Design LeetCode

System Design Interview

Agenda

1. Clarify Requirements
2. Back of the Envelope Estimation
3. API Design / Flow Design
4. Data Model Design
5. High-Level Design
6. Detailed Design
7. Identify and Discuss Bottlenecks

Step 1: Clarify Requirements

Functional Requirements

Feature	Description
View Problems	Browse problem list, view details, examples, constraints
Submit Solutions	Submit code in multiple languages, run against test cases
Run Code	Test code with custom input before submitting
Coding Contests	Timed events (2 hours), real-time leaderboard
Discussion	Comment on problems, share solutions

Non-Functional Requirements

Requirement	Target
Availability	99.9% uptime (24/7 access)
Scalability	10K+ concurrent submissions
Latency	Code execution < 10s, Leaderboard update < 1s
Security	Isolated code execution, prevent malicious code
Consistency	Accurate results, fair contest ranking

Out of Scope

- User authentication/authorization details
- Payment system (Premium features)
- Social features (Friends, Following)
- Mobile app specifics

Step 2: Back of the Envelope Estimation

Traffic Estimation

Metric	Value
Daily Active Users (DAU)	500,000
Concurrent users (peak)	50,000
Contest participants	10,000
Problems in database	3,000+

Submission Estimation

```
Daily submissions:
  500,000 DAU × 5 submissions/user = 2.5M submissions/day

Peak submissions (contest):
  10,000 users × 20 submissions = 200,000 in 2 hours
  = 200,000 / 7,200 sec ≈ 28 submissions/sec
  Peak (10x burst): ~300 submissions/sec

Test case executions:
  2.5M submissions × 20 test cases = 50M executions/day
```

Storage Estimation

```
Per submission:
  - Code: ~10 KB
  - Metadata: ~1 KB
  - Results: ~2 KB
  Total: ~13 KB

Daily storage:
  2.5M × 13 KB = 32.5 GB/day

Monthly storage:
  32.5 GB × 30 = ~1 TB/month

Problems + Test cases:
  3,000 problems × 50 test cases × 10 KB = 1.5 GB
```

Bandwidth Estimation

```
Incoming (submissions):
  2.5M × 10 KB = 25 GB/day

Outgoing (results + problems):
  - Problem views: 10M × 50 KB = 500 GB/day
  - Results: 2.5M × 5 KB = 12.5 GB/day
  Total: ~515 GB/day ≈ 6 MB/s average
```

Step 3: API Design / Flow Design

Problem APIs

```
GET /api/v1/problems
    ?page=1&limit=20&difficulty=medium&tag=array
    → { problems: [...], total: 3000, page: 1 }

GET /api/v1/problems/{problem_id}
    → { id, title, description, examples, constraints,
        starter_code: { python, java, cpp, ... },
        difficulty, tags, acceptance_rate }

GET /api/v1/problems/{problem_id}/solutions
    ?sort=votes&page=1
    → { solutions: [...] }
```

Submission APIs

```
POST /api/v1/problems/{problem_id}/run
    Body: { code, language, test_input }
    → { output, expected, passed, runtime_ms, memory_kb }

POST /api/v1/problems/{problem_id}/submit
    Body: { code, language }
    → { submission_id }

GET /api/v1/submissions/{submission_id}
    → { status: "pending" | "running" | "accepted" | "wrong_answer",
        results: [...], runtime_ms, memory_kb,
        runtime_percentile, memory_percentile }
```

