CSE 5306 - Distributed Systems Programming Assignment 2

Distributed URL Shortener Service

Team Members:
Abhijit Challapalli, Chaitanya Krishna Namburi

GitHub Repository:
GitHub Link

Instructor:

Dr. Jiayi Meng

Table of Contents

---

1. Functional Requirements

This distributed URL shortener system implements five core functional requirements:

FR1: URL Shortening

Purpose: Convert long URLs into short, memorable codes

Key Features: - Accepts URLs up to 2048 characters - Generates unique 7-character

Example:

```
Input:  https://www.example.com/very/long/path?id=12345
Output: http://localhost:8080/abc123X
```

FR2: URL Resolve

Purpose: Redirect users from short URLs to original destinations

Key Features: - HTTP 301 (Moved Permanently) redirects - Smart click counting

Status Codes: - **301** - Redirect successful - **404** - Link not found or expired - **410** - Maximum clicks exhausted - **429** - Rate limit exceeded

Flow:

```
User visits short URL
   ↓
Check if exists & active
   ↓
Count click (if GET request)
   ↓
Redirect to original URL
```

FR3: Expire By TTL(Time to Live)

Purpose: Support automatic link expiration based on Time

Time-based (TTL) - Set expiration time in seconds (1 sec to 1 year) - Automatic deletion by Redis - Status 404 when expired

FR4: Analytics & Top Links Leaderboard

Purpose: Track click counts and show most popular links

Key Features: - Sorted by popularity (descending click count) - Configurable limit (default: top 10) - Filters out expired links

Output Example:

```
Top 5 Links:
1. abc123 → 1,523 clicks → https://example.com
2. xyz789 →   892 clicks → https://google.com
3. def456 →   654 clicks → https://github.com
```

FR5: Link Expiration

Purpose: Support automatic link expiration based on usage

Click-based (Max Clicks) - Set maximum number of clicks (1 to 1,000,000) - Atomic counter decrement - Status 410 when exhausted

2. Proposed System Architectures

Overview

This project implements two distinct architectures to demonstrate different distributed system design patterns:

Aspect	Architecture 1	Architecture 2
Pattern	Microservices	Layered (3-Tier)
Communication	HTTP/REST + JSON	gRPC + Protocol Buffers
Port	8080	8081
Nodes	5 independent services	5 nodes
Coupling	Loose	Tight
Scalability	Horizontal (per service)	Vertical (entire app)
Data Format	JSON (text)	Protocol Buffers (binary)

Communication Models Comparison

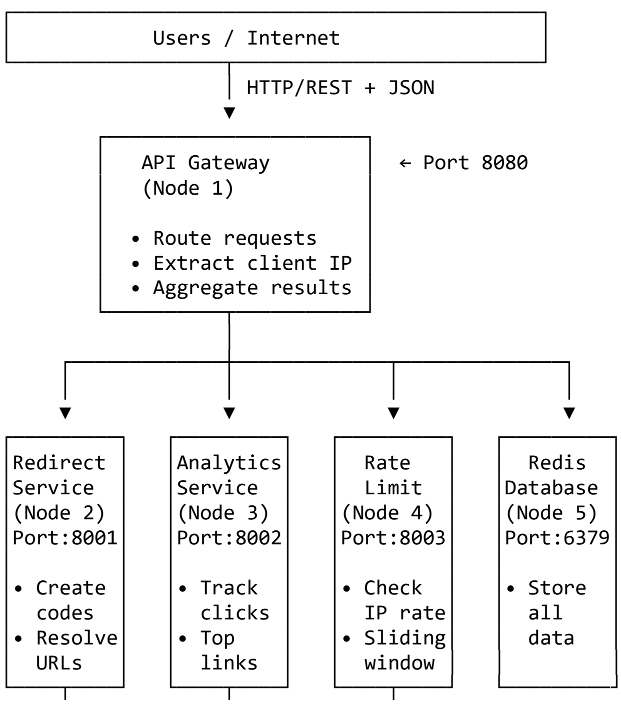
HTTP/REST (Microservices)

Characteristics: - **Protocol:** HTTP/1.1 - **Format:** JSON (human-readable) - **Size:** ~150 bytes per request - **Speed:** ~0.5ms serialization - **Debugging:** Easy (curl, browser)

gRPC + Protocol Buffers (Layered)

---

3. Architecture 1: Microservices (HTTP/REST)



```
│           │           │           ▲
└───────────┴───────────┴───────────┘
        All services connect to Redis
```

How the System Supports at Least Five Nodes

Node 1: API Gateway

Role: Single entry point, request orchestrator

Responsibilities: - Route requests to appropriate backend services - Extract client IP (handles X-Forwarded-For headers) - Aggregate responses from multiple services - Handle errors and return proper HTTP status codes

Node 2: Redirect Service

Role: Core URL shortening logic

Responsibilities: - Generate unique 7-character codes - Store URL mappings in Redis - Resolve short codes to original URLs - Manage click counting (atomic operations) - Detect and handle code collisions (5 retries)

Node 3: Analytics Service

Role: Click tracking and statistics

Responsibilities: - Increment click counters for each URL - Maintain sorted leaderboard (Redis Sorted Set) - Provide top N links API - Filter out expired URLs from results

