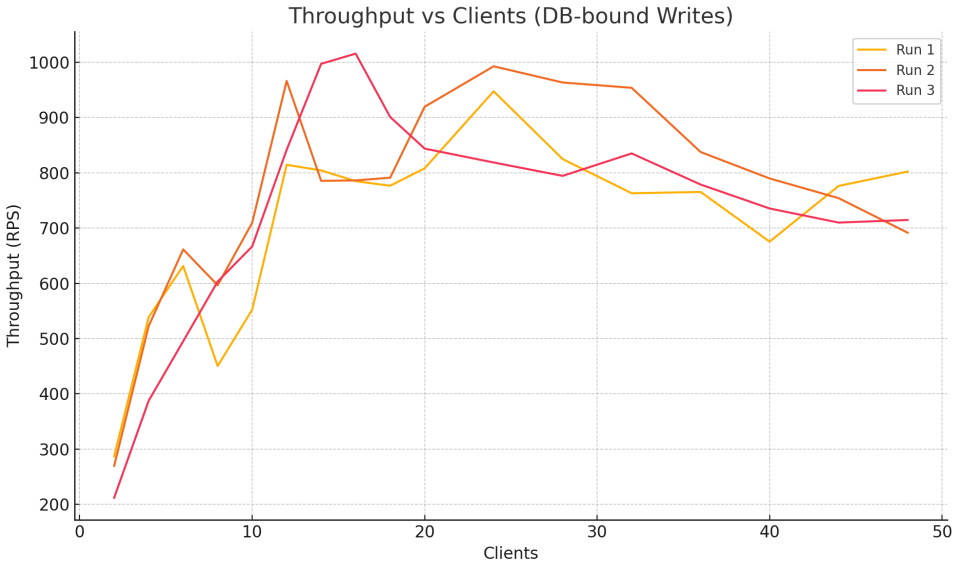
Graphs: **1.** Disk Utilization (%) vs Time under heavy updates



2. Clients vs Average Latency (ms) — increases steadily with more concurrent writers



3. Clients vs Throughput (req/s) — throughput saturates when MySQL becomes the bottleneck



Key Finding:

The `/road/update` path becomes **I/O-limited** at high concurrency,
Caching and batching help mitigate reads, but updates remain serialized and slow.