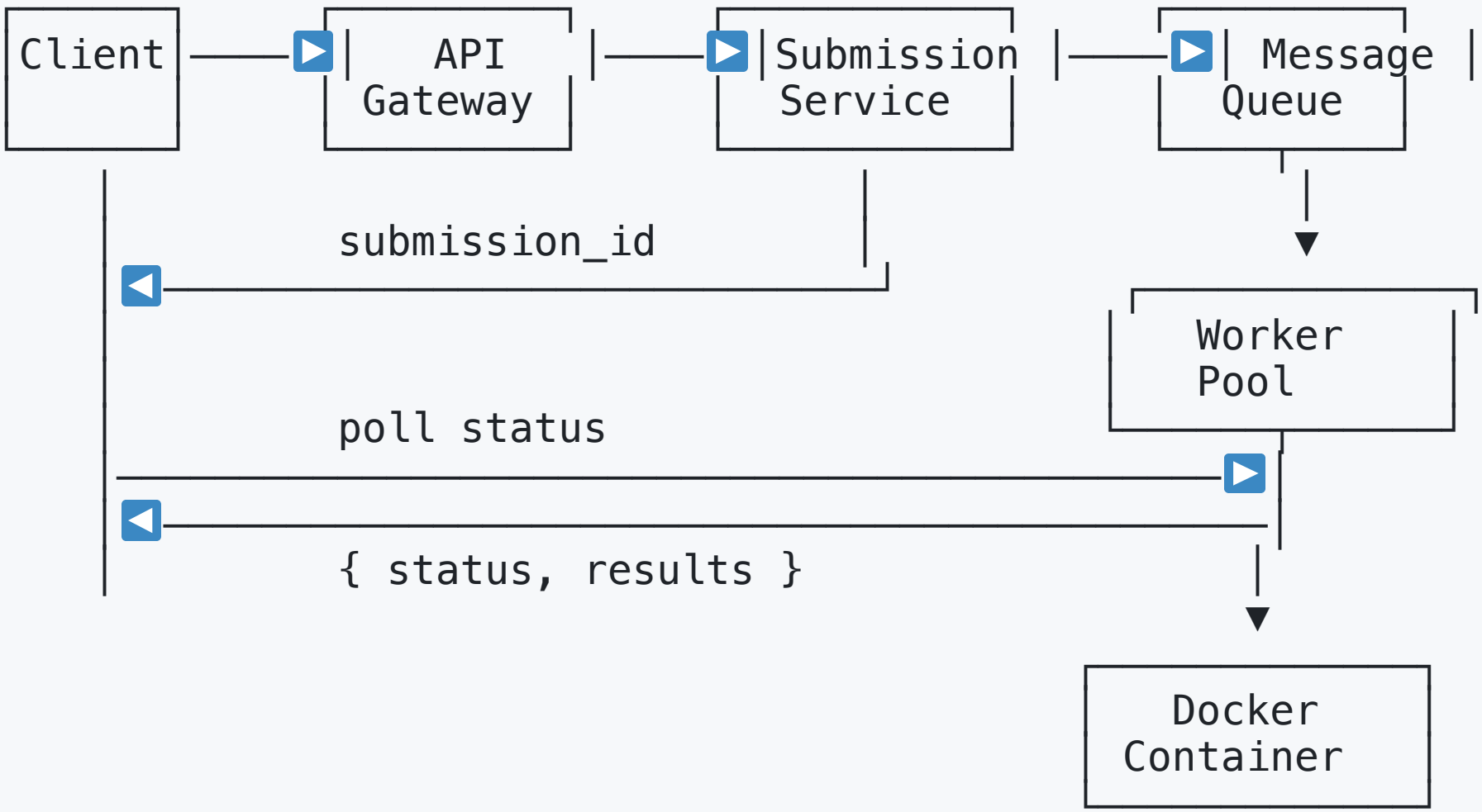
Contest APIs

```
GET /api/v1/contests
    → { upcoming: [...], ongoing: [...], past: [...] }

GET /api/v1/contests/{contest_id}
    → { id, title, start_time, end_time, problems: [...] }

POST /api/v1/contests/{contest_id}/register
    → { success: true }

GET /api/v1/contests/{contest_id}/leaderboard
    ?page=1&limit=50
    → { rankings: [{ rank, user, score, finish_time }] }
```

User APIs

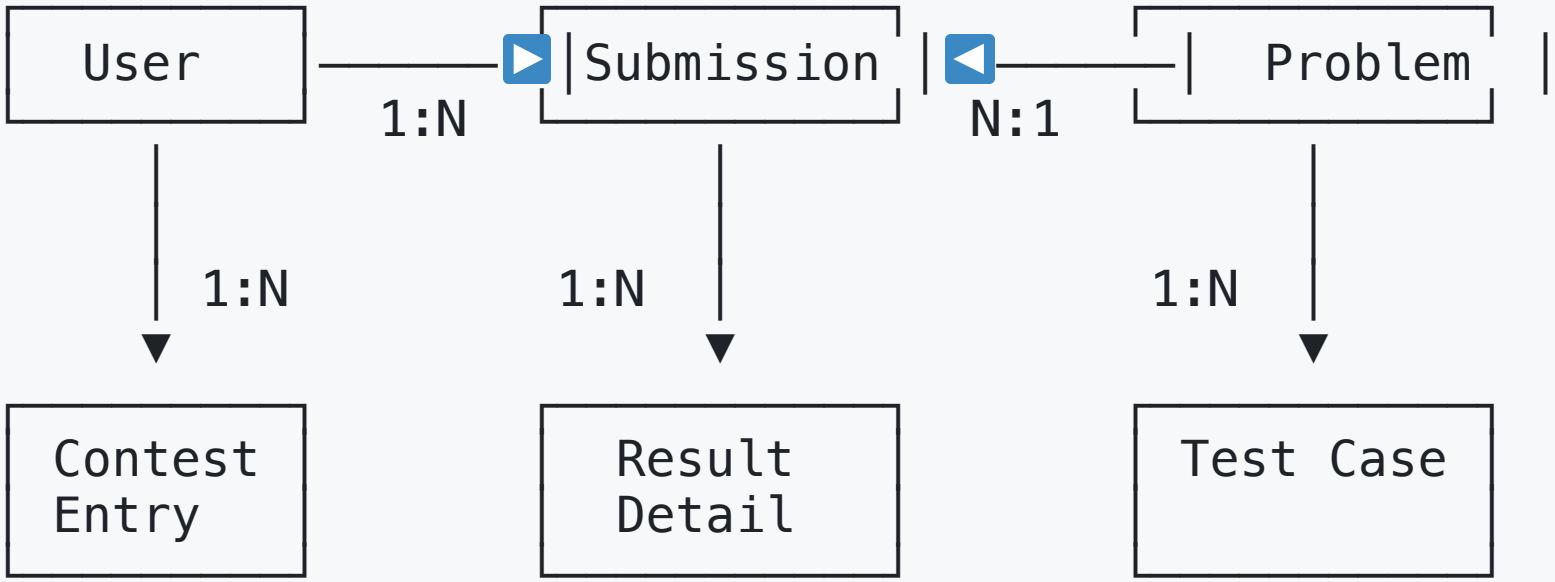
```
GET /api/v1/users/{user_id}/profile
    → { username, avatar, solved_count, ranking,
        submission_calendar, badges }

GET /api/v1/users/{user_id}/submissions
    ?problem_id=1&status=accepted
    → { submissions: [...] }

GET /api/v1/users/{user_id}/progress
    → { easy: 150/600, medium: 200/1300, hard: 50/550 }
```

