Architectural design for research experiments

F.R.P. Stolk* D. Debreczeni† P. Andrikopoulos‡

October 24, 2024

1 Introduction

In today's data-driven world, the management and monitoring of critical environmental variables play a pivotal role in scientific research and experimentation. In this project, we simulate a leading research institution with a diverse team of 10,000 scientists specializing in chemistry, biology, and physics. To address the need for accurate temperature observability, they have commissioned our team to develop a Temperature Observability microservice. This microservice will have two main tasks:

- Store temperatures that were measured during the experiment
- Notify a researcher when a temperature is outside a pre-defined range

2 Difficulties During Development

In this section the main problems we found are discussed. We start about the database architecture, following up with variable consistency and we finish this section with the gRPC communication for the timestamps.

2.1 Database architecture

In the beginning, we had designed a relational database to store temperature measurements and out-of-bounds measurements (see Figure 1). However, after a group discussion, we decided this architecture would face two problems.
First, due to this design following a relational database design, it would be slow as every query would need to do multiple joins on the data. Second, relational databases are not optimized to run in-memory, which further slows down operations. To tackle these problems, we decided to use MongoDB, a No-SQL database that can store data in-memory.

*f.r.p.stolk@student.uu.nl
†d.debreczeni@student.uu.nl
‡p@student.uu.nl

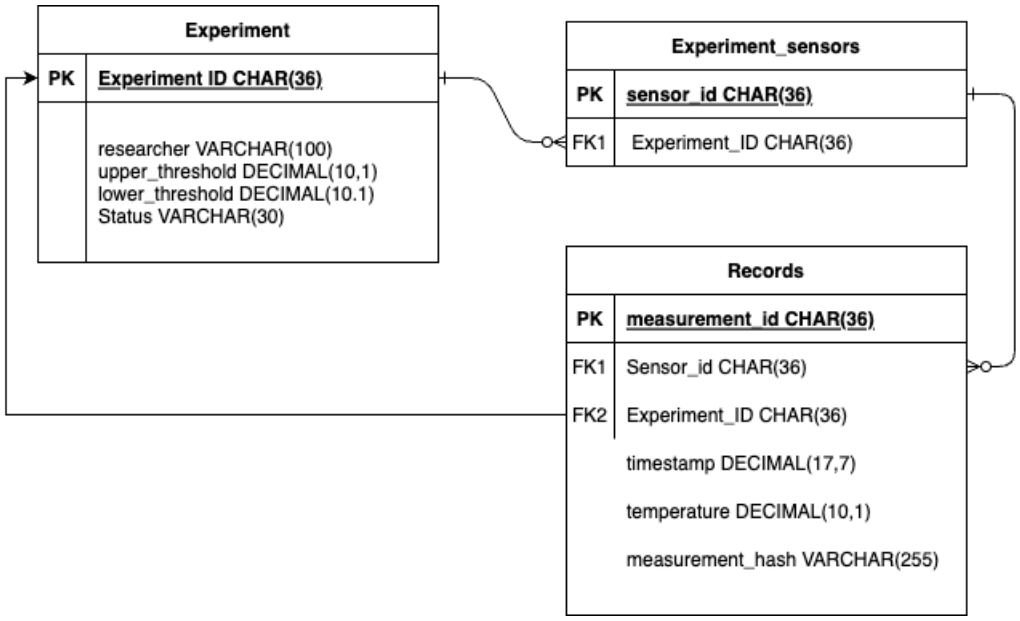

Figure 1: Initial ER diagram of database

Unfortunately, in the last days before the demo, we concluded that the in-memory storage function of MongoDB was only available for enterprise accounts. As a workaround, we mounted a tmpfs storage to the container, enabling MongoDB to emulate its standard behavior for writing to IO while utilizing RAM as the storage medium. While this approach is not as optimal as employing an in-memory device, it does offer improved performance in comparison to traditional disk-based operations.

2.2 gRPC-communication

We decided to use gRPC as a communication medium between our services, as it allows the system to very quickly communicate via http/2 using protocol buffers. When everything was set up, all seemed to be working. However, when querying the API-services we saw that all timestamps were converted into the same timestamp. We eventually realized that the timestamps were lost in the gRPC communication, as float variables were too small to hold the epoch timestamp values. After changing the timestamps to double, everything was fine and we completed the demo without any errors.

