

Design Leetcode

Members: Tuan Le Hoang - Hoang Long Le

Agenda

1. Clarify Requirements
2. Back of the Envelope Estimation
3. API Design
4. Data Model Design
5. High-level Design
6. Detailed Design
7. Identify and Discuss Bottle necks

1. Clarify Requirements

Functional Requirements

- View Problems
- Submit Solutions in multiple languages
- Join Coding Contests

Non-Functional Requirements

- Availability: 99.9% uptime
- Scalability: 10K+ current submissions
- Latency: Code execution < 10s, leaderboard update: < 5s
- Security: Isolated code execution, prevent malicious code

Out of Scope

- User Authentication
- Payment processing
- User analytics / User management

2. Back of the Envelope Estimation

- Daily Active Users: 500,000
- Contest participants: 10,000
- Problems in DB: 3000+, growth: 8 problems/week => 418 problems/year
- Leetcode has 26.3 million monthly visitors
=> $26.3 * 10^6 / 30 / 86000 = 10$ QPS
- Support 10+ popular programming Languages: Python 3, python 2, Java, C++, C, C#, C, Javascript, Typescript, Go, Swift, Rust, PHP, Kotlin...

Submission Estimation

Daily submissions:

$500,000 \text{ DAU} * 3 = 1.5 \text{ M submissions/day}$

$\Rightarrow 1.5 * 10^6 / 86,000 = 17 \text{ submissions/second}$

$\Rightarrow 2 \text{ submissions/language/second}$

Peak submissions (contest):

$10,000 * 20 \text{ submissions} = 200,000 \text{ in 90 minutes}$

$\Rightarrow 2 * 10^5 / 5400 = 37 \text{ submissions/second}$

$\Rightarrow 4 \text{ submissions/language/second}$

Storage Estimation

Per submission:

Code: ~10 KB

Metadata: ~1KB

Results: ~2 KB

=> Total: ~13 KB

=> Daily storage: $1.5 \text{ M} * 13 \text{ KB} = 20 \text{ GB/day}$

=> Monthly storage growth: 600 GB/month

Problems + Test cases:

$3,000 * 50 \text{ test cases} * 10 \text{ KB} = 1.5 \text{ GB}$

Bandwidth Estimation

Incoming (submissions):

$1.5 \text{ M} * 10 \text{ KB} = 15 \text{ GB/day}$

Outgoing

Problem views: $10 \text{ M} * 50 \text{ KB} = 500 \text{ GB/day}$

Results: $1.5 \text{ M} * 5 \text{ KB} = 7.5 \text{ GB/day}$

=> Total: $\sim 520 \text{ GB/day} \approx 6 \text{ MB/s}$ average

3. API Design

Problem APIs

```
GET /problems?page=1&limit=100  
-> Partial<Problem>[] : [{id, title, difficulty, tags...}]
```

```
GET /problems/{problem_id}  
-> {id, title, description, examples, constraints, starter_code, difficulty, tags, acceptance_rate...}
```

```
GET /problems/{problem_id}/solutions?sort=votes&page=1  
->
```

Submission APIs

```
POST /problems/{problem_id}/run
```

```
Body: {code, language, test_input}
```

```
-> {results, runtime_ms, memory_kb}
```

```
POST /problems/{problems_id}/submit
```

```
Body: {code, language}
```

```
-> {submission_id}
```

```
GET /submissions/{submission_id}
```

```
-> {status: "pending" | "running" | "accepted" | "wrong_answer", results, runtime_ms, memory_kb}
```