Submission Flow Diagram



Step 4: Data Model Design

Entity Relationship Diagram



User Table

```sql
CREATE TABLE users (
    id              BIGSERIAL PRIMARY KEY,
    username        VARCHAR(50) UNIQUE NOT NULL,
    email           VARCHAR(255) UNIQUE NOT NULL,
    password_hash   VARCHAR(255) NOT NULL,
    avatar_url      VARCHAR(500),
    ranking         INT DEFAULT 0,
    solved_easy     INT DEFAULT 0,
    solved_medium   INT DEFAULT 0,
    solved_hard     INT DEFAULT 0,
    created_at      TIMESTAMP DEFAULT NOW(),
    updated_at      TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_users_ranking ON users(ranking);
```

Problem Table

```sql
CREATE TABLE problems (
    id              BIGSERIAL PRIMARY KEY,
    title           VARCHAR(255) NOT NULL,
    slug            VARCHAR(255) UNIQUE NOT NULL,
    description     TEXT NOT NULL,
    difficulty      VARCHAR(10) NOT NULL, -- easy/medium/hard
    acceptance_rate DECIMAL(5,2) DEFAULT 0,
    submission_count BIGINT DEFAULT 0,
    accepted_count  BIGINT DEFAULT 0,
    starter_code    JSONB NOT NULL, -- {python: "...", java: "..."}
    is_premium      BOOLEAN DEFAULT FALSE,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_problems_difficulty ON problems(difficulty);
CREATE INDEX idx_problems_slug ON problems(slug);
```

Test Case Table

```
CREATE TABLE test_cases (
    id              BIGSERIAL PRIMARY KEY,
    problem_id      BIGINT REFERENCES problems(id),
    input           TEXT NOT NULL,
    expected_output TEXT NOT NULL,
    is_hidden       BOOLEAN DEFAULT TRUE,
    order_index     INT NOT NULL,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_test_cases_problem ON test_cases(problem_id);
```

Submission Table

```sql
CREATE TABLE submissions (
    id              BIGSERIAL PRIMARY KEY,
    user_id         BIGINT REFERENCES users(id),
    problem_id      BIGINT REFERENCES problems(id),
    code            TEXT NOT NULL,
    language        VARCHAR(20) NOT NULL,
    status          VARCHAR(20) DEFAULT 'pending',
    -- pending, running, accepted, wrong_answer,
    -- time_limit, memory_limit, runtime_error, compile_error
    runtime_ms      INT,
    memory_kb       INT,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_submissions_user ON submissions(user_id);
CREATE INDEX idx_submissions_problem ON submissions(problem_id);
CREATE INDEX idx_submissions_status ON submissions(status);
CREATE INDEX idx_submissions_created ON submissions(created_at DESC);
```

Submission Results Table

```
CREATE TABLE submission_results (
    id              BIGSERIAL PRIMARY KEY,
    submission_id   BIGINT REFERENCES submissions(id),
    test_case_id    BIGINT REFERENCES test_cases(id),
    passed          BOOLEAN NOT NULL,
    actual_output   TEXT,
    runtime_ms      INT,
    memory_kb       INT,
    error_message   TEXT
);

CREATE INDEX idx_results_submission ON submission_results(submission_id);
```

Contest Tables

```sql
CREATE TABLE contests (
    id              BIGSERIAL PRIMARY KEY,
    title           VARCHAR(255) NOT NULL,
    start_time      TIMESTAMP NOT NULL,
    end_time        TIMESTAMP NOT NULL,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE contest_problems (
    contest_id      BIGINT REFERENCES contests(id),
    problem_id      BIGINT REFERENCES problems(id),
    order_index     INT NOT NULL,
    points          INT DEFAULT 100,
    PRIMARY KEY (contest_id, problem_id)
);
```

