# Cloud computing

Springer Christian

2024-03-19

## The exercise

You are tasked with identifying, deploying, and implementing a cloud-based file storage system. The system should allow users to upload, download, and delete files. Each user should have a private storage space. The system should be scalable, secure, and cost-efficient. Suggested solutions to use for the exam are Nextcloud and MinIO.

### Requirements

The deployed platform should be able to:

1. Manage User Authentication and Authorization:

- Users should be able to sign up, log in, and log out.
- Users should have different roles (e.g., regular user and admin).
- Regular users should have their private storage space.
- Admins should have the ability to manage users.

2. Manage File Operations:

- Users should be able to upload files to their private storage.
- Users should be able to download files from their private storage.
- Users should be able to delete files from their private storage.

3. Address Scalability:

- Design the system to handle a growing number of users and files.
- Discuss theoretically how you would handle increased load and traffic.

4. Address Security:

- Implement secure file storage and transmission.
- Discuss how you would secure user authentication.
- Discuss measures to prevent unauthorized access.

5. Discuss Cost-Efficiency:

- Consider the cost implications of your design.

- Discuss how you would optimize the system for cost efficiency.

6. Deployment:

- Provide a deployment plan for your system in a containerized environment on your laptop based on docker and docker-compose.
- Discuss how you would monitor and manage the deployed system.
- Choose a cloud provider that could be used to deploy the system in production and justify your choice.

7. Test your infrastructure:

- Consider the performance of your system in terms of load and IO operations

## Proposed solution

### Preliminary remarks

The system proposed is cloud-based storage system implemented through containerization with Docker and Docker Compose, utilizing an image of Nextcloud.

#### What is a cloud-based storage system?

A cloud-based file storage system is a type of storage solution that allows users to store and manage their files and data in remote servers over the internet, commonly referred to as the "cloud".

These systems offer various features and functionalities, including file uploading, downloading, sharing, synchronization, and access control. Known third-party cloud-based services are for example Google Drive, Dropbox and Microsoft OneDrive.

Instead of using them, hosting your own file storage system offers several perks, some of them being increased control and privacy over your data, scalability of the structure, cost savings and many more.

#### What is Docker?

Docker is a platform that enables developers to develop, ship, and run applications using containerization technology. Containerization is a lightweight alternative to full machine virtualization, where applications are packaged along with their dependencies and runtime environment into a container image. These container images can then be run consistently across different environments, whether it's a developer's laptop, a test server, or a production environment.

#### What about Docker Compose?

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file (usually named docker-compose.yml) to define the services, networks, and volumes required by an application. With Docker Compose, you can define multiple containers that make up an application, specify their dependencies and configurations, and then deploy them together as a single unit.

#### And finally, what is Nextcloud?

Nextcloud is an open-source, self-hosted file synchronization and sharing platform. It provides similar functionalities to popular cloud-based file storage and collaboration services, but it allows users to retain full control over their data by hosting it on their own servers or using a third-party hosting provider.

## Implementation

### The Docker Compose file

This is the content of the "docker-compose.yaml" file, which is used to define a Docker Compose configuration for setting up a Nextcloud instance with a MariaDB database backend. This file was taken as a blueprint and customized accordingly to fit the exercise's needs.

```yaml
version: '3'

networks:
  nextcloud:

volumes:
  nextcloud:
  db:

services:
  db:
    image: mariadb:10.6
    restart: always
    command: --transaction-isolation=READ-COMMITTED --log-bin=binlog --binlog-format=ROW
    volumes:
      - db:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=Mj8VldcvKqBsI65
      - MYSQL_PASSWORD=Mj8VldcvKqBsI65
      - MYSQL_DATABASE=nextcloud
      - MYSQL_USER=nextcloud
    networks:
      - nextcloud

  app:
    image: nextcloud:28.0
    restart: always
    ports:
      - "8081:80"
    links:
      - db
    volumes:
      - nextcloud:/var/www/html
    environment:
      - MYSQL_PASSWORD=Mj8VldcvKqBsI65
      - MYSQL_DATABASE=nextcloud
      - MYSQL_USER=nextcloud
      - MYSQL_HOST=db
      - NEXTCLOUD_ADMIN_USER=christian
      - NEXTCLOUD_ADMIN_PASSWORD=Mj8VldcvKqBsI65
    networks:
      - nextcloud
```

In the Docker Compose file, two services are defined: `db` (for MariaDB) and `app` (for Nextcloud).