Node 4: Rate Limit Service

Role: Request throttling and abuse prevention

Responsibilities: - Track requests per IP address - Implement sliding window algorithm - Return remaining quota to clients - Auto-cleanup expired rate limit data

Node 5: Redis Database

Role: Centralized data storage

Data Stored: - URL mappings (url:{code} → long_url) - Click counters (rem_clicks:{code} → remaining) - Analytics (zset:clicks → sorted by clicks) - Rate limits (ratelimit:{ip} → request timestamps) - Metadata (meta:{code} → creation time, settings)

Communication Flow

Example: Creating a Short URL

1. User → API Gateway
   POST /shorten
   {"long_url": "https://example.com"}

2. API Gateway → Rate Limit Service
GET /check?ip=192.168.1.1
Response: {"allowed": true, "remaining": 119}

3. API Gateway → Redirect Service
POST /shorten
{"long_url": "https://example.com"}

4. Redirect Service → Redis
- Check code collision
- Store url:abc123 → "https://example.com"
- Store metadata

5. Redirect Service → API Gateway
{"code": "abc123", "short_url": "http://localhost:8080/abc123"}

6. API Gateway → User
200 OK
{"code": "abc123", "short_url": "http://localhost:8080/abc123"}

How the System Supports the Five Functional Requirements

FR1: URL Shortening - Redirect Service generates unique codes - Redis stores URL mappings - API Gateway coordinates the process

FR2: URL Resolution - Redirect Service resolves codes to URLs - Lua script ensures atomic click counting - API Gateway returns 301 redirect

FR3: Expire by TTL - Rate Limit Service TTL: Redis EXPIRE command

FR4: Analytics - Analytics Service (dedicated service) - Redis Sorted Set for leaderboard - Async updates (non-blocking)

FR5: Link Expiration Max Clicks: The link expire when it reaches the max count

Docker Deployment

Container Configuration

```
5 Docker Containers:
├── api-gateway     (Python/FastAPI, Port 8080)
├── redirect        (Python/FastAPI, Port 8001)
├── analytics       (Python/FastAPI, Port 8002)
├── ratelimit       (Python/FastAPI, Port 8003)
└── redis           (Redis 7 Alpine, Port 6379)
```

Network: urlshortener-net (bridge)

Name	Last Pushed ↑	Contains
abchalla/urlshort-layered-redis-replica	about 5 hours ago	IMAGE
abchalla/urlshort-layered-redis-master	about 21 hours ago	IMAGE
abchalla/urlshort-gateway	about 21 hours ago	IMAGE
abchalla/urlshort-redirect	about 22 hours ago	IMAGE
abchalla/urlshort-layered-nginx	about 23 hours ago	IMAGE
abchalla/urlshort-layered-worker	about 23 hours ago	IMAGE
abchalla/urlshort-layered-app	about 23 hours ago	IMAGE
abchalla/urlshort-redis	7 days ago	IMAGE
abchalla/urlshort-ratelimit	7 days ago	IMAGE
abchalla/urlshort-analytics	7 days ago	IMAGE

---

4. Architecture 2: Layered (gRPC)

System Architecture Diagram

```
┌──────────────────────────────────────────────┐
│              Users / Internet                │
└──────────────────────────────────────────────┘
                  │ HTTP/2 (gRPC)
                  ▼
        ┌──────────────────────┐
        │  Nginx Proxy         │  ← Port 8081
        │  (Node 1)            │
        │ • HTTP/2 termination │
        │ • Load balancing     │
        └──────────────────────┘
                  │ gRPC
                  ▼
     ┌──────────────────────────────────────┐
     │  Layered Application (Node 2)        │
     │                                      │
     │  ┌───────────────────────────────┐  │
     │  │ Layer 1: Presentation         │  │
     │  │ • gRPC handlers               │  │
     │  │ • Request validation          │  │
     │  │ • Response marshaling         │  │
     │  └───────────────────────────────┘  │
     │              │ Function Calls        │
     │  ┌───────────▼───────────────────┐  │
```

```
│  │  Layer 2: Business Logic        │  │
│  │  • URL validation               │  │
│  │  • Code generation              │  │
│  │  • Rate limiting                │  │
│  │  • Expiration logic             │  │
│  └──────────────┬──────────────────┘  │
│                 │ Function Calls      │
│  ┌──────────────▼──────────────────┐  │
│  │  Layer 3: Repository            │  │
│  │  • Redis operations             │  │
│  │  • Lua script execution         │  │
│  │  • Data consistency             │  │
│  └──────────────┬──────────────────┘  │
└─────────────────┼─────────────────────┘
                  │ Redis Protocol
     ┌────────────┴─────────────┐
     ▼                          ▼
┌──────────┐              ┌──────────┐
│  Redis   │◄─Replication─│  Redis   │
│  Master  │              │ Replica  │
│ (Node 3) │──────────────│ (Node 4) │
└────┬─────┘              └──────────┘
     │
     ▼
┌──────────┐
│Analytics │
│ Worker   │
│ (Node 5) │
│          │
│• Global  │
│  stats   │
│• Cleanup │
└──────────┘
```

Layer Components

Node 1: Nginx Proxy (Edge / Ingress)

Purpose: Public entry point, HTTP/2 + gRPC reverse proxy, TLS termination, load balancing to Node 2 replicas.

