Explanation of important choices in your solutions:

There were many choices in those applications, but I would underline only those that may seem fishy—and probably are!

Database layer.

Countries and currencies

I used the java libraries to handle all existing countries and currencies.
From my perspective, it's a bad practice. We lose flexibility when working with particular countries and currencies. There could be cases where:
- Country / Currency blocked, excluded, inactive, should be applied to the exchange rate.
- Currency can be crypto
- Default value of the currency is a BIG_decimal with the precision (16, 2), which will not work for Crypto, because one Satoshi(BITCOIN) 0,00000000001 -> such value is equal to x-amount of dollars, but cant be recorded, but its also not included in requirements.

Basically, countries and currencies cause a lot of questions from the business perspective, but I have put the SET / STRING values and applied validation.

Exceptions

More expectations to the god of the exceptions (same actually for tests)

There are always not enough tests. I used Global expectation handler and annotation error handling in DTO, but I would say they still need refinement and more systematization.

At the moment, they cover the business cases (TUUM test requirements), but me personally not satisfied

Test

Coverage is higher than requested, but I would say that I haven't calculated the time for tests properly, and I had to sacrifice their quality for the sake of finishing the application at the right time

MyBatis VS JPA/JDBC
I appreciate, that you asked to implement myBatis, but we will not find any JPA or JDBC library. They are compatible, but I didn't want to increase the complexity of the application And especially tests. So myBatis is responsible for the persistence layer.

OpenAPI
It hasn't been requested, but it's literally uncomfortable to develop and test without proper documentation tools. README.md has guidance on how to find the correct URL.

Docker-compose VS Local Build

Initially, I wanted to use several profiles, but I ran out of time and didn`t have time to properly set up application.test.yml and respective docker-compose.test.yml for intensive tests.
So, please use the document in the README.md file to run integration/unit tests properly and build the application.

Prometheus
Used for checking the load on the application during the stress tests. At the moment, it's suppressed with comment. If you would like, can be uncommented and reached.

Stress Test
JMeter was used for stress testing the application and simulating up to 10k requests per sec.

Describe what you have to consider to be able to scale applications Horizontally

Disclaimer: When I finally wrote that until the end, I understood it would be a bit overkill, but I'll keep that.

Firstly, let's set up the definition of **“scale horizontally”**.
From my perspective, it means:
- Adding more instances of application to spread the load
- Spread them via multiple nodes VS add RAM / CPU (vertical scaling)
- Create a proper pupping mechanism to synchronize, track, and manage nodes (Kubernetes, Docker Swarm, Terraform) + Internal mechanism of AWS / AZURE / Google Cloud
- Set of the application itself
- Separate nodes and consider them to be independent / self-reconstructing
- Use load balancers to mitigate load

If we go more high-level:
Applications that aim for horizontal scaling should be able to use effectively:
- Sessioning / Caching data to reduce load on the application node. In our case, it could be a Spring Session.
- Implement Load Balancers. It can be proposed by cloud platform solutions or more traditional ngNix of whatever, but these proxy gates should regulate routing, load, visibility, and security.
- Database node should be able to hold large load flow, properly support concurrency atomicity, being able to support multiple instances referring to the DB at the same time, probably even to the same resource, so some locking mechanism can be applied as well

Also, essential to ensure data consistency

One exciting thing that is actually sharding / partitioning of the db, especially effective, could be in Postgres. But I am not really sure how realistically effective that is, seen only on Medium posts.

- Excellent practice can also be considered when adding an application that is responsible for configuration/routing/visibility. Eureka? Or set up in Kubernetes

- Messaging Queues via RabbitMq / Kafka or others can also help create an event-driven, horizontally scalable architecture.

- Logging and monitoring. Would be a good practice to centralize the collection of logs from containers. Dynatrace / Kibana + Grafana

- Security is also important, but I am not the best expert in that. Definitely should be set up wardens / ket vaults / node security and limited port usages.

Honestly, it's all that I know, about how to build more or less stable applications.
Definitely will know more, but still need to read articles, books and take practice.