MariaDB is an open-source relational database management system, here it's used as the database backend for the Nextcloud instance.

The keyword `db` specifies the MariaDB service configuration. It uses the “mariadb:10.6” Docker image, sets up volume mapping for persisting MariaDB data, and defines environment variables for configuring the MariaDB instance, including root password, database name, user, and password.

On the other hand, the keyword `app` defines the Nextcloud application service. It uses the nextcloud:28.0 Docker image, exposes the selected port `8081` on the host for accessing the Nextcloud application, sets up volume mapping for persisting Nextcloud data, and defines environment variables for configuring Nextcloud, including database connection details, admin username, and password.

Under the `app` definition, there is a `links` keyword that creates a connection between the Nextcloud and the MariaDB containers.

Environment variables are specified for both services to configure their respective components. In the app service, environment variables like MYSQL_PASSWORD, MYSQL_DATABASE, MYSQL_USER, and MYSQL_HOST are set. These variables specify the password, database name, username, and host of the MariaDB database, respectively. By setting `MYSQL_HOST=db`, Nextcloud knows to connect to the MariaDB service using the service name `db`.

To run this file one must install Docker, then open a terminal, navigate to the directory where this file is stored and run the following bash command:

```bash
sudo docker-compose up -d
```

This command will start the deployment and the hosting of the containerized instance of Nextcloud. It’s worth mentioning that the command “-d” stands for “detached” mode. It starts the file in the background, and then returns you to your command prompt without blocking the terminal. This means that you can continue to use your terminal for other tasks while the Docker containers continue to run in the background.

Without the -d flag, docker-compose up runs the services in the foreground, and you see the output logs from the running containers in your terminal, which is be useful for debugging or monitoring purposes. However, this also blocks your terminal until you manually stop the services with Ctrl+C.

In the second case, when you close the terminal the cloud closes abruptly, while in the first case it keeps running in the background and to put it down one must use the following command:

```bash
sudo docker-compose down
```

Now that the cloud is being hosted you can open a web browser and navigate to the respective port, that being

- “http://localhost:8081/”

which will take you to the Nextcloud instance hosted in the container. To log in the admin must use the credentials defined in the file above, in this case:

- Username: christian
- Password: Mj8VldcvKqBsI65

The password was obtained through a random password generator set to 15 characters excluding symbols, ensuring a very strong password and hopefully protection.

Once logged in, the admin have a lot of possible tasks, including:

- upload files
- download files

- remove files
- customize the interface
- create new users (both admins or regular users)
- change it's password and username
- manage users (e.g. deleting one)

Regular users instead can receive the credentials to an account from the admin, then log in and customize their username and password to have a personal space on the cloud. They can manage the files inside their storage, but not manage the cloud itself, as it should be.

To ensure more protection for the user's accounts, the admin can log in, click on it's icon on the top right > Administration settings > Privacy and from there enforce password rules like forbid common passwords and set a minimum length. Additionally it's possible to enforce a Two-Factor Authentication, making the accounts even more secure.

### Scalability

To tackle the scalability issue, an idea would be to horizontally scale the application and database containers. This can be achieved by deploying multiple instances of the application container (Nextcloud) and using a load balancer to distribute incoming traffic among these instances. Similarly, deploying multiple instances of the database container (MariaDB) and configure them in a master-slave replication setup to distribute database reads and writes. This can be done by adding the `deploy` key with the `replicas` option and selecting the desired number of replicas. For example:

```
# ...
app:
  image: nextcloud:28.0
# ...
  deploy:
    replicas: 3
# ...
```

By default, Docker Swarm will automatically load balance incoming traffic among the replicas of the service.