Node 2: Layered Application (gRPC server)

Port: 50051 (gRPC).

Purpose: Implement core business rules

Layer 1: Presentation (Transport/Adapter)

- gRPC service/handlers, protobuf (validate/marshal), map RPCs ↔ service calls, status codes.

Layer 2: Business Logic (Service)

- URL validation, code generation, per-client rate limiting, TTL/expiration policies, orchestration of repo ops.
- Key functions: create_short_url() (FR1), resolve_url() (FR2), check_rate_limit() (FR3, internal), get_top_links() (FR4), expiration handling (FR5).

Layer 3: Repository (Data Access / Ports & Adapters)

- Redis client ops, Lua scripts for atomic read-modify-write, consistency rules, connection pooling, caching strategies.
- Ops: store_url(), resolve_url() (atomic read + decrement), increment_click(), check_rate_limit() (sliding window), get_top_links().

Node 3: Redis Master

Purpose: Primary data storage

Responsibilities: - Handle all write operations - Replicate data to replica - Execute Lua scripts - Manage TTL expiration

Node 4: Redis Replica

Purpose: Read scalability and redundancy

Responsibilities: - Async replication from master - Serve read-only queries (optional) - Provide data redundancy - Automatic failover (with Sentinel)

Node 5: Analytics Worker

Purpose: Background data processing

Responsibilities: - Calculate global statistics every 10 seconds - Aggregate click data - Clean up expired entries - Store summary metrics

Runs Independently: Non-blocking background process

How the System Supports Five Functional Requirements

FR1: URL Shortening - Service Layer generates codes & validates - Repository Layer stores in Redis - Presentation Layer exposes gRPC API

FR2: URL Resolution - Repository Layer executes Lua script (atomic) - Service Layer orchestrates resolution - Presentation Layer returns gRPC response

FR3: TTL Expiry - TTL: Repository sets Redis EXPIRE

FR4: Analytics - Repository Layer updates sorted set - Service Layer provides top links API - Worker Node calculates global stats

FR5: Link Expiration - Max Clicks: Lua script in Repository - Service Layer handles expiration logic

Docker Deployment

Container Configuration

```
5 Docker Containers:
├── nginx           (Nginx Alpine, Port 8081)
├── layered-app     (Python/gRPC, 3 layers in 1)
├── redis-master    (Redis 7 Alpine, Primary)
├── redis-replica   (Redis 7 Alpine, Read-only)
└── analytics-worker (Python, Background job)
```

Network: layered-net (bridge)

Name	Last Pushed ↑	Contains
abchalla/urlshort-layered-redis-replica	about 5 hours ago	IMAGE
abchalla/urlshort-layered-redis-master	about 21 hours ago	IMAGE
abchalla/urlshort-gateway	about 21 hours ago	IMAGE
abchalla/urlshort-redirect	about 22 hours ago	IMAGE
abchalla/urlshort-layered-nginx	about 23 hours ago	IMAGE
abchalla/urlshort-layered-worker	about 23 hours ago	IMAGE
abchalla/urlshort-layered-app	about 23 hours ago	IMAGE
abchalla/urlshort-redis	7 days ago	IMAGE
abchalla/urlshort-ratelimit	7 days ago	IMAGE
abchalla/urlshort-analytics	7 days ago	IMAGE

---

5. Evaluation

1) Experimental Setup

Host & Runtime

- Single laptop host, Windows (MINGW64 shell), Docker network urlshortener-net.

Containers Involved (per test run)

- **Application – Microservices:** api-gateway exposing an HTTP resolve path on **:8080** (expected **301** with Location).
- **Application – Layered:** single layered (monolithic) app exposing a **gRPC** resolve method on **:8081** (status **OK**).
- **Data Plane: Redis** shared by both architectures.
- **Test Infrastructure:**
    - Microservices → grafana/k6:latest (load) + influxdb:8086 (metrics)
    - Layered → **ghz** (gRPC load; HTML summaries)

Service-Level Objectives (SLOs)

- **p95 latency < 200 ms**; **p99 latency < 500 ms**; **error rate < 1%**.

Workloads

- **Microservices (k6, constant VUs, 60s): 50**, **100**, **200** VUs; script /work/k6-resolve.js validates **301** + Location.
- **Layered (ghz, constant RPS, 60s): 100 rps** (conc **25**), **200 rps** (conc **50**), **400 rps** (conc **100**); success = gRPC **OK**.

Fairness Note (test model)

- Microservices used **open-loop** (constant users); arrivals keep coming even if the system slows—this exposes saturation and inflates tails.
- Layered used **closed-loop** (constant RPS); arrival rate is capped—this shows how well the system tracks a target pace.
- Results are interpreted with this difference in mind.

What tools did I use and why?

- **k6** (from Grafana Labs): a tool that pretends to be a bunch of users hitting my website. I tell it “act like 50 / 100 / 200 people constantly using the app for 60 seconds,” and it measures how fast the system responds and how many requests it completes per second.
- **InfluxDB + Grafana**: I treat **InfluxDB** as a time-series notebook where k6 writes all the numbers it measures every second. **Grafana** is the dashboard app I can use to graph those numbers (lines going up/down).

