Points to consider if the API server needs to support 1 million users accessing the live quiz at the same time.

Below I have mentioned a few bottlenecks and also some of my approaches on how to tackle scaling in such scenarios.

1. Fetching quiz for 1 million users

So a quiz can have anywhere from few to hundreds of questions associated with it, in such case a lot of READ queries will be executed just to fetch the same data. In order to reduce the total hits to the database, we can simply implement a caching strategy that will cache the entire quiz alongside the question as json data.

We can use Redis to cache the quiz data as key value pair and since a quiz can only be changed by the admin, and it's very unlikely that quiz will be changed during a live test is being conducted caching the quiz data is an optimum strategy since :-

1. It's not personalized and is same for all the users
2. It doesn't change frequently

2. Fetching stats related to question attempts

For every answer given by a user we are also providing stats related to that question's answers i.e how many people attempted it, what was their answer etc. These stats will change very frequently as more and more users progress in their quiz. And calculating new stats for every user is a very expensive operation.

In such a situation we can implement a short lived caching policy, in which the cache related to the stats of a question will be frequently updated some set amount of time which can be pre configured or determined by the traffic on the server at that moment, this process can be carried out by an asynchronous process, which will make the required queries to the database perform the computation and update the database. Alongside that we would need to make a separate uri for getting the stats unlike right now where it's being returned alongside the questions itself.

3. Database connection pooling

When such a huge number of users are hitting databases for write queries the application might exhaust the total number of connections it can open to the database. In such a case database connection pooling must be implemented to reuse open connections to the database effectively.

If we use a Postgresql database we can use pgbouncer for connection pooling, more information about it is available here. In case we are using MySql connection pooling api's are provided by Mysql as wrapper classes so we can use that.

4. Autoscaling

Almost all the cloud services provide load balancing with their elastic compute services in our case with AWS we can simply point our Elastic load balancer to the autoscaling group. And for determining scaling policy in ASG we can use the load balancer metrics of request per target.