Contest Participation Table

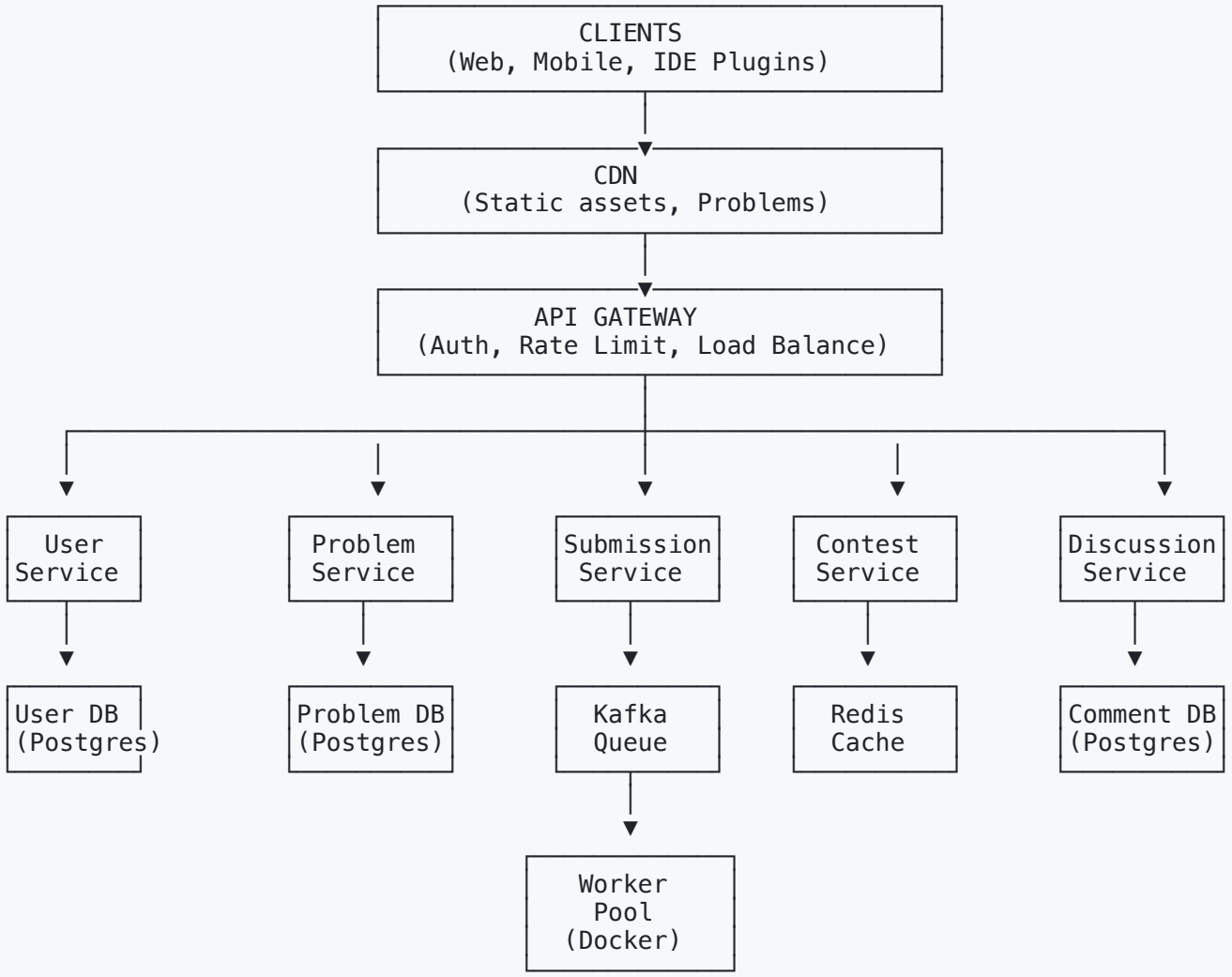
```sql
CREATE TABLE contest_entries (
    id              BIGSERIAL PRIMARY KEY,
    contest_id      BIGINT REFERENCES contests(id),
    user_id         BIGINT REFERENCES users(id),
    score           INT DEFAULT 0,
    finish_time     INT DEFAULT 0, -- seconds from start
    wrong_attempts  INT DEFAULT 0,
    ranking         INT,
    UNIQUE(contest_id, user_id)
);

CREATE INDEX idx_entries_contest_score
    ON contest_entries(contest_id, score DESC, finish_time ASC);
```

Redis Data Structures

```
# Leaderboard (Sorted Set)
ZADD contest:{id}:leaderboard {score} {user_id}
ZREVRANGE contest:{id}:leaderboard 0 49 WITHSCORES

# Problem statistics (Hash)
HSET problem:{id}:stats submissions 1000 accepted 500

# User session (String with TTL)
SET session:{token} {user_id} EX 86400

# Rate limiting (String with TTL)
INCR ratelimit:{user_id}:{endpoint}
EXPIRE ratelimit:{user_id}:{endpoint} 60
```

Step 5: High-Level Design

System Architecture



Service Responsibilities

Service	Responsibilities
User Service	Auth, profiles, progress tracking, rankings
Problem Service	CRUD problems, test cases, categories
Submission Service	Accept code, queue jobs, store results
Contest Service	Contest CRUD, registration, leaderboards
Discussion Service	Comments, solutions, votes
Worker Pool	Execute code in isolated containers