- **ghz**: a load tester specifically for **gRPC** (the protocol my layered app uses). Instead of "X fake users," with ghz I usually say "send **Y requests per second** for 60 seconds" and it tries to keep that pace.

What kinds of tests did I run?

Microservices path (HTTP via API gateway) — with k6

- I ran **constant VU** tests: "pretend I have **50**, then **100**, then **200** people all clicking continuously for **60 seconds.**"
- k6 checked each response: it should be an HTTP **301 redirect** and include a **Location** header (meaning the short link correctly redirects).
- I set goals (SLOs):
  - **p95 latency < 200 ms** (95% of requests should finish faster than 200 ms)
  - **p99 latency < 500 ms**
  - **Error rate < 1%**

Load Tests for Microservices

vus	http_reqs	rps	p95_ms	p99_ms	err_rate	throughput_per_vu	met_p95_lt_200ms	met_p99_lt_500ms
50	19291	321.057842	199.02	268.51	0	6.42115684	TRUE	TRUE
100	18999	315.651064	385.4	445.08	0	3.15651064	FALSE	TRUE
200	4587	75.609345	2940	3240	0	0.378046725	FALSE	FALSE

What did the results say

Microservices (HTTP via gateway, k6)

- 50 users → about 321 requests/second, and it's fast enough (95% done in ~199 ms).
- 100 users → still about 316 requests/second (so no extra throughput). This means I've hit a ceiling; adding more "people" doesn't increase completed work. Also, 95% finish in ~385 ms now—too slow vs the 200 ms goal.
- 200 users → throughput drops to ~76 requests/second and response times explode to seconds. The system is backed up—like checkout lines wrapping around the store.

Bottom line: My microservices route works well up to ~320 RPS, but after that it's saturated: the line grows, people wait longer, and the store actually checks out fewer people per second.

Performance & Scalability :

Microservices (HTTP via API Gateway)

Raw results (60s runs)

VUs	RPS	p95 (ms)	p99 (ms)	Errors
50	321.06	199.02	268.51	0%
100	315.65	385.40	445.08	0%
200	75.61	2,940	3,240	0%

Interpretation: (Performance)

50 VUs: Fast and SLO-compliant (≈321 RPS, p95 ≈ 199 ms).

100 VUs: Throughput flat (~316 RPS). This indicates a ceiling: adding users doesn't increase work done. p95 rises to ~385 ms (SLO breach).

200 VUs: The system backs up (queues form). Throughput falls to ~76 RPS, and p95/p99 jump into seconds.

Scalability takeaway

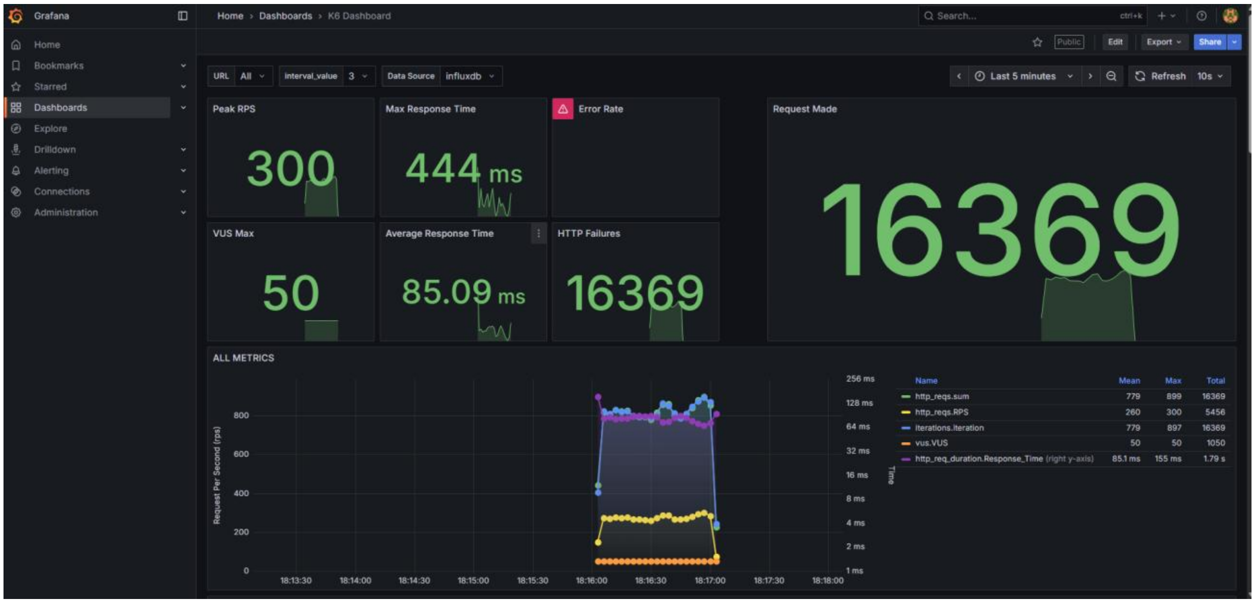
SLO-compliant capacity ≈ 320 RPS. Past that knee, tail latencies spike and throughput collapses under load.