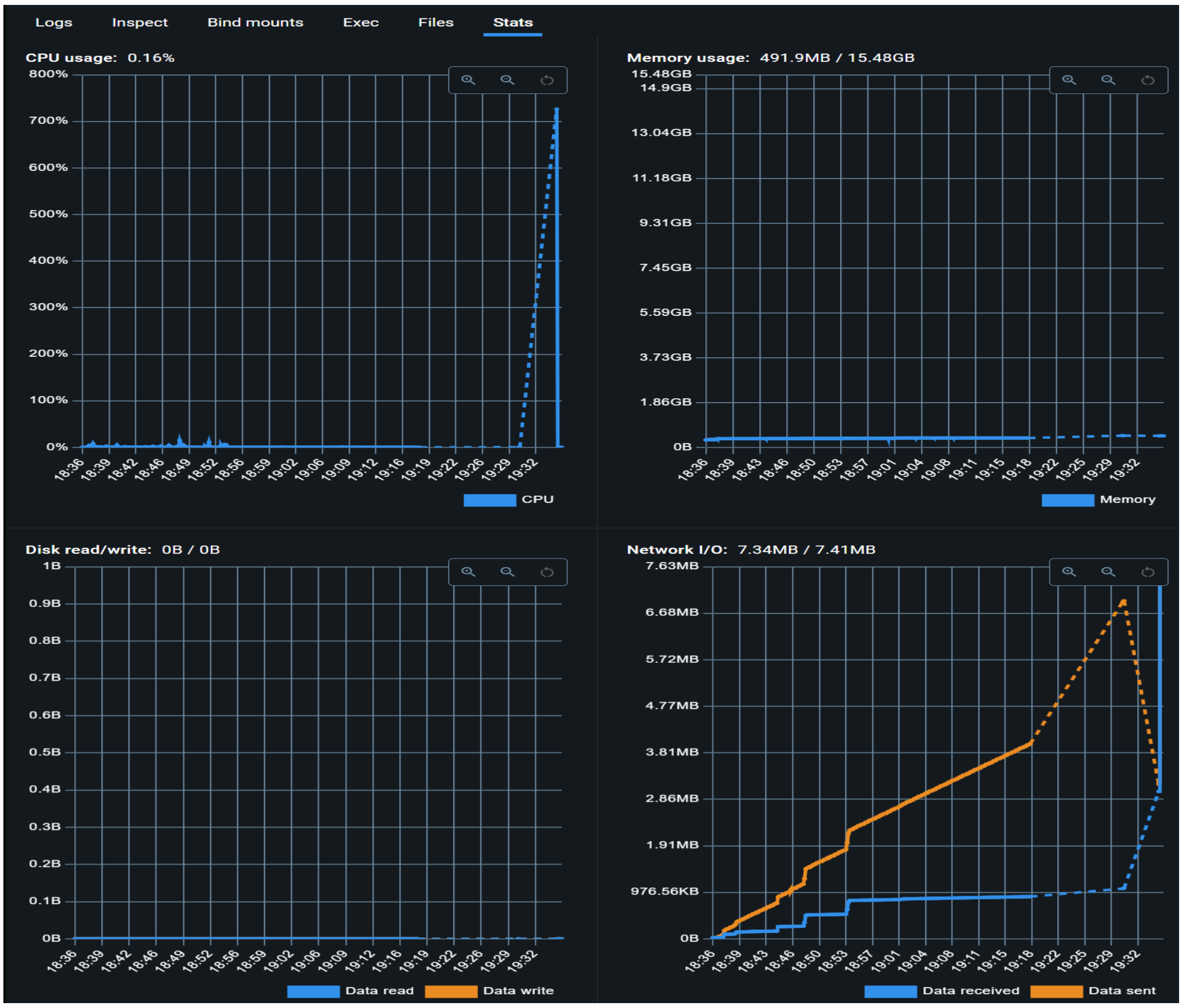
Estimate on how many transactions can your account application can handle per second on your development machine

Machine:
Processor 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz, 2304 Mhz, 8 Core(s), 16 Logical Processor(s)
32 GB operating