Contest APIs

GET /contests

-> {upcoming: [...], ongoing: [...], past: [...]}

GET /contests/{contest_id}

-> {id, title, start_time, end_time, problems: [...]}

POST /contests/{contest_id}/register

-> {success: true}

GET /contests/{contest_id}/leaderboard?page=1&limit=50

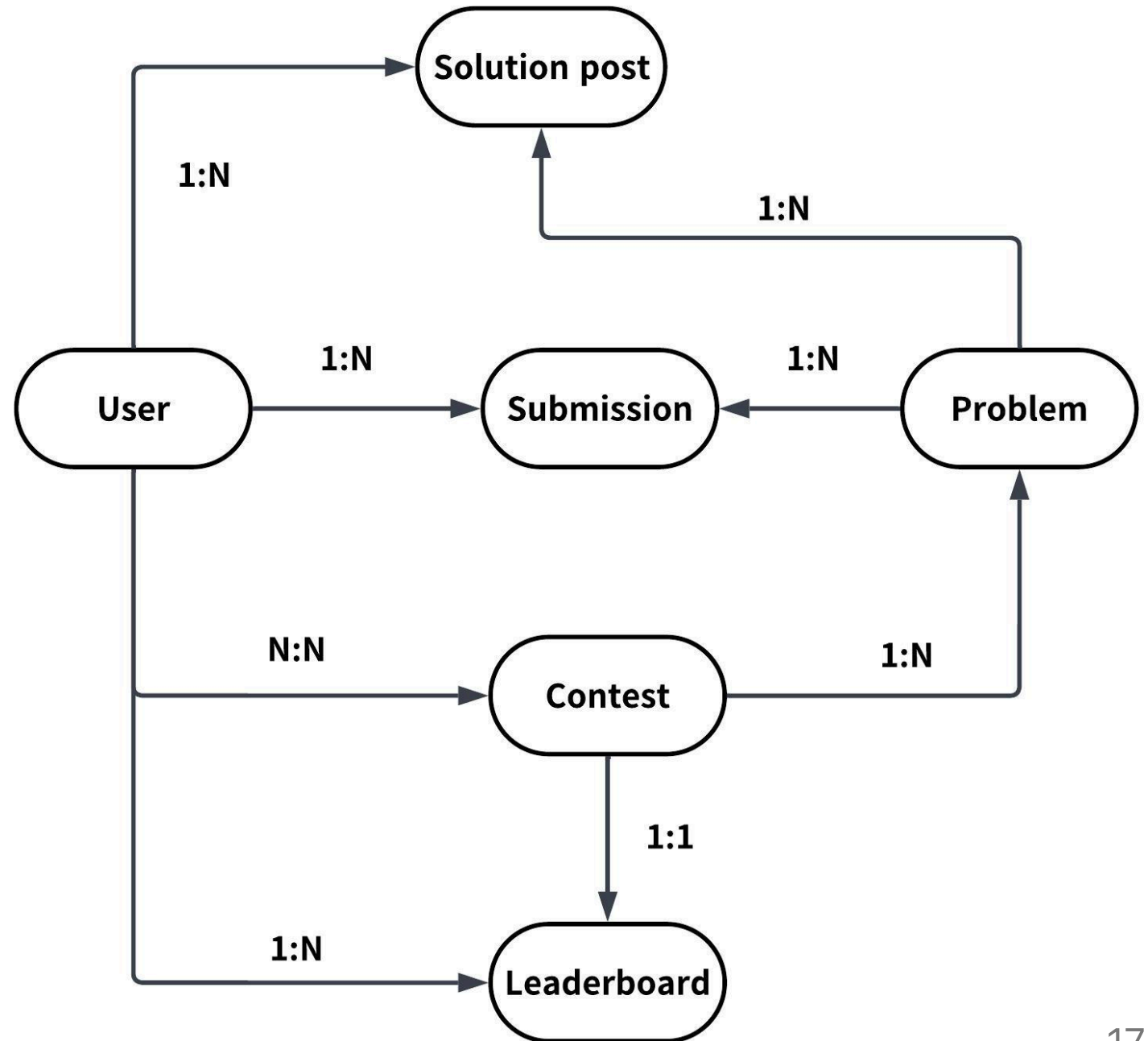
{rankings: [{rank, user, score, finish_time}]}

User APIs

GET: profile, submissions, progress.

