## Architecture Discussion

So here's how I changed the system to make it faster and handle more traffic.

Instead of writing to the database directly every time a truck sends its location, I added a **queue and a batch worker**. Now the system works like this:

1. **API Layer:**
   When a truck sends location data to `/api/v1/location/update`, the API **doesn't write to the database immediately**.
   - It first checks that the data is correct.
   - Then it pushes it into a queue (`locationQueue`).
   - The API can respond **very quickly.**
2. **Queue:**
   The queue is like a buffer. It holds all incoming updates until the worker is ready.
   - We can handle thousands of updates at the same time.
   - This stops the database from being overloaded when many trucks send data at once.
3. **Batch Worker:**
   The worker keeps reading from the queue and collects updates into a batch.
   - Once the batch reaches a certain size (1000 updates) or a timeout (10 seconds), it does a **bulk insert** into the database.
   - This is much faster than inserting each update one by one.
4. **Stats & Monitoring:**
   I added a `/stats` endpoint to see things like:
   - How many updates are processed
   - How many batches ran
   - Queue length
   - Last batch time
   - This helps keep an eye on the system health.

## Why This Is Better

- **Less load on the database:** One bulk insert = 1000 updates at once instead of 1000 separate queries.
- **Faster API response:** The API doesn't wait for database writes, so users get a response almost immediately.
- **Can handle spikes:** Even if lots of trucks send updates at the same time, the queue absorbs the load.
- **Database is happier:** Bulk inserts are much easier for the database to handle than thousands of single inserts.

## Things to Watch Out For

- **Data loss if the system crashes:** If updates are still in the queue, they can be lost. Using a persistent queue (like Kafka) can fix this.
- **The queue can fill up:** If updates come in faster than the worker can process, some requests get rejected. Need monitoring and maybe more workers.
- **Small delay in database:** Updates are not immediately in the DB, can be up to 10 seconds behind.
- **Batch insert fails:** If a batch fails, those updates are lost unless we add retries.

## In Short

This system makes everything faster and more scalable. The API is quick, the database isn't overloaded, and the system can handle more trucks at the same time. Yes, there are some risks like data loss and queue overflow, but these can be solved with persistent queues and retries.

## Testing:

**Environment Setup:**

- I used Windows with CORE i5 8th gen, 16GB RAM, 238 GB storage. (With running Docker)
- I have 500 VUS (Virtual user), increasing from 50-500 in 4 mins, and 4 mins continues with 500 user requests, total of 10 mins.

**Un-optimized:**

checks_total.......................: 344779 (574.515918/s)

checks_succeeded...................: 62.88% 216,792 out of 344779

checks_failed......................: 37.11% 127,987 out of 344779

Request / Response:

http_req_duration.....................................................: avg=952.82ms, min=0s, med=421.35ms, max=10.81s

http_res_duration...........................................: avg=1.03s, min=12.07ms, med=401.8ms, max=9.99s

**Optimized**

checks_total.......................: 2753207 (5098.446324/s)

checks_succeeded...................: 90.88%  2502380 out of 2753207

checks_failed......................: 9.11%   250827 out of 2753207

Request/ Response:

http_req_duration.......................................................: avg=112.58ms, min=0s, med=99.27ms, max=1.06s

http_res_duration............................................: avg=101.09ms, min=0s,   med=87.74ms, max=1.06s

iteration_duration......................................................: avg=124.79ms, min=10ms, med=110.75ms, max=1.07s