Test results for 50 Virtual users with duration 60 seconds

```
running (1m00.0s), 50/50 VUs, 16310 complete and 0 interrupted iterations
default   [ 100% ] 50 VUs  1m00.0s/1m0s
time="2025-10-12T23:17:03Z" level=info msg="[k6-reporter v3.0.0] Generating HTML summary report, with theme: default" source=console
     ✓ status is 301
     ✓ has Location

     checks.........................: 100.00% ✓ 32738      ✗ 0
     data_received..................: 2.3 MB  39 kB/s
     data_sent......................: 1.8 MB  30 kB/s
     http_req_blocked...............: avg=7.61µs   min=939ns    med=3.35µs   max=2.87ms   p(90)=4.8µs    p(95)=5.45µs
     http_req_connecting............: avg=1.46µs   min=0s       med=0s       max=2.03ms   p(90)=0s       p(95)=0s
   ✓ http_req_duration..............: avg=82.89ms  min=8.06ms   med=74.82ms  max=443.74ms p(90)=127.14ms p(95)=149.27ms
       { expected_response:true }...: avg=82.89ms  min=8.06ms   med=74.82ms  max=443.74ms p(90)=127.14ms p(95)=149.27ms
   ✓ http_req_failed................: 0.00%   ✓ 0          ✗ 16369
     http_req_receiving.............: avg=47.36µs  min=8.2µs    med=41.71µs  max=2.8ms    p(90)=67.69µs  p(95)=80.36µs
     http_req_sending...............: avg=11.27µs  min=2.08µs   med=8.52µs   max=3.63ms   p(90)=15.08µs  p(95)=20.43µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=82.84ms  min=7.99ms   med=74.76ms  max=443.7ms  p(90)=127.07ms p(95)=149.2ms
     http_reqs......................: 16369   272.138822/s
     iteration_duration.............: avg=183.56ms min=109.14ms med=175.42ms max=544.55ms p(90)=227.89ms p(95)=249.83ms
     iterations.....................: 16369   272.138822/s
     vus............................: 50      min=50       max=50
     vus_max........................: 50      min=50       max=50
running (1m00.1s), 00/50 VUs, 16369 complete and 0 interrupted iterations
default ✓ [ 100% ] 50 VUs  1m0s

abhij@Abhijit MINGW64 ~/PhD/Fall 2025/CSE 5306 Distributed Systems/Assignments/Assignment2/CSE-5306-DS-PA2-URLShortener (main)
```



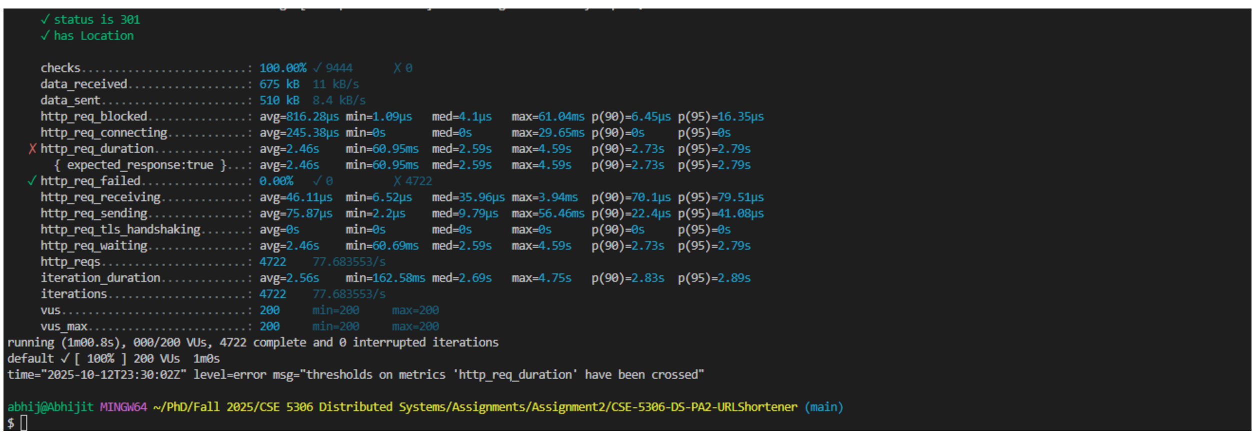
100 Virtual users with duration 60 seconds

```
time="2025-10-12T23:19:48Z" level=info msg="[k6-reporter v3.0.0] Generating HTML summary report, with theme: default" source=console
     ✓ status is 301
     ✓ has Location

     checks.........................: 100.00% ✓ 37910      X 0
     data_received..................: 2.7 MB  45 kB/s
     data_sent......................: 2.1 MB  34 kB/s
     http_req_blocked...............: avg=71.02µs  min=800ns    med=3.38µs   max=26.42ms  p(90)=5.07µs   p(95)=5.75µs
     http_req_connecting............: avg=23.5µs   min=0s       med=0s       max=22.11ms  p(90)=0s       p(95)=0s
   ✓ http_req_duration..............: avg=216.21ms min=3.98ms   med=216.97ms max=537.9ms  p(90)=301.77ms p(95)=333.74ms
       { expected_response:true }...: avg=216.21ms min=3.98ms   med=216.97ms max=537.9ms  p(90)=301.77ms p(95)=333.74ms
   ✓ http_req_failed................: 0.00%   ✓ 0          X 18955
     http_req_receiving.............: avg=39.31µs  min=6.1µs    med=33.37µs  max=14.09ms  p(90)=61.09µs  p(95)=73.08µs
     http_req_sending...............: avg=13.78µs  min=1.96µs   med=8.31µs   max=21.82ms  p(90)=14.69µs  p(95)=20.65µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=216.16ms min=3.94ms   med=216.92ms max=537.87ms p(90)=301.73ms p(95)=333.64ms
     http_reqs......................: 18955   315.16331/s
     iteration_duration.............: avg=317ms    min=104.79ms med=317.61ms max=638.86ms p(90)=402.55ms p(95)=434.87ms
     iterations.....................: 18955   315.16331/s
     vus............................: 100     min=100      max=100
     vus_max........................: 100     min=100      max=100
running (1m00.1s), 000/100 VUs, 18955 complete and 0 interrupted iterations
default ✓ [ 100% ] 100 VUs  1m0s

abhij@Abhijit MINGW64 ~/PhD/Fall 2025/CSE 5306 Distributed Systems/Assignments/Assignment2/CSE-5306-DS-PA2-URLShortener (main)
$
```