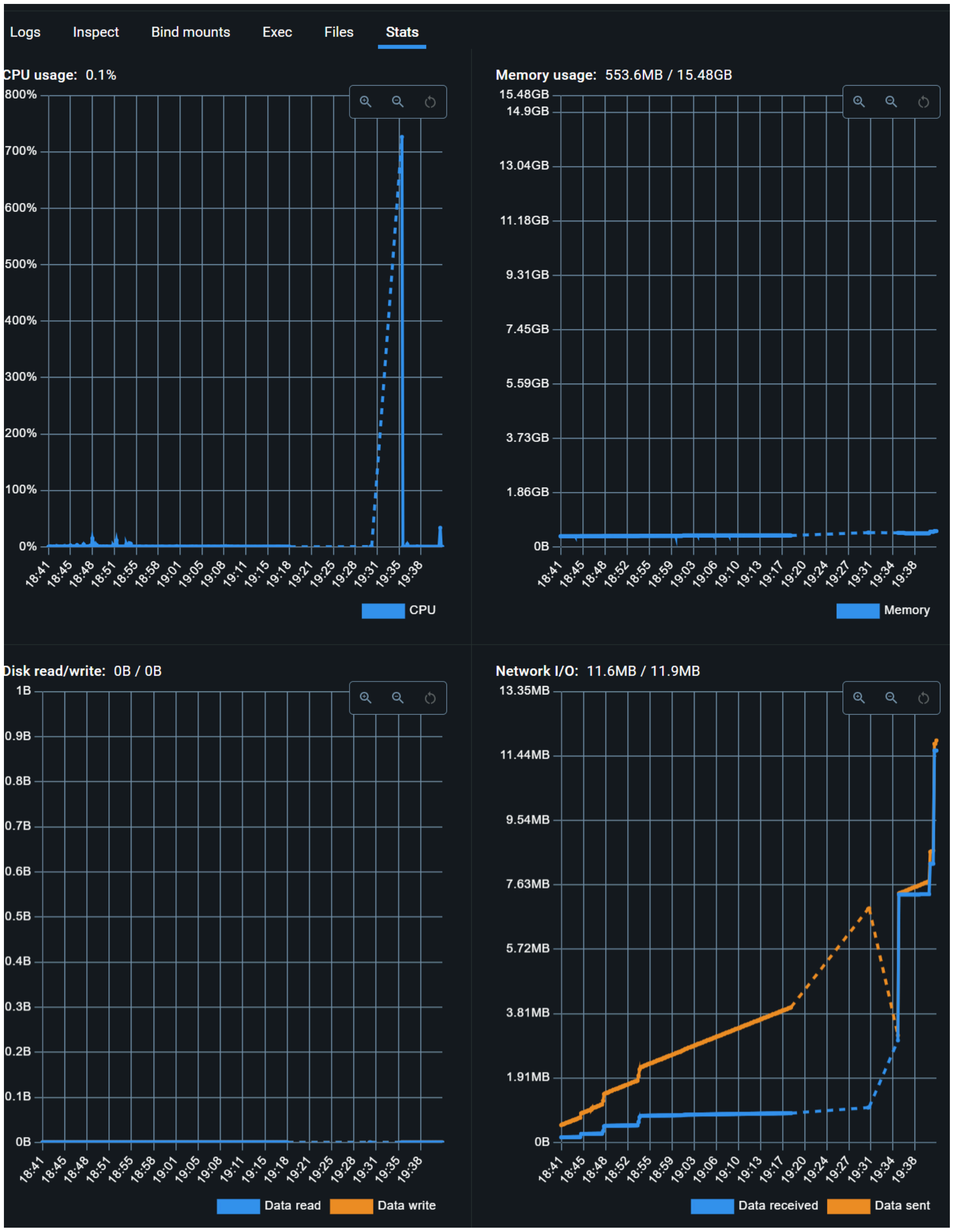
NB!!
Stress tests are highly impacted by the limitation of the Docker container. My local set-up suppresses Intelije Idea, Docker Compose, Kubernetes, and several other applications to balance.
So we assume that the Docker machine had smth like 4GB of operation power and 4 cores allowed for computation.

I used JMeter from Apache to simulate the edge case of the 10000 request per sec and got a CPU spike of 800%

