DESIGN ONLINE JUDGE (LEETCODE)

Core features of a online judge platform

- Platform that helps software engineers prepare for coding interviews
- vast collection of problems, easy → hard
- Users to answer questions, get feedback on their solutions
- Also run periodic contests

FR

- View list of problems
- View a given problem and code a solution
- submit solution & get feedback
- support contests w live leaderboard

NFR

- availability >> consistency
- security & isolation when running users code
- scale to support contests with 100K users
- fresh/new real-time leaderboard

Scale

- 100K DAU (5M total accounts)
- 3K problems
- 100K peak for competitions

Core Entities

- User
- Problems
- submission
- competition
- Leaderboard

APIs / Interfaces (satisfy our FR)

// view list of problems

query parameters (optional)

→ GET /problems?category={}&difficulty={}&page={}&size={}

↳ problems[]

returns list of Problems

I will minimize the payload by returning only the name/id.

// View a problem

→ GET /problems/:problemId (when things are required)

→ // submit solution

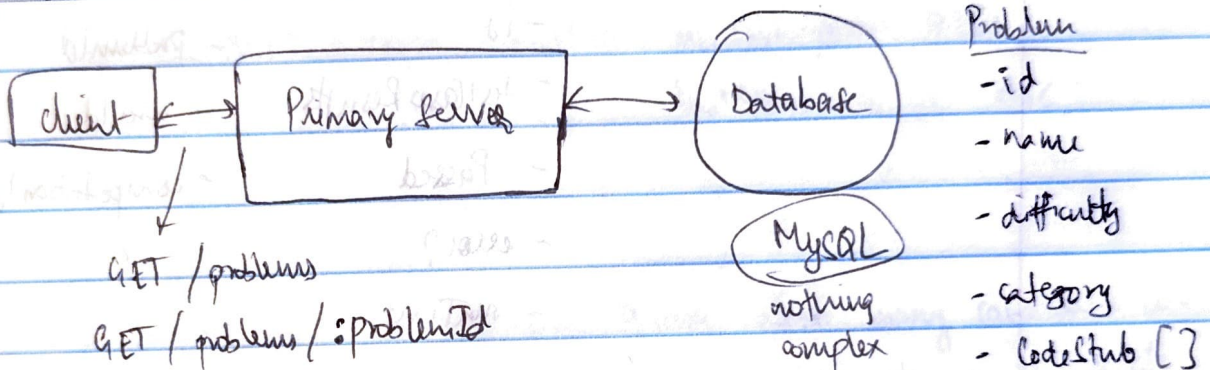
POST /problems/:problemId → submission

{
  code:
  language:
}

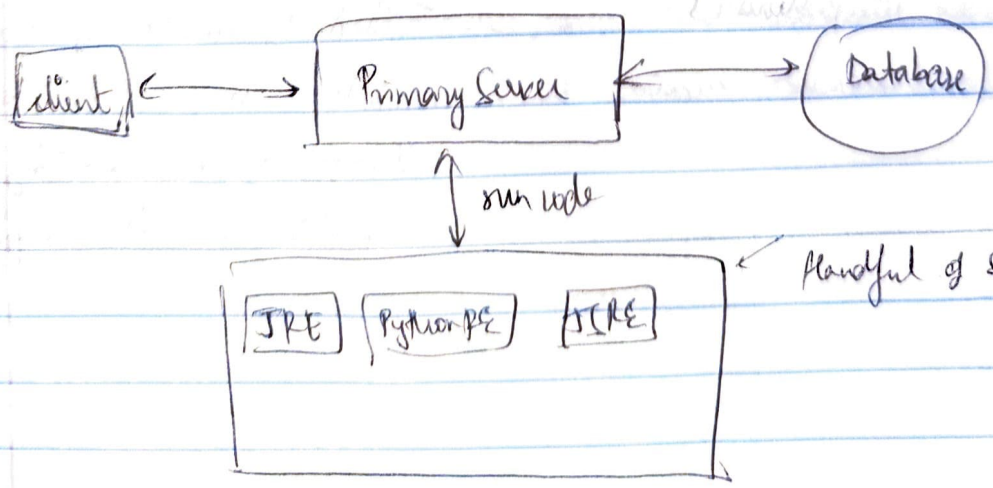
→ // Give leaderboard
GET /leaderboard/:competitionId?page={}&size={}
→ Leaderboard object is returned

High Level Design

first we will start like this -



* User code, do not run in a primary server which is connected to all of our models and services. (security concern)



Other options are to use serverless functions (AWS Lambda).
(they run and respond to events/triggers)
configured with our run-time environment.

