### CSE 5306 - Distributed Systems Programming Assignment 2

## Distributed URL Shortener Service

**Team Members:**  
Abhijit Challapalli, Chaitanya Krishna Namburi

**GitHub Repository:**  
[GitHub Link](https://github.com/AbhijitChallapalli/CSE-5306-DS-PA2-URLShortener)

**Instructor:**

**Dr. Jiayi Meng**

## Table of Contents

1. Functional Requirements
2. Proposed System Architectures
3. Architecture 1: Microservices (HTTP/REST)
4. Architecture 2: Layered (gRPC)
5. Evaluation
6. AI Integration
7. Contribution
8. References

## 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)

┌─────────────────────────────────────────────┐  
│ Users / Internet │  
└──────────────────┬──────────────────────────┘  
 │ HTTP/REST + JSON  
 ▼  
 ┌──────────────────────┐  
 │ API Gateway │ ← Port 8080  
 │ (Node 1) │  
 │ │  
 │ • Route requests │  
 │ • Extract client IP │  
 │ • Aggregate results │  
 └──────────┬───────────┘  
 │  
 ┌─────────────┼─────────────┬──────────────┐  
 │ │ │ │  
 ▼ ▼ ▼ ▼  
┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐  
│Redirect │ │Analytics│ │ Rate │ │ Redis │  
│Service │ │Service │ │ Limit │ │Database │  
│(Node 2) │ │(Node 3) │ │(Node 4) │ │(Node 5) │  
│Port:8001│ │Port:8002│ │Port:8003│ │Port:6379│  
│ │ │ │ │ │ │ │  
│• Create │ │• Track │ │• Check │ │• Store │  
│ codes │ │ clicks │ │ IP rate│ │ all │  
│• Resolve│ │• Top │ │• Sliding│ │ data │  
│ URLs │ │ links │ │ window │ │ │  
└────┬────┘ └────┬────┘ └────┬────┘ └─────────┘  
 │ │ │ ▲  
 └─────────────┴─────────────┴─────────────┘  
 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)

A screenshot of a computer

AI-generated content may be incorrect.

DockerHub: <https://hub.docker.com/repositories/abchalla>

## 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)

A screenshot of a computer

AI-generated content may be incorrect.

Dockerhub: <https://hub.docker.com/repositories/abchalla>

## 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

**A close-up of a white background

AI-generated content may be incorrect.**

**What did the results say**

Microservices (HTTP via gateway, k6)

* 50 users → ~271 RPS; p95 ≈ 245 ms (miss vs 200 ms), p99 ≈ 285 ms (meets).
* 100 users → ~320 RPS (flat vs 50 → clear ceiling); p95 ≈ 332 ms (miss), p99 ≈ 433 ms (meets).
* 200 users → ~94 RPS; p95 ≈ 2.5 s, p99 ≈ 2.6 s → heavy queuing/saturation.

**Performance & Scalability :**

**Microservices (HTTP via API Gateway)**

**Interpretation (Performance)**

* **50 VUs:** ~**271 RPS**; **p95 ≈ 245 ms** (slight SLO miss vs 200 ms), **p99 ≈ 285 ms** (meets 500 ms).
* **100 VUs:** ~**320 RPS**; **p95 ≈ 332 ms** (SLO miss), **p99 ≈ 433 ms** (meets 500 ms). Throughput is **flat vs 50 VUs** → clear ceiling.
* **200 VUs:** ~**94 RPS**; **p95 ≈ 2.5–2.6 s**, **p99 ≈ 2.6 s** → heavy queuing / saturation.

**Scalability takeaway (updated)**

* The **knee** is still around **~300–320 RPS**.
* On this run, **p95 at 50 VUs** drifted above 200 ms, so the **SLO-compliant capacity** is **just under ~300 RPS** (vs. ~320 RPS in your earlier run).

**Bottom line:** the microservices path still **hits a knee around ~300–320 RPS**. On this run the 50-VU point drifted above the 200-ms p95 SLO, so **SLO-compliant capacity is just under ~300 RPS**. Past the knee, tail latencies spike and throughput drops.

Test results for 50 Virtual users with duration 60 seconds

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Grafana Dashboard:

A screenshot of a computer

AI-generated content may be incorrect.

#100 Virtual users with duration 60 seconds

A computer screen with text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Grafana Dashboard:



A screenshot of a computer

AI-generated content may be incorrect.

# 200 Virtual users with duration 60 seconds

A computer screen with text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Grafana Dashboard:

A screenshot of a computer

AI-generated content may be incorrect.

**All the dashboards and the html files are pushed to github**

**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 |
|  |  |  |  |  |  |  |  |  |  |  |

**A screenshot of a white background

AI-generated content may be incorrect.**

**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)

A screenshot of a data report

AI-generated content may be incorrect.

200 Requests Per Second

A screenshot of a data report

AI-generated content may be incorrect.

400 Requests Per Second

A screenshot of a computer screen

AI-generated content may be incorrect.

### 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.

A screenshot of a phone

AI-generated content may be incorrect.

**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/