200 Virtual users with duration 60 seconds



```
  ✓ status is 301
  ✓ has Location

  checks.........................: 100.00% ✓ 9444      ✗ 0
  data_received..................: 675 kB  11 kB/s
  data_sent......................: 510 kB  8.4 kB/s
  http_req_blocked...............: avg=816.28µs min=1.09µs   med=4.1µs   max=61.04ms p(90)=6.45µs p(95)=16.35µs
  http_req_connecting............: avg=245.38µs min=0s       med=0s      max=29.65ms p(90)=0s     p(95)=0s
✗ http_req_duration..............: avg=2.46s    min=60.95ms  med=2.59s   max=4.59s   p(90)=2.73s  p(95)=2.79s
    { expected_response:true }...: avg=2.46s    min=60.95ms  med=2.59s   max=4.59s   p(90)=2.73s  p(95)=2.79s
✓ http_req_failed................: 0.00%   ✓ 0         ✗ 4722
  http_req_receiving.............: avg=46.11µs  min=6.52µs   med=35.96µs max=3.94ms  p(90)=70.1µs p(95)=79.51µs
  http_req_sending...............: avg=75.87µs  min=2.2µs    med=9.79µs  max=56.46ms p(90)=22.4µs p(95)=41.08µs
  http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s      p(90)=0s     p(95)=0s
  http_req_waiting...............: avg=2.46s    min=60.69ms  med=2.59s   max=4.59s   p(90)=2.73s  p(95)=2.79s
  http_reqs......................: 4722    77.683553/s
  iteration_duration.............: avg=2.56s    min=162.58ms med=2.69s   max=4.75s   p(90)=2.83s  p(95)=2.89s
  iterations.....................: 4722    77.683553/s
  vus............................: 200     min=200      max=200
  vus_max........................: 200     min=200      max=200
running (1m00.8s), 000/200 VUs, 4722 complete and 0 interrupted iterations
default ✓ [ 100% ] 200 VUs  1m0s
time="2025-10-12T23:30:02Z" level=error msg="thresholds on metrics 'http_req_duration' have been crossed"

abhij@Abhijit MINGW64 ~/PhD/Fall 2025/CSE 5306 Distributed Systems/Assignments/Assignment2/CSE-5306-DS-PA2-URLShortener (main)
$
```

Layered path (gRPC) — with ghz:

I ran constant arrival rate tests: “send 100, 200, then 400 requests per second for 60 seconds,” while limiting maximum parallel requests (concurrency 25/50/100).

ghz checked that each response was gRPC OK.

I judged results against the same SLOs (p95 < 200 ms, p99 < 500 ms, errors < 1%).

Load Tests for Layered

target_rps	concurrency	count	rps	p50_ms	p95_ms	p99_ms	ok	errors	err_rate	throughput_per_conn
100	25	5999	99.9	1.56	2.55	3.36	5999	0	0	3.996
200	50	11999	199.98	1.33	2.15	2.82	11999	0	0	3.9996
400	100	23999	399.96	1.54	3.74	4.81	23998	1	4.17E-05	3.9996

Raw results (60s runs)

Target RPS	Concurrency	Achieved RPS	p95 (ms)	p99 (ms)	Errors
100	25	**99.90**	**2.55**	3.36	0.000%
200	50	**199.98**	**2.15**	2.82	0.000%
400	100	**399.96**	**3.74**	4.81	0.004% (1 transient)

What did the results say: (Performance and Scalability)

Layered (gRPC, ghz)

- 100, 200, 400 requests/second → it keeps up perfectly with the target pace.
- Response times are tiny: 95% finish in ~2–4 milliseconds (that's thousandths of a second), even at 400 RPS.
- Errors are basically zero (one harmless connection blip at 400 RPS).

