The way I see it, the original code had one major flaw: it was trying to do too much with a single database query. It would pull every single piece of a driver's trip data and then force my own server to do all the calculations, like figuring out the total earnings and average rating. That approach is slow and just doesn't scale as the amount of data grows.

How I solve it:

At first, I implemented **connection pooling**. Before this, our system could only handle one connection at a time. With pooling, it can handle multiple connections at once, which is best for handling many users at the same time.

Next, I reorganize the database queries. Instead of one huge, inefficient request, I now use two separate, much smarter ones:

- **For the analytics,** the first query I wrote uses **COUNT**, **SUM**, and **AVG** directly in the SQL. This approach database does all the heavy lifting and just gives me the total trips, total earnings, and average rating. It's a huge performance boost because the database is way more efficient at this than my server is.
- **For the trip details:** Fetching every single trip at once was just not a scalable solution. So for this, I went with **cursor-based pagination**. The old **LIMIT/OFFSET** method is inefficient because the database has to scan the full data and get data from the right place. My new approach is much better. Because I pass a cursor, like id, and my database is already indexed, so it does not require a full scan, that's why it is much faster. Plus, by asking for **LIMIT + 1** trips, I can instantly tell if there's more data to fetch for the next page without a second request.

Trade-offs:

This new approach also has some trade-offs.

- **More Requests:** I've moved from one big request to two small, fast ones. While it's more network traffic, the two quick requests combined are faster than the single, slow one.
- **More Complex Queries:** The queries themselves are a bit more advanced because they use features like **GROUP BY** and pagination logic. But this complexity is necessary to move the hard work to the database.
- **Reduced Server Load:** By letting the database handle the aggregation, the server's workload is lower. This means it can handle way more users.

Ultimately, my new approach is much better than the previous one, because now its can handle multiple users and faster response time than the previous one.

Testing Strategy:

I used Grafana k6 for load testing.

Environment setup:

- I seed 1000 drivers, with each driver having 1000 trips, and each trip has payment and review, so the dataset is huge.
- I set 10000 virtual users with 2 minutes onboarding time and 10 minutes concurrent requests.
- I used Windows with CORE i5 8th gen, 16GB RAM, 238 GB storage. (With running Docker)

Result:

Un-optimized:

checks_total.....: 528180 (607.124834/s)

checks_succeeded.....: 1.51% 8004 out of 528180

checks_failed.....: 98.48% 520176 out of 528180

Request / Response time:

http_req_duration.....: avg=15.38s, min=0s , max=1m1s

http_res_duration.....: avg=30.61s, max=59.99s

iteration_duration.....: avg=21.37s, min=0s, max=1m24s

Conclusion:

It can`t sustain in heavy load, and it's not able to handle 10000 users.

Optimized:

checks_total.....: 528180 (607.124834/s)

checks_succeeded.....: 50.75% 268,051 out of 528180

checks_failed.....: 49.24%

Request / Response time:

http_req_duration.....: avg=33.22s min=0s, max=1m0s

http_res_duration.....: avg=44.39s, min=21.03ms, max=59.99s

iteration_duration.....: avg=37.64s, min=1.5ms, max=1m26s