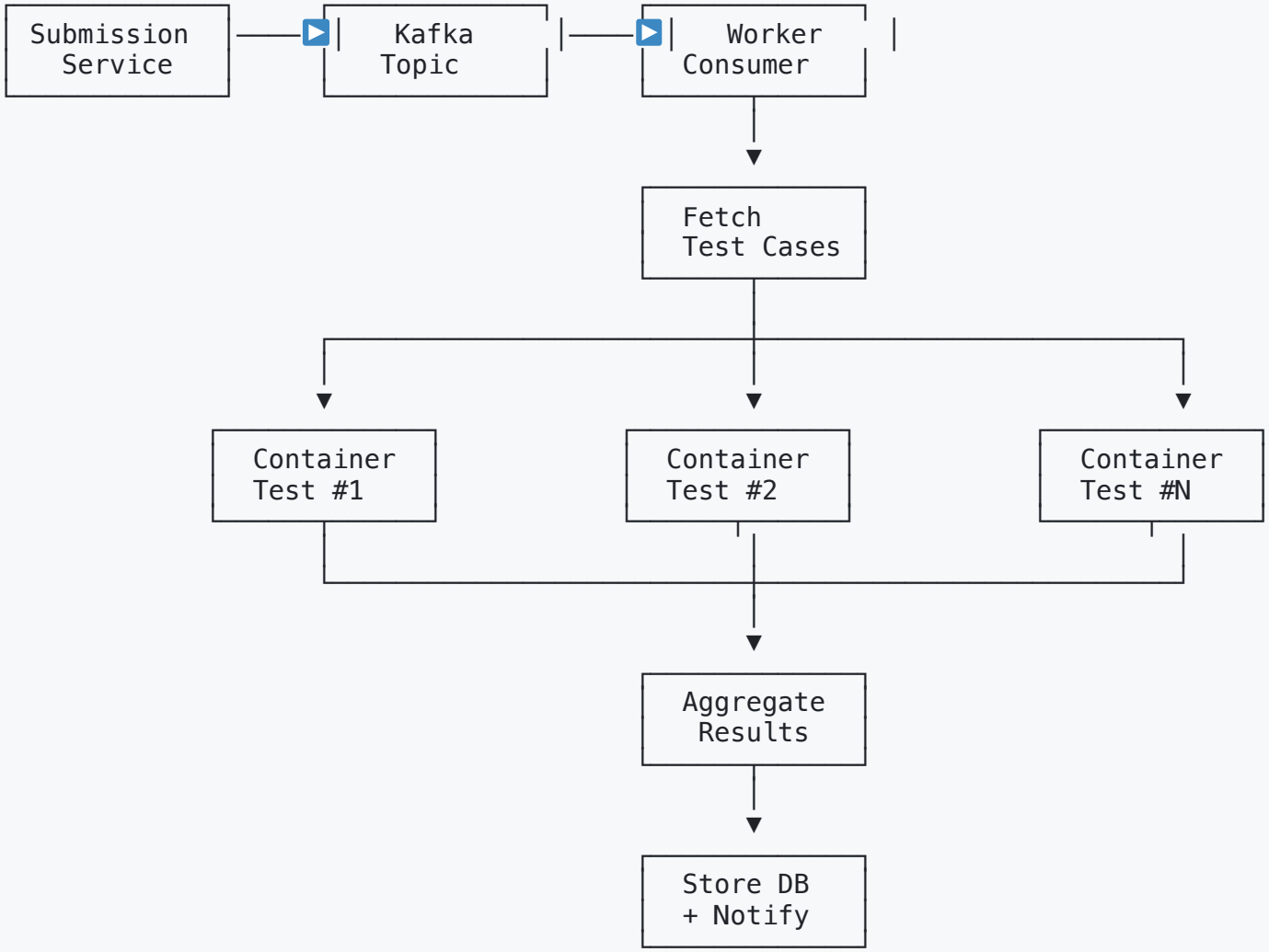
Technology Stack

Layer	Technology
Frontend	React/Next.js, Monaco Editor
API Gateway	Kong / AWS API Gateway
Backend	Go / Node.js microservices
Database	PostgreSQL (primary), Redis (cache)
Message Queue	Apache Kafka / AWS SQS
Container	Docker + Kubernetes
Storage	AWS S3 (code, test cases)
CDN	CloudFront / Cloudflare

Step 6: Detailed Design

6.1 Code Execution Engine

Execution Flow



Container Security

```
Docker Container Config:
  resources:
    limits:
      cpus: "0.5"
      memory: "512m"
    reservations:
      cpus: "0.25"
      memory: "256m"

  security_opt:
    - "no-new-privileges:true"
    - "seccomp:seccomp-profile.json"

  network_mode: "none"           # No network access
  read_only: true                # Read-only filesystem
  user: "nobody"                 # Non-root user

  tmpfs:
    - /tmp:size=64m,noexec       # Limited temp space
```

Language Support

```
Supported Languages:
  Python:
    image: python:3.11-slim
    compile: null
    run: "python3 solution.py"
    timeout: 10s

  Java:
    image: openjdk:17-slim
    compile: "javac Solution.java"
    run: "java Solution"
    timeout: 15s

  C++:
    image: gcc:12
    compile: "g++ -O2 -o solution solution.cpp"
    run: "./solution"
    timeout: 10s

  JavaScript:
    image: node:18-slim
    compile: null
    run: "node solution.js"
    timeout: 10s
```

6.2 Real-time Leaderboard

Redis Sorted Set Design

```
Score Formula:
  score = (solved_problems × 10000)
        + (max_time − finish_time)
        − (wrong_attempts × penalty)

Example:
  Solved: 3 problems      = 30000
  Time: 45 min (2700s)    = 7200 − 2700 = 4500
  Wrong: 2 attempts       = −2 × 300 = −600
  Final Score             = 33900
```

Leaderboard Operations

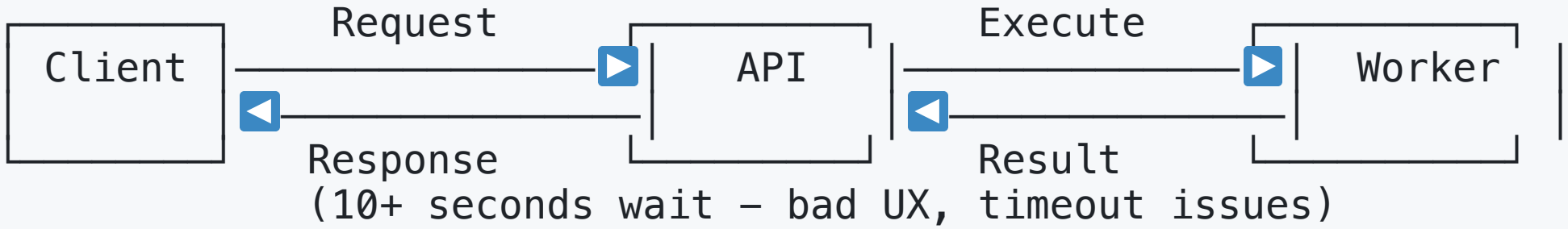
```
# Update score after submission
ZADD contest:123:leaderboard 33900 "user_456"

# Get top 50
ZREVRANGE contest:123:leaderboard 0 49 WITHSCORES

# Get user's rank (0-indexed)
ZREVRANK contest:123:leaderboard "user_456"

# Get users around a rank
ZREVRANGE contest:123:leaderboard 45 55 WITHSCORES

# Count total participants
ZCARD contest:123:leaderboard
```