Interpretation (Performance)

- Tracks the **target pace perfectly** at **100/200/400 RPS**.
- Latency is **tiny** (p95 **2–4 ms**, p99 **3–5 ms**) with **near-zero** errors.
- No visible saturation at **400 RPS**; substantial headroom remains.

Scalability takeaway

- SLO-compliant **≥ 400 RPS** with millisecond-level tails; no knee observed in tested range.

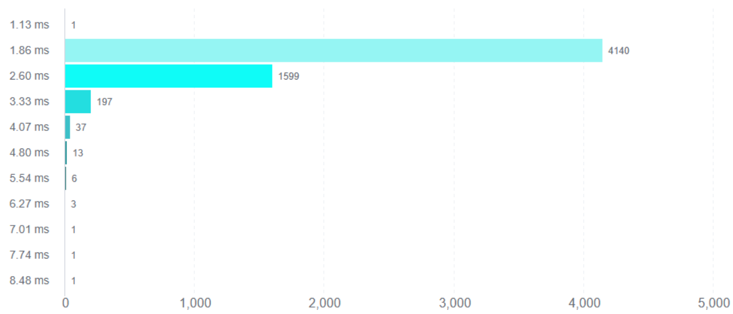
Test Plots:

(100 Requests Per Second)

Summary

Count	5999
Total	60.05 s
Slowest	8.48 ms
Fastest	1.13 ms
Average	1.70 ms
Requests / sec	99.90

Histogram



Latency distribution

10 %	25 %	50 %	75 %	90 %	95 %	99 %
1.23 ms	1.31 ms	1.56 ms	1.97 ms	2.31 ms	2.55 ms	3.36 ms

Status distribution

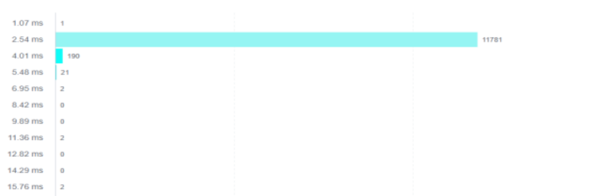
Status	Count	% of Total
OK	5999	100.00 %

200 Requests Per Second

Summary

Count	11999
Total	60.00 s
Slowest	15.76 ms
Fastest	1.07 ms
Average	1.46 ms
Requests / sec	199.98

Histogram



Latency distribution

10 %	25 %	50 %	75 %	90 %	95 %	99 %
1.19 ms	1.23 ms	1.33 ms	1.56 ms	1.92 ms	2.15 ms	2.82 ms

Status distribution

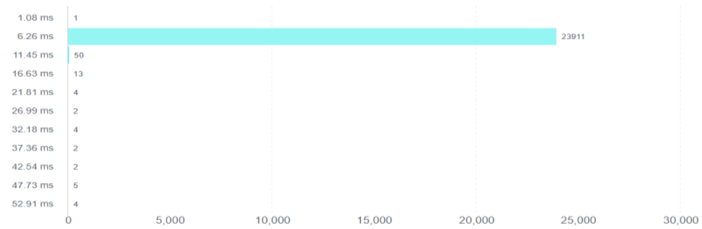
Status	Count	% of Total
OK	11999	100.00 %

400 Requests Per Second

Summary

Count	23999
Total	60.00 s
Slowest	52.91 ms
Fastest	1.08 ms
Average	1.91 ms
Requests / sec	399.96

Histogram



Latency distribution

10 %	25 %	50 %	75 %	90 %	95 %	99 %
1.21 ms	1.27 ms	1.54 ms	2.14 ms	2.80 ms	3.74 ms	4.81 ms

Status distribution

Status	Count	% of Total
OK	23998	100.00 %
Unavailable	1	0.00 %

Side-by-Side Comparison

Why is layered so much faster than microservices here?

I think of it like travel:

- **Layered** is a **nonstop flight**: a request goes straight to the code that knows the answer—very few "stops," very little overhead.
- **Microservices** is a **connecting flight**: a request hits the **gateway**, then hops to another service, then to Redis, then back—more "stops" and more chances for queues to build up.

Architecture	SLO-Compliant Capacity	Onset of Saturation ("knee")
Microservices	~**321 RPS** (50 VUs)	~**316 RPS** (100 VUs): p95 breach; at 200 VUs throughput drops sharply
Layered	≥ **400 RPS** (p95 < **4 ms**)	None observed up to **400 RPS**

What this means in practice

- At similar throughput levels (~300–400 RPS), **layered** operates in **single-digit milliseconds**, while **microservices** is in **hundreds of milliseconds** and becomes unstable once past its knee.

Analysis of System-Design Trade-offs

Why the layered path is faster here

- **Fewer hops & lighter protocol:** Layered is a **nonstop flight** (client → app → Redis), with efficient **gRPC** framing. Microservices is a **connecting flight** (client → gateway → service → Redis → back), adding network hops and coordination overhead.
- **Queueing behavior:** Microservices hits pool/connection bottlenecks sooner; once queues form, p95/p99 balloon (classic backpressure).

But microservices still buys you important benefits

- **Fault isolation & blast radius:** a failing service is contained.
- **Independent scaling & team autonomy:** scale hot services; deploy independently.
- **Finer security/observability boundaries:** clearer per-service SLOs and policies.