3 Architecture

In this section, we will dive into the architecture of our microservices. Our architectural design is structured so that each component serves a distinct and well-defined purpose with clear separation of concern. We will begin by exploring the consumer, followed by an examination of the notifier, the database gateway. Finally, we will conclude with an analysis of the API service.

3.1 Consumer

The consumer is designed to run in an infinite loop to consume every published message from a Kafka topic. A separate thread is created to handle the asynchronous communication with the notifier and the database_gateway. The main task of the consumer is to parse the messages and keep track of the experiments. It can keep the experiment configuration in-memory, as experiments are produced with the experiment ID as key, so even if the component is scaled out, a single experiment will not be distributed among multiple consumers.

When the average temperature collected from all sensors is stabilized for the first time the consumer starts a notification procedure. To achieve this, the consumer sends all the necessary information (notification_type, researcher, measurement_id, experiment_id, cipher_data) to the notifier component. The consumer does not have to know anything about how the notification takes place nor does it need to handle any errors, these tasks are offloaded to the notifier component. After the stabilization, the recording of the measurements starts. The consumer has to communicate with the database-gateway in order to send the information that has to be stored in the database (experiment-id, out-of-bounds = a Boolean value that declares whether the measured temperature is out of bounds or not, timestamp, temperature=the mean temperature). Again, the data storage specifics is hidden from the consumer component, it is fully managed by the gateway. The consumer again communicates with the notifier when the temperature is out of range for the first time and then again when it is stable again.

3.2 Database Gateway

The database_gateway component includes all the necessary methods for retrieving and storing data in the database. It includes the query_out_of_range and the query_temperature that will be called from the rest-api service when a user makes a rest request to the rest-api interface, and the add_temperature, which is used to store new temperature records in the database. The database gateway component connects to MongoDB and provides a simple, async interface to it. Additionally, it also sets up indexing on the database. We set up two indexes, firstly on (experiment ID, timestamp), second on (experiment ID, out-of-bounds). These are optimized for the two fetch queries the rest-api is making.