Non-optimistic case: 10000 user requests per sec



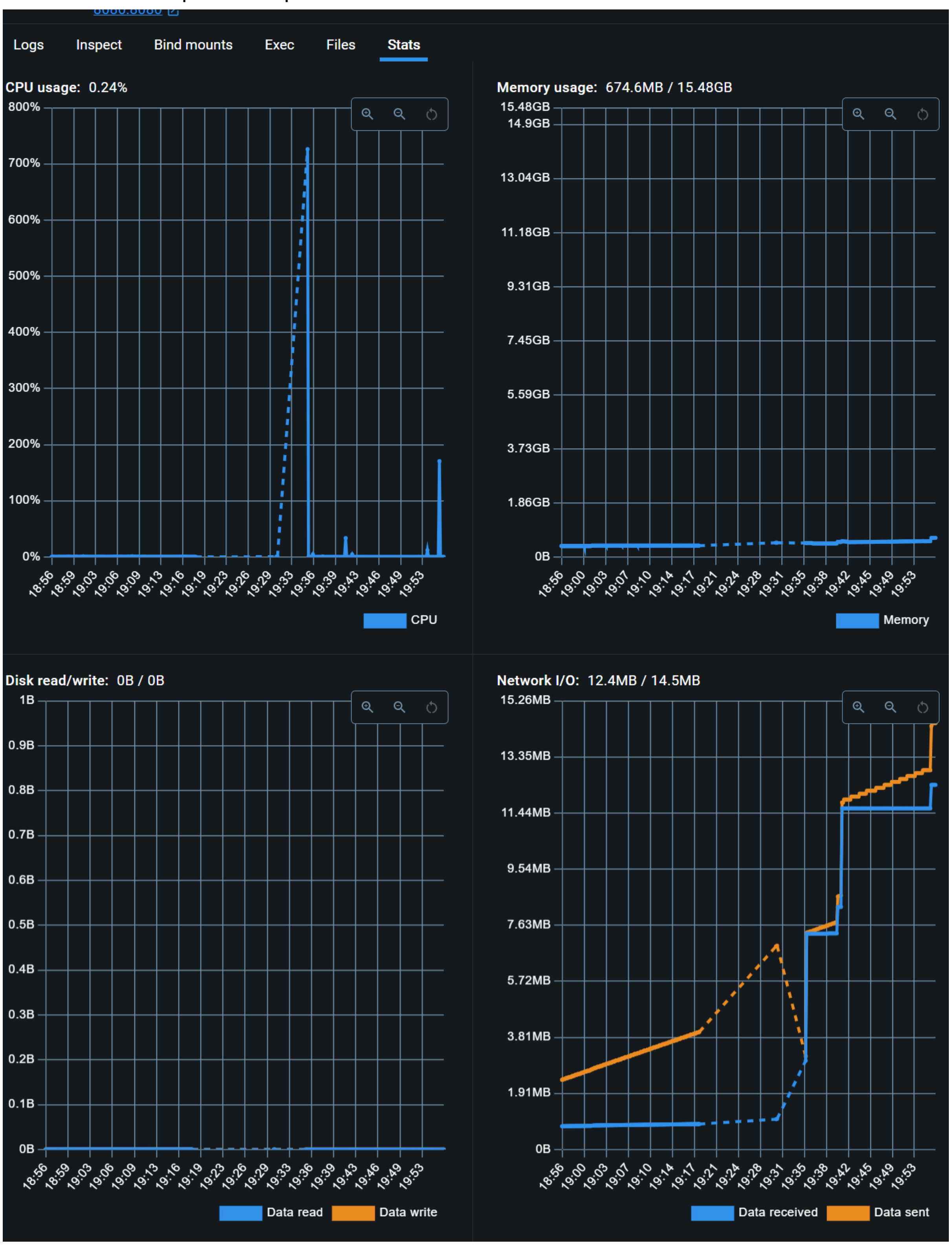
5000 get user per sec consumed only 33% of CPU



Based on that, we can say, reasonably, we can handle up to 7k /api/accounts/x per sec.

Let's get to more costly operations, such as Transactions.