Trade-Off Summary:

Dimension	**Layered (gRPC)**	**Microservices (Gateway + Service)**
Latency / Throughput	Excellent (p95 ≈ 2–4 ms @ 400 RPS)	Good to ~320 RPS; tails rise fast past knee
Scalability Mechanics	Coarse-grained (scale whole app)	Fine-grained (scale hot services)
Fault Isolation	Weaker (shared process)	Stronger (per-service isolation)
Team Autonomy	Lower	Higher (independent deploys)
Operational Complexity	Lower (fewer moving parts)	Higher (discovery, retries, tracing, config)
Caching / Locality	In-process locality is easy	Cross-service/cache hops add latency
Baseline Cost	Lower	Higher (more infra components)

Conclusion

So what should I take away?

- For **raw speed and smooth scaling**, my **layered** (gRPC) path wins by a mile in these tests. It's doing **400 RPS** with **millisecond** response times.
- My **microservices** path is fine up to **~320 RPS**, but beyond that it **clogs**—wait times shoot up and throughput can even **fall**. To push it higher, I'd typically add **more service replicas**, **increase connection pool sizes**, and **cache** the hottest lookups.

---

6. AI Integration

AI Tools and Help

Prompt A: Project Architecture & Design

"I need to implement a distributed URL shortener with two architectures: microservices (HTTP/REST) and layered (gRPC). Explain the key concepts I should understand and provide a roadmap for implementing both. The system should have 5 functional requirements: URL shortening, redirection, rate limiting, analytics, and expiration. Each architecture must run on 5 containerized nodes."

Prompt B: File Structure & Organization

"What is the optimal file structure for a project with two different architectures (microservices and layered)? I want to share common libraries between both, have separate deployment configurations, and organize the code cleanly."

Prompt C: grpcurl Tool

"Show me the right way to use grpcurl with my Dockerized system. I need: - How to run it without installing locally (containerized method) - How to run it inside my app container - Correct flags (-plaintext, -import-path, -proto, -d) for my layered architecture via NGINX - Windows shell differences (Git Bash vs PowerShell) - Typical errors and how to fix them (e.g., command not found, DNS, rate limit, port conflicts)"

What AI Responded

Reply to Prompt A - Architecture Concepts & Roadmap

Core Concepts: - Microservices pattern: Decompose into independent services (API Gateway, Redirect, Analytics, Rate Limit, Redis); each service owns its deployment lifecycle

Layered pattern: 3-tier architecture (Presentation → Business Logic → Data Access) all in single application; strict layer separation with function calls - Communication models: HTTP/REST + JSON for microservices (loose coupling, human-readable); gRPC + Protocol Buffers for layered (high performance, type-safe)

5 Functional Requirements Implementation: - FR1 (URL shortening): Cryptographic code generation with Base62 encoding - FR2 (Redirection): HTTP 301 with atomic click counting via Lua scripts - FR3 (Rate limiting): Sliding window algorithm using Redis Sorted Sets - FR4 (Analytics): Real-time leaderboard with Redis Sorted Sets (ZINCRBY) - FR5 (Expiration): TTL-based (Redis EXPIRE) and click-based (Lua atomic decrement)

Reply to Prompt B - File Structure & Organization

Recommended Structure:

Reply to Prompt C - grpcurl Tool

Common Errors and Solutions:

- **grpcurl: command not found**
  grpcurl is not a Python package; you can't pip install grpcurl. Use the container image or install the binary in your image.
- **lookup nginx on 127.0.0.11:53: no such host**
  You're not on the same network. Add –network or run inside the Compose service.
- **Bind for 0.0.0.0:8081 failed: port is already allocated**
  Another process/container is using 8081 on the host. Stop it or change your host port mapping.
- **HTTP 404 / 410 on Resolve**
  Code is expired (TTL or max-clicks). Recreate with large ttl_sec and max_clicks.
- **HTTP 429 in microservices**
  Rate limiter is throttling you under load. Rotate client IPs via X-Forwarded-For in tests or temporarily raise limits.
- **"TLS handshake" / gRPC transport errors**
  You're missing -plaintext (or pointing at the wrong endpoint). For the layered NGINX proxy, use plaintext to nginx:8081.

---

7. Contribution

Contribution of Each Teammate

Abhijit: - Microservices: Wrote the experimental setup, results (k6 runs at 50/100/200 VUs), figures, and system trade-offs for the microservices/Gateway path - Docker Hub: Built and tagged images, pushed to Docker Hub, updated compose references - GitHub push: Organized commits/PRs for microservices code and report assets; maintained repo structure and version tags

Chaitanya: - Layered: Wrote the experimental setup, results (ghz runs at 100/200/400 RPS), figures, and system trade-offs for the layered (gRPC) path - GitHub push: Committed layered service code, ghz scripts/reports (ghz-100/200/400.html), and integrated plots into the repo

---

8. References

1. Google. (2024). gRPC Documentation. https://grpc.io/docs/
2. Redis Labs. (2024). Redis Documentation. https://redis.io/documentation
3. FastAPI. (2024). FastAPI Documentation. https://fastapi.tiangolo.com/