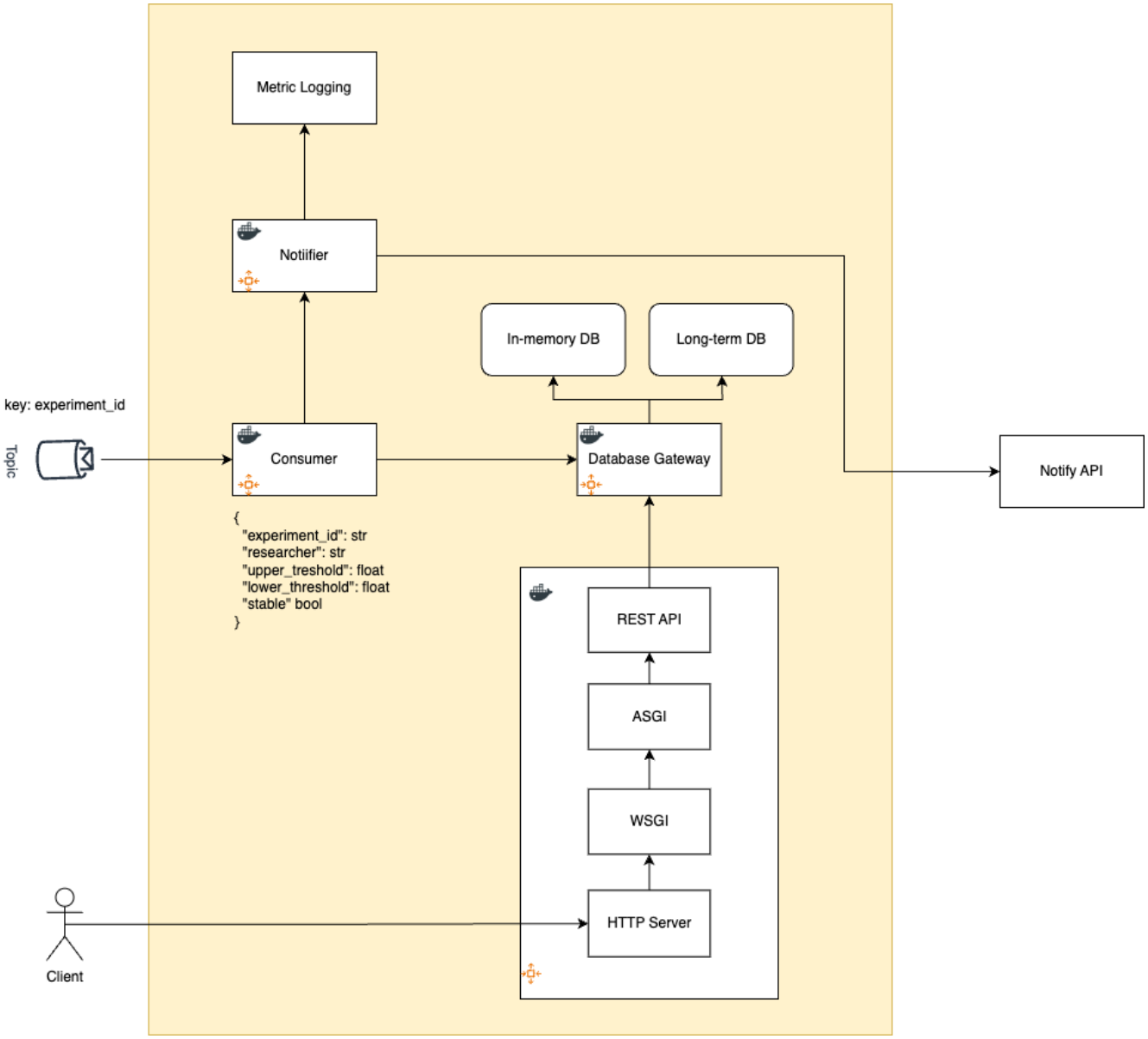

Figure 2: Architecture

3.3 Notifier

The notifier service's task is to parse and forward the data from the consumer to an external notification service, and provide graceful error handling. When receiving a 5xx HTTP error from the external service our notifier service will retry the query with exponential backoff until a maximum retry time is reached. Communication between our notifier component and the external norification_service is achieved through a POST request to the external service's REST API interface. The notifier is also made async.

3.4 REST-API

The REST API service uses FastAPI, which provides a high performance interface to our system. The service runs multiple Asynchronous Server Gateway Interface (ASGI) uvicorn workers, managed by a Web Server Gateway Interface (WSGI) called gunicorn. We found the optimal worker numbers to be four, although as the rest-api is asynchronous and is not doing anything CPU-heavy, there is minimal difference between multiple workers. Still, it does decrease mean latency in cases where the workload is very high. The purpose of the rest-api service is to allow users to access the measurement of specific experiments. More specifically, rest-api allows the user to request all the recorded temperatures of an experiment or those that were out of bounds. Finally, we use nginx proxy to make the rest-api service accessible from the browser and from any device through our group VM IP address and port 3003.

4 Results

4.1 Benchmark

In Figure 3 you can see the benchmark CPU and Memory usage, the spike is setting up the clusters from Kubernetes but overall, it averages out at 10% CPU usage. In Figure 3, the benchmark for the memory is also shown. In this figure, two different levels are shown, the left side is without virtualization (driver=none), and the right side is with a default driver for virtualization.

4.2 Data consistency test

For the data consistency test, our notifier ran very well. As the reader can see in Figure 4, the notifier classified 184 out of 184 notifications and all of them were in the fastest category.
Our API service went also very well. It could handle every request that was sent and the response time was less than 1 second every time as is shown in Figure 4.
Our CPU usage and memory usage were not increasing very much as is shown in Figure 5. The CPU usage is still around 10% and the memory has increased

by around 20 MegaBytes. This is during the in-memory storage of the consumer and the database.

4.3 Stress test

For the stress test our notifier ran well as well. The results did not change in comparison with the data consistency test as is shown in Figure 6.
As for our API, the response time worsened a bit, the median latency rose from 0.25s to 0.5s but the maximum latency never surpassed 2.5s, as is also shown in Figure 6.
Figure 7 shows that the CPU and memory usage increases the most. Our CPU usage went up to 40% and our memory went to 2.15 GigaByte. However, this is expected if a stress test is run. The CPU needs to do more work and by running more experiments more is stored in-memory.