Stress test 1:
500 Transactions per sec equals 180%


Logs Inspect Bind mounts Exec Files **Stats**

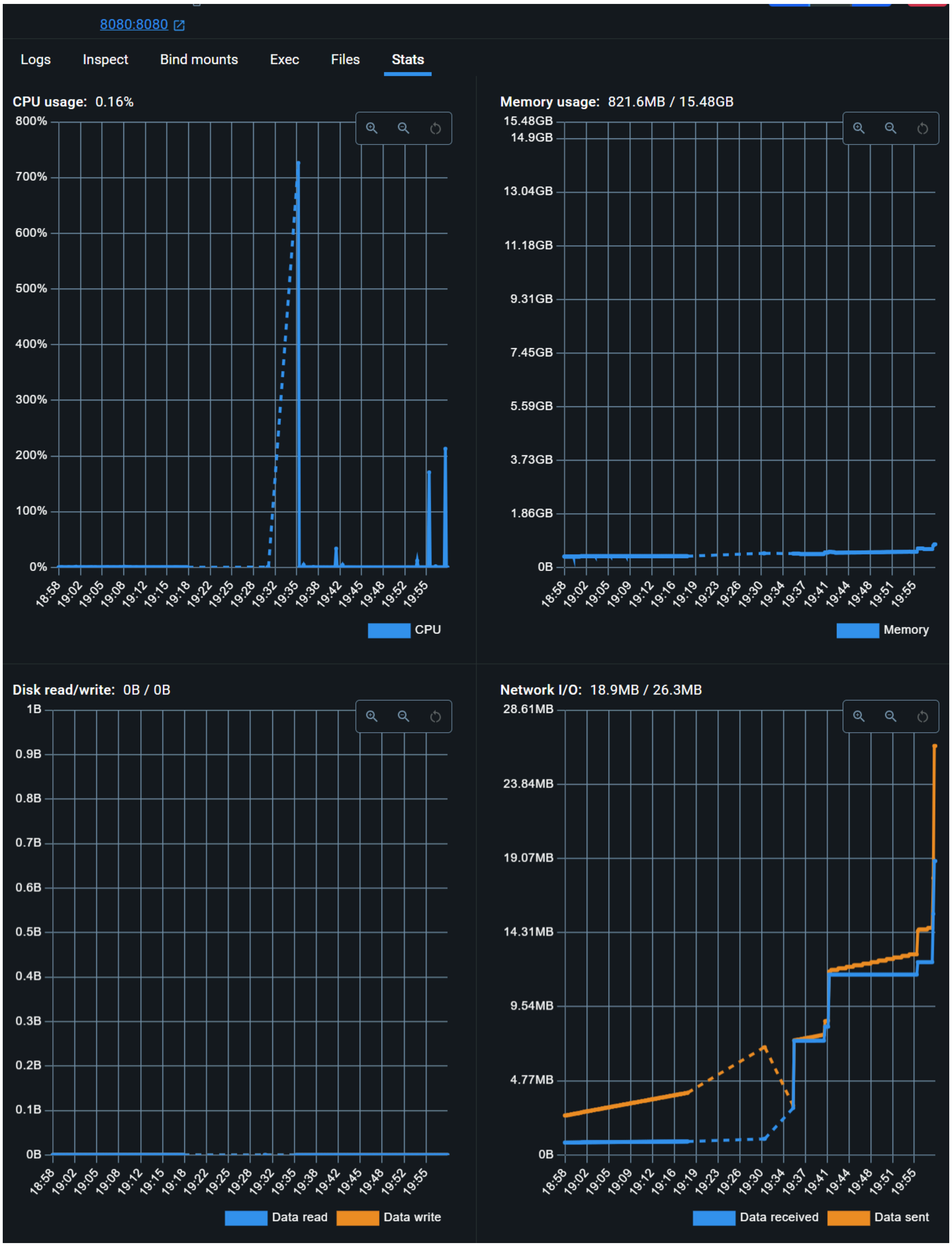
CPU usage: 0.24%

Memory usage: 674.6MB / 15.48GB

Disk read/write: 0B / 0B

Network I/O: 12.4MB / 14.5MB

Stress test2:
With 5000 per sec equals to 200%



8080:8080

Logs Inspect Bind mounts Exec Files Stats

CPU usage: 0.16%

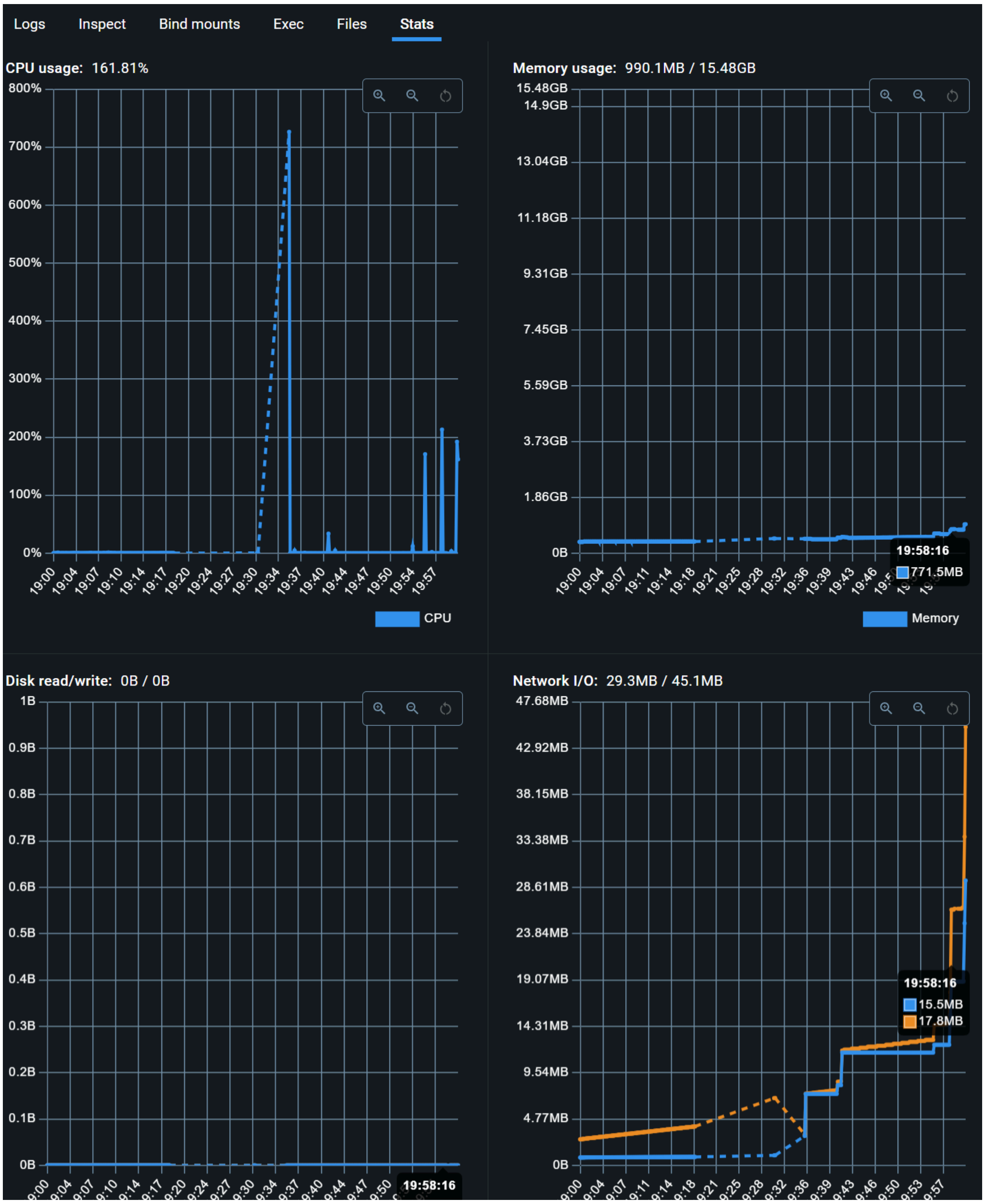
Memory usage: 821.6MB / 15.48GB

Disk read/write: 0B / 0B

Network I/O: 18.9MB / 26.3MB

Stress3

Increased to 10k transactions per sec, spice fixed on the 200% CPU load



Summary:
From the data we have, we can make some observations:

Stress Test 1: At 500 Transactions per second, we see 180% CPU utilization.
This is already above the 100%, indicating the system is overutilized. We're likely pushing past the limits of the available CPU resources.

Stress Tests 2 and 3: Increasing the load to 5000 and then to 10,000 TPS doesn't significantly change the reported CPU utilization (200%), which could mean we have hit a bottleneck.

This bottleneck could be the CPU itself, I/O limits, network limits, or a limitation within the application.

Given this, and assuming that the Docker container is constrained, it's reasonable to suggest that the actual transaction handling capacity of the application on the development machine is slightly below the point where you first hit the 180% CPU utilization.

—
We assume adequate utilization is around 70-85% and want to avoid spikes.

We consider that we have 4 GB RAM and 4 cores used for docker tests

Sustainable transaction per sec = 4(core) / (1.8) * 500 = ≈ 1111 TPS

In my current machine setup, there would be 1100 Transactions per second.

Where 500 is the result of a test where the CPU was 180%