Time Complexity: O(log N) for updates, O(log N + M) for range queries

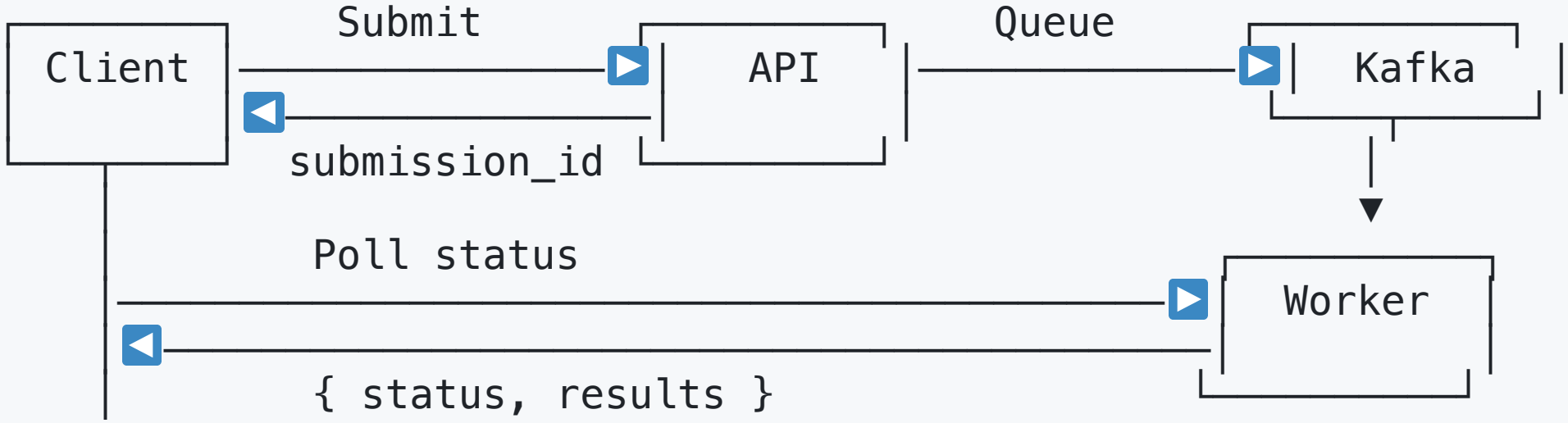
6.3 Async Processing Pattern

Why Async?

❌ Synchronous Approach:



✅ Asynchronous Approach:



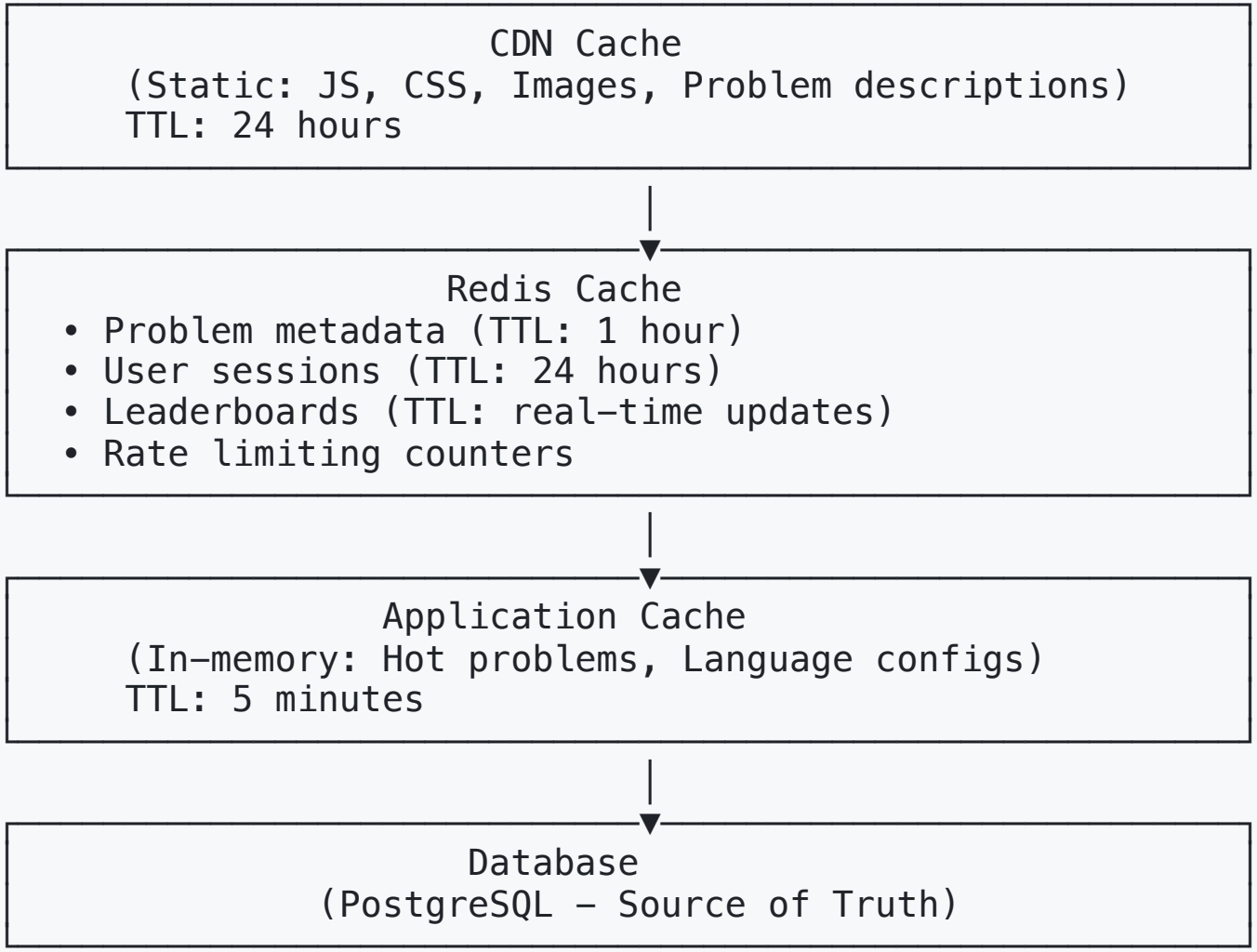
Polling vs WebSocket

Aspect	HTTP Polling	WebSocket
Complexity	Simple	Complex
Server Load	Predictable	Connection overhead
Latency	1-2 sec delay	Real-time
Use Case	Submission results	Live contest updates

Decision: HTTP Polling for submission status (simpler, sufficient)

6.4 Caching Strategy

Multi-Layer Caching



Cache Invalidation

```
Problem Update:
  1. Update PostgreSQL
  2. Delete Redis cache key
  3. Purge CDN cache
  4. Next request fetches fresh data

Leaderboard Update:
  1. Worker completes submission
  2. Directly update Redis ZSET
  3. Async sync to PostgreSQL (batch every 30s)

User Session:
  1. Login → Set Redis key with TTL
  2. Logout → Delete Redis key
  3. Auto-expire after 24 hours
```

Step 7: Identify and Discuss Bottlenecks

Potential Bottlenecks

Component	Bottleneck	Impact
Code Execution	Limited workers	Slow queue processing
Database	Write-heavy submissions	High latency
Contest Peak	Thundering herd	Service degradation
Leaderboard	Frequent updates	Redis hotspot
Test Cases	Large I/O	Slow execution

Bottleneck 1: Code Execution Scaling

Problem: Fixed worker pool can't handle burst traffic

Solutions:

```
1. Kubernetes Horizontal Pod Autoscaler (HPA)
   - Scale workers based on queue depth
   - Min: 10 workers, Max: 500 workers
   - Target: Queue depth < 100

2. Spot/Preemptible Instances
   - Use cheap compute for non-critical executions
   - Fallback to on-demand for contests

3. Priority Queues
   - High: Contest submissions
   - Medium: Regular submissions
   - Low: "Run" (test) requests
```

Bottleneck 2: Database Write Pressure

Problem: 300 submissions/sec during contests

Solutions:

```
1. Write-Behind Caching
   - Buffer writes in Redis
   - Batch insert every 5 seconds
   - Async worker handles persistence

2. Database Sharding
   - Shard submissions by user_id
   - Each shard handles subset of users

3. Time-Series Optimization
   - Partition submissions table by month
   - Auto-archive old partitions

4. Read Replicas
   - Route reads to replicas
```

Bottleneck 3: Contest Thundering Herd

Problem: 10K users submit simultaneously at contest end

Solutions:

```
1. Request Queuing
   - Accept all requests immediately
   - Queue processes in order
   - No dropped submissions

2. Rate Limiting per User
   - Max 1 submission per 5 seconds
   - Prevents spam, smooths traffic

3. Two-Phase Evaluation
   Phase 1 (Contest): Run 10% test cases → Quick feedback
   Phase 2 (After):   Run 100% test cases → Final verdict

4. Regional Distribution
   - Deploy workers in multiple regions
```

Bottleneck 4: Hot Leaderboard

Problem: 10K users polling leaderboard every second

Solutions:

```
1. Server-Side Caching
   - Cache top 100 for 1 second
   - Single Redis read serves all

2. Incremental Updates
   - Send only changed ranks
   - Reduce payload size 90%

3. Tiered Polling
   - Top 100: Poll every 2 seconds
   - Others: Poll every 10 seconds
   - Outside top 1000: Poll every 30 seconds

4. WebSocket for Top Users
   - Push updates to top 100
```

Bottleneck 5: Large Test Cases

Problem: Some problems have 100MB+ test cases

Solutions:

```
1. Test Case Streaming
   - Stream input to container
   - Don't load entire file in memory

2. Test Case CDN
   - Store test cases in S3
   - Workers fetch from nearest edge

3. Precompiled Binaries
   - Pre-compile test generators
   - Generate test cases on-the-fly

4. Compression
   - Compress test cases (gzip)
   - Decompress in container
```