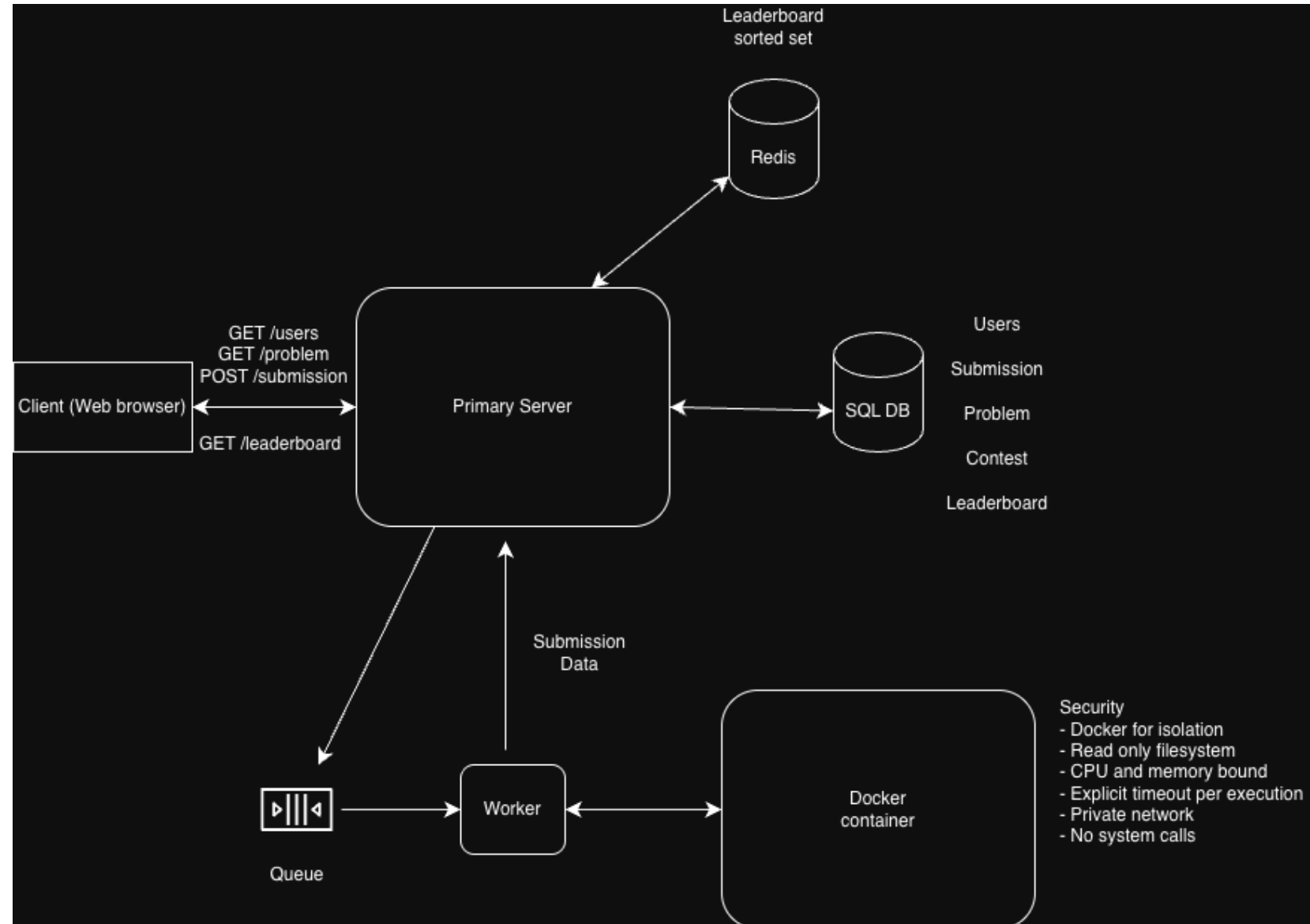
4. Data Model Design

Entity Relationship Diagram



5. High Level Design

System Architecture



FR1: View Problems

1. Client sends `GET /problems` or `GET /problems/{id}` to Primary Server
2. Server queries SQL DB for problem data
3. Returns problem list or details to client

FR2: Submit Solutions

1. Client sends `POST /submission` with code and language
2. Primary Server pushes submission to **Queue** (async processing)
3. **Worker** pulls from queue, executes code in **Docker Container**
4. Worker sends results back to Primary Server
5. Server stores results in SQL DB and returns to client via polling `GET /submissions/{id}`

FR3: Join Coding Contests

1. Client polls `GET /leaderboard` every 5 seconds
2. Primary Server fetches rankings from **Redis Sorted Set** (fast $O(\log N)$)
3. On submission completion, Worker updates both SQL DB and Redis cache
4. Returns real-time leaderboard to client

6. Discussion

Q1: How to ensure isolation and security when running user code?

Solution: Docker Containers with Security Configurations

- **Read-Only Filesystem:** Mount code directory as read-only, use temp directory for output
- **CPU & Memory Limits:** Kill container if limits exceeded, prevent resource exhaustion
- **Explicit Timeout:** 5-second limit to prevent infinite loops
- **Network Isolation:** Disable network access using VPC Security Groups/NACLs
- **Seccomp:** Restrict system calls to prevent host compromise

Q2: How to make leaderboard fetching more efficient?

Solution: Redis Sorted Sets + Polling

- Store leaderboard in Redis sorted set: `competition:leaderboard:{contestId}`
- Score = user's total score/solve time, Value = userId
- Update Redis on each submission: `ZADD competition:leaderboard:{contestId} {score} {userId}`
- Retrieve top N users: `ZREVRANGE ... 0 N-1 WITHSCORES` ($O(\log N)$)
- Client polls every 5 seconds - simpler than WebSockets, acceptable latency
- Reduces database load significantly

Q3: How to scale for 100K concurrent users during contests?

Solution: Queue-based Horizontal Scaling

- Add **message queue** (SQS) between API server and workers
- Buffer submissions during peak times, prevent container overload
- Workers pull and process submissions independently
- **Async flow**: API returns immediately with `submission_id`
- Client polls `GET /submissions/{id}` every second for results
- Enables retries on container failures
- Fun fact: This is exactly how LeetCode works! (Check network tab)

Q4: How to handle running test cases across multiple languages?

Solution: Standardized Serialization + Language-Specific Test Harness

- Write **one set of test cases per problem** (language-agnostic)
- Serialize inputs/outputs in standard format (e.g., JSON arrays for trees)
- Each language has a **test harness** that:
 - Deserializes standardized input
 - Passes to user's code
 - Compares output with expected result
- Example: Tree input `[3,9,20,null,null,15,7]` → deserialized to `TreeNode` object