→ But Lambda has cold start latency issues. (we can warm them up etc.)

Submission
- id
- testCaseResults
- Passed
- error?
- runTime
- problemId
- userId
- competetionId

Now for competetion, Ask interviewer, what is Competetion?

competition
- id
- startTime
- endTime
- problems []

- 90 mins
- 10 problems
- upto 100K users
- Rank by number of problems solved in the 90 mins.
- In case of tie rank by fastest time to complete.

Now GET /leaderboard/:id

we will do some query something like

Dont need to write this in the interview

```
SELECT userId, COUNT(*) AS passedSubmissions,
       MAX(submittedAt) AS lastSubmissionTime
FROM SUBMISSIONS
WHERE
    competetionId = {competetion_id} AND passed = TRUE
GROUPBY userId ORDERBY passedSubmissions DESC,
                       lastSubmissionTime ASC;
```

Would work but is a expensive query, when many candidates are trying to access & when there are so many records in the table.

Deep Dive (to satisfy the NFR)

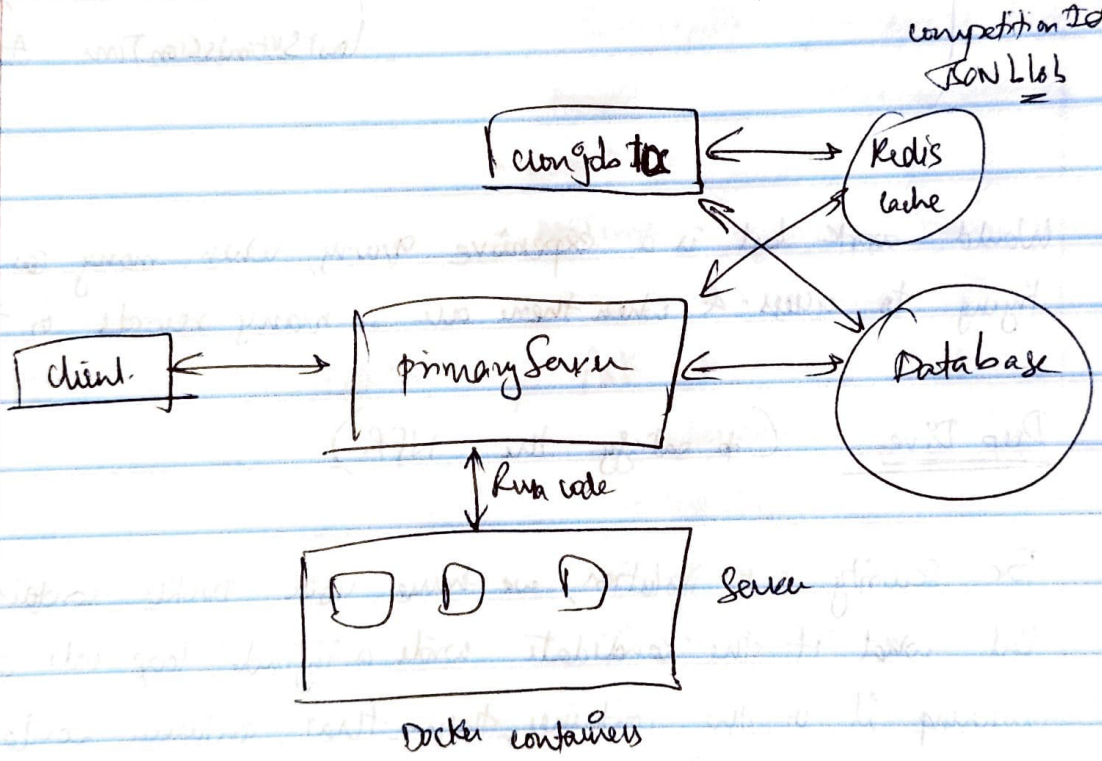
For security and isolation, we have used Docker containers etc. But what if the candidate wrote a infinite loop code and we are running it in the container, then that container would be forever pre-occupied and then wastes resources.

Security

- Explicit Timeout per execution
- CPU & memory bound as well.
- Read-only filesystem.