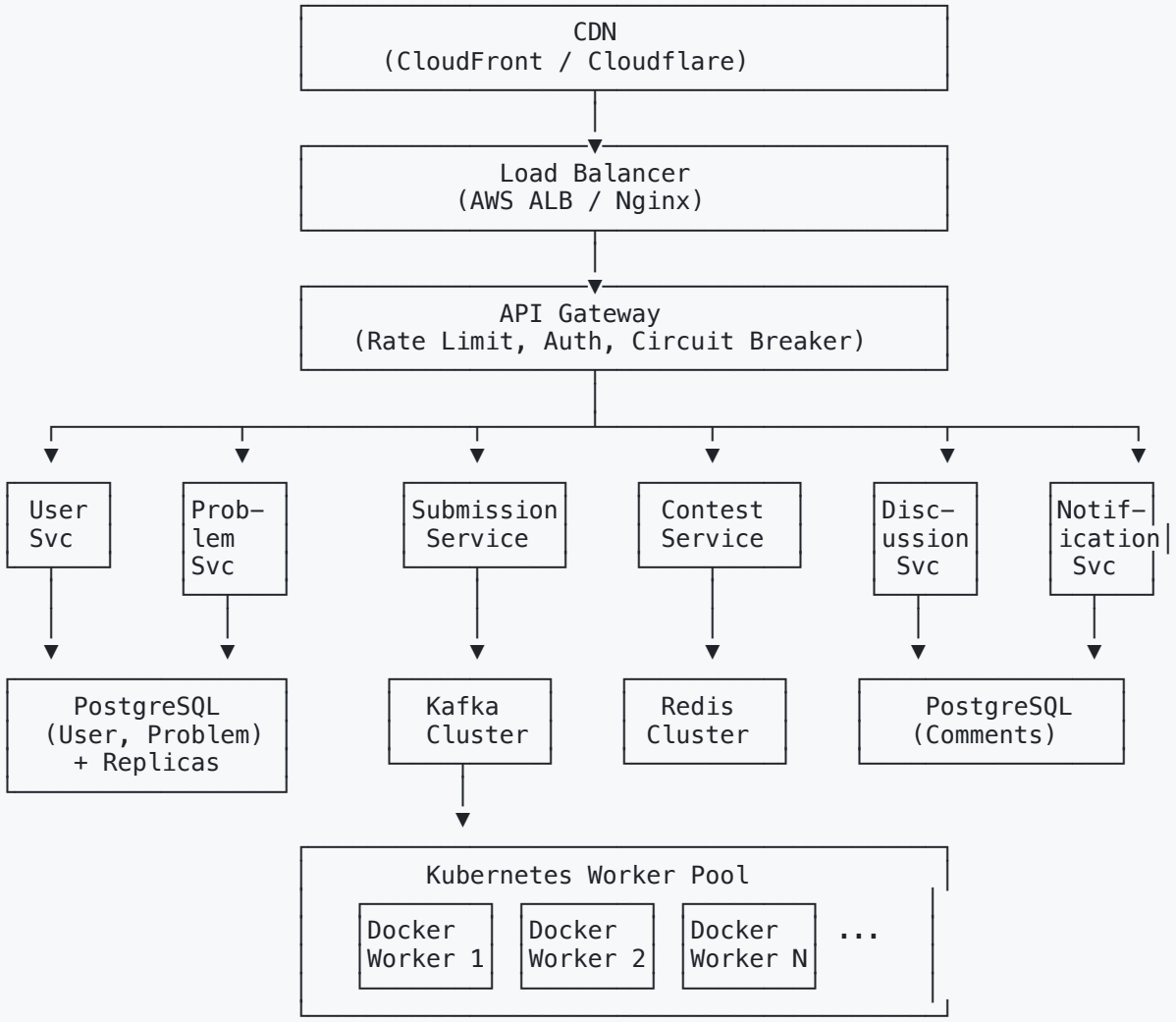
Monitoring & Alerting

```
Key Metrics:
  - submission_queue_depth      # Alert if > 1000
  - submission_latency_p99      # Alert if > 30s
  - worker_pool_utilization     # Alert if > 80%
  - database_connections        # Alert if > 80%
  - cache_hit_ratio             # Alert if < 90%
  - error_rate                  # Alert if > 1%

Dashboards:
  - Real-time submission throughput
  - Queue depth over time
  - Worker scaling events
  - Database query latency
  - Contest-specific metrics
```

Disaster Recovery

```
Backup Strategy:
  - PostgreSQL: Daily full + Hourly WAL
  - Redis: RDB snapshots every 15 min
  - S3: Cross-region replication

Failover:
  - Database: Automatic failover to standby
  - Redis: Sentinel-managed failover
  - Workers: K8s auto-restart failed pods

RTO/RPO:
  - RTO (Recovery Time): < 5 minutes
  - RPO (Data Loss): < 1 minute
```

Final Architecture



Summary

Step	Key Points
1. Requirements	View, Submit, Contest + HA, Scalability, Security
2. Estimation	300 TPS peak, 1TB/month storage
3. API Design	RESTful, async submission flow
4. Data Model	PostgreSQL + Redis, optimized indexes
5. High-Level	Microservices, Kafka queue, Docker workers
6. Detailed	Container security, leaderboard, caching
7. Bottlenecks	Auto-scaling, sharding, two-phase eval

References

- Hello Interview - Design LeetCode
- System Design School - LeetCode Solution
- Interviewing.io - Design LeetCode

Thank You!

Questions?