5 Why did our architecture succeed

Our architecture succeeded because we made every service asynchronous, which allowed our components to not block during network and IO calls which would be the main bottleneck in this application environment. We optimized our components heavily, such as storing the experiment data in-memory, indexing our database based on the queries run on it, using gRPC for communication, using python version 3.11 for its speedup benefits, utilizing the orjson library in our rest-api for faster JSON parsing, etc.

Additionally to latency requirements, our architecture also holds up well to software architecture standards and clean coding best practices. With the clean separation of concern, our components are easily maintainable and changes to the components can be done locally without interference with other components.

Further improvements are possible by using an in-memory database, either by upgrading to an enterprise mongo account, or using a different database solution. Additionally, the consumer service can be refactored to use an asynchronous library to consume the kafka messages, which would allow us to get rid of the manual thread creation there.

6 Self-reflection

6.1 Friso

I have learned a lot during this course. I started with no experience with microservices and during this course, I have struggled, but also learned a lot. I understand a lot more about virtualization and Cloud computing. The microservice assignment taught me to think about how you can program with a low-latency perspective which was really helpful.

6.2 Domonkos

This project allowed me to get practical experience in using gRPC, which I have wanted to do for a long time. It allowed me to further improve my knowledge on async programming and to get more experience with using kubernetes. I love working on end-to-end software projects, where we start off from designing the system, moving onto developing the components, and after a test environment is set-up we finish with optimizing it. Seeing everything turn green and the evaluation metrics improving is incredibly rewarding.

6.3 Panagiotis

Before this assignment, my knowledge of microservices architecture was very limited. Through this project, I learned a lot about the implementation of an application that is distributed in different components and how all these components have to communicate effectively with each other in an optimal way in order to minimize the latency and give the desired result in the most efficient way. Thanks to this assignment and my colleagues, I am now reconciled with concepts such as microservices, scalability, containerization, distributed programming, and producer-consumer patterns.