*,

→ for leaderboard latency issue,
we can use a Redis cache.
But we do not want them to hit the cache forever, as the leaderboard starts changing quickly.
So we will introduce a TTL, say 10s.
We have saved crazy lot of money & time.
But this can still be enhanced / optimized.



↳ Cron job to update the cache every 10s. So that every user now hits the cache.

But if cronjobs fails or something happens, then the cache is not updated.

In that case - users would get stale data & would overburden the DB suddenly. Also called the thundering herd problem.

What is Thundering herd problem

→ ~~Many requests~~ Cache miss / invalidation occurs say if many requests hits the cache but all values expire at once then all these requests simultaneously go to the backend to fetch data, overloading the backend.

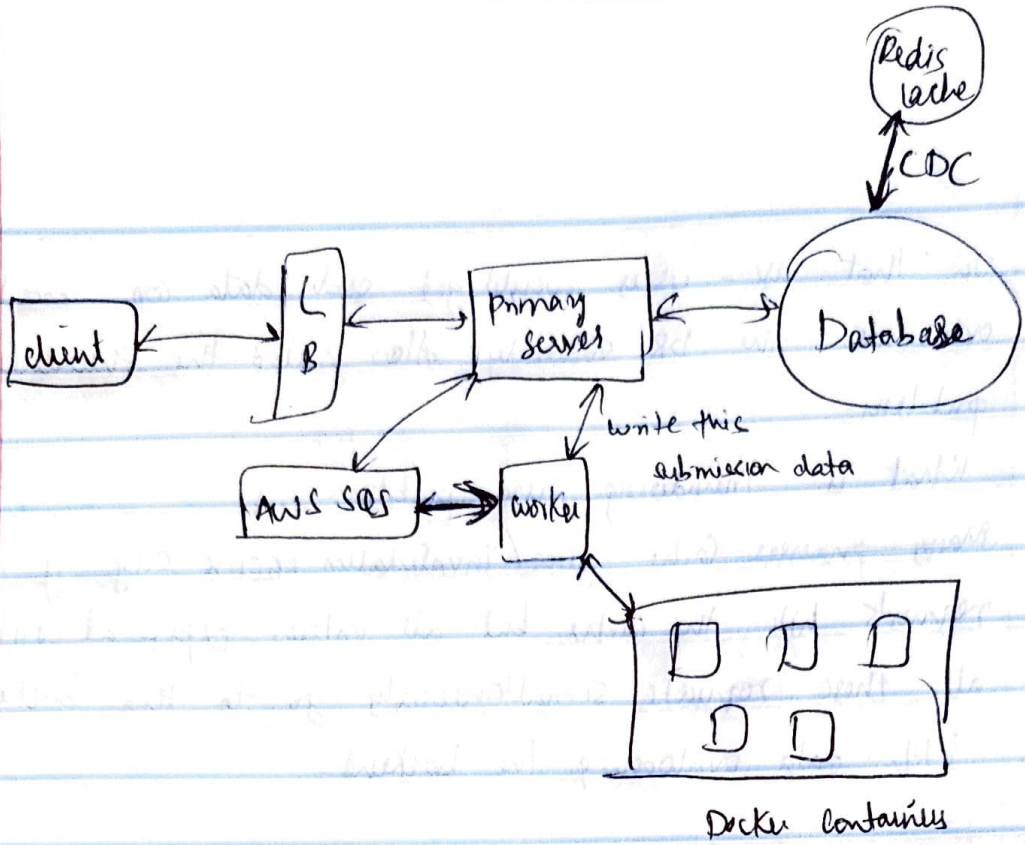
→ So the previous solution is a good option to answer, but not the best.

Can use websocket as well, ~~but~~ polling. (but feels like over-engineering).

→ Have Load Balancers b/w servers and all dont forget

→ We can horizontally scale the containers as well (ASG). Auto-scaling.

Towards end of the contest, there will be most number of submissions, and say Auto scaling would take a bit of time to configure new containers. So we can introduce a buffer.



→ As this is asynchronous, now how will the user know quickly about the result of their submission.
like say 2 sec wait time in SQS and 5sec to run in containers and display ≈ 10 sec → becomes slow.

→ Other options w.r.t cache

* Update cache on each write to DB. But not great as Redis is single threaded.

so, say I use a CDC (change data capture)
whenever datasource changes, CDC captures event-streams and puts into the queue, and some workers that pulls from the queue and writes to the cache.