Figure 3: Baseline CPU and Memory

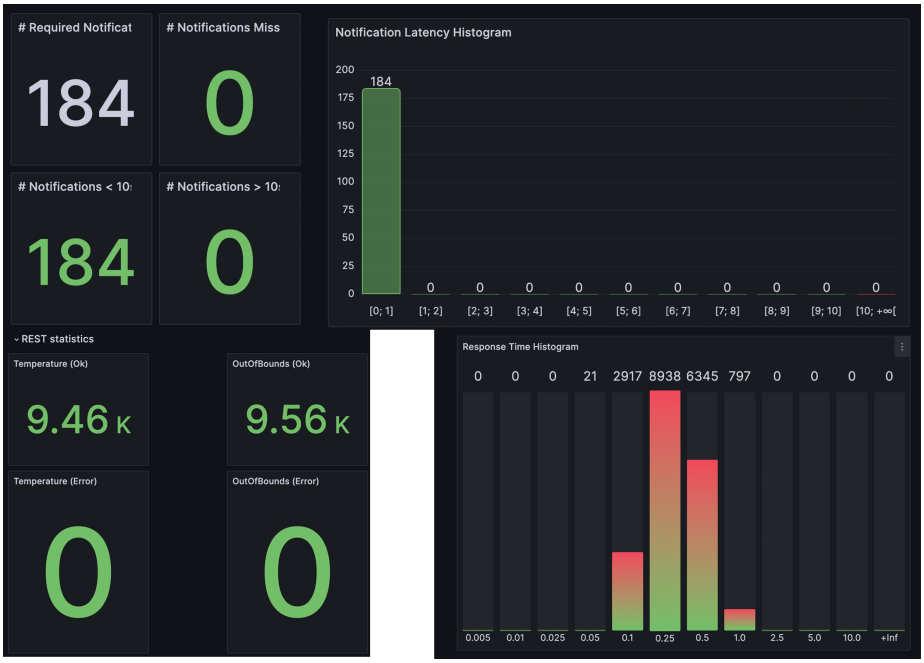

Figure 4: Valid Notifications

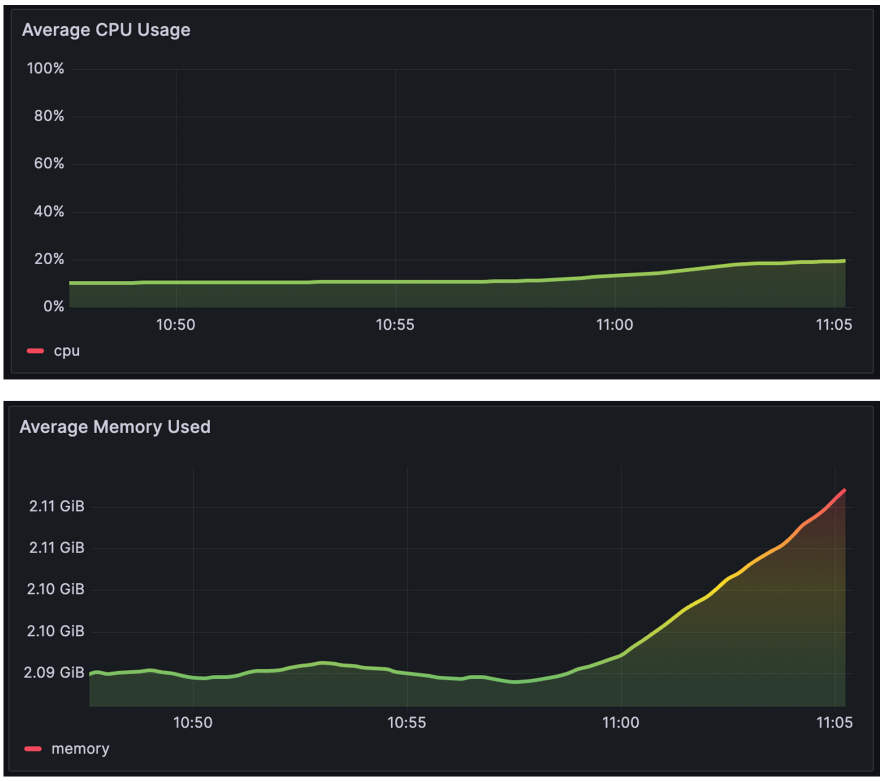

Figure 5: CPU & Memory dataconsistency

Figure 6: Performance stress-test

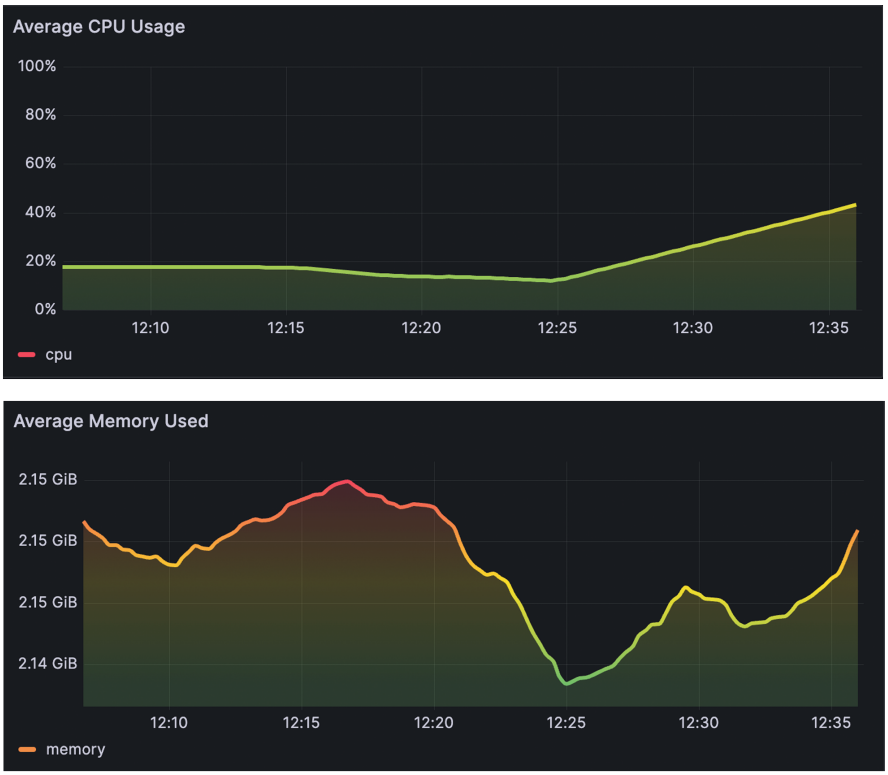

Figure 7: CPU & Memory stress-test