### Cost-Efficiency

One possiblity could be ensuring that the resources allocated to the Nextcloud and database containers in the Docker Compose file are appropriate for the user's needs. It's important to avoid over-provisioning resources, which can lead to unnecessary costs. Monitor container resource usage and adjust as necessary. Consequently, it would be wise to consider scaling the Docker containers vertically (by increasing container resources) or horizontally (by adding more container instances). Horizontal scaling allows for better distribution of workload and can improve performance and resilience. It's important to evaluate if hosting a local cloud-based storage system is cheaper than subscribing to one from a third-party, since it eliminates the need of buying additional hardware. If the user wants at all costs to keep the data and files to himself than opting for a local system would be the way to go, since using third-party systems involves giving them the files.

## Performance testing

To test the performance of the system I used Locust, an open-source load testing tool used for assessing the performance and scalability of web applications. It allows to simulate thousands of concurrent users interacting with the wanted application, generating a load that closely resembles real-world usage patterns.

All the computations were performed on a laptop with the following specifics:

```
## [1] "Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz"
```

The file used to create users for testing is the following:

```
#!/bin/bash

URL="http://localhost:8081/ocs/v1.php/cloud/users"
USERNAME="christian"
PASSWORD="Mj8VldcvKqBsI65"
PASSWORD_ENCODED="abc123abc!"


for i in {1..20}; do
    USERID="user$i"
    docker exec -i -u 33 cloud_app_1 bash -c "export OC_PASS=$PASSWORD && /var/www/html/occ user:add $U
done
```

This creates 20 users named user1, user2, ecc. all with the same password "Mj8VldcvKqBsI65".

The Locust file ('locustfile.py') used to define tasks performed by the spawned users is the following:

```
from locust import HttpUser, task
from requests.auth import HTTPBasicAuth
import requests
import random

with open("output.txt", "a") as f:
    f.write(f"__________________________________________________\n")

class NextcloudUser(HttpUser):
    auth = None
    users_list = list(range(1, 20))

    def on_start(self):
        random.shuffle(self.users_list)
        i = self.users_list.pop()
        self.user = 'user' + '{:d}'.format(i)
        self.password = 'Mj8VldcvKqBsI65'
        self.auth = HTTPBasicAuth(self.user, self.password)
        self.verify_authentication()

    def verify_authentication(self):
        response = self.client.head("/remote.php/dav", auth=self.auth)
        if response.status_code != 200:
            with open("output.txt", "a") as f:
                f.write(f"Authentication failed for user {self.user}: {response.text}.\n")
            raise Exception(f"Authentication failed for user {self.user}")

    @task
    def propfind(self):
        try:
            response = self.client.request("PROPFIND", "/remote.php/dav", auth=self.auth)
            response.raise_for_status()
        except Exception as e:
            with open("output.txt", "a") as f:
                f.write(f"Error during PROPFIND request: {e} for user {self.user}.\n")

    @task
    def upload_small(self):
        filename = "files_test/picture.png"
        with open( filename, 'rb') as f:
            response = self.client.put("/remote.php/dav/files/" + self.user +                         "

        if response.status_code != 201 and response.status_code != 204 :
            with open("output.txt", "a") as f:
                f.write(f"Error during PUT request: {response.status_code}
            return

        for i in range(0,5):
            self.client.get("/remote.php/dav/files/" + self.user + "/" +

        self.client.delete("/remote.php/dav/files/" + self.user + "/" +

    @task
    def upload_medium(self):
        filename = "files_test/dataset.dat"
        with open(filename, 'rb') as f:
            response = self.client.put("/remote.php/dav/files/" + self.user +

        if response.status_code != 201 and response.status_code != 204:
            with open("/mnt/locust/output.txt", "a") as f:
                f.write(f"Error during PUT request: {response.status_code}
            return

        for i in range(0, 5):
            self.client.get("/remote.php/dav/files/" + self.user + "/" +

        self.client.delete("/remote.php/dav/files/" + self.user + "/" +

    @task
    def upload_large(self):
        filename = "files_test/lotr.mp4"
        with open(filename, 'rb') as f:
            response = self.client.put("/remote.php/dav/files/" + self.user +

        if response.status_code not in (201, 204):
            with open("/mnt/locust/output.txt", "a") as f:
                f.write(f"Error during PUT request: {response.status_code}
            return


        self.client.get("/remote.php/dav/files/" + self.user + "/" + filename,
                        auth=self.auth,

        self.client.delete("/remote.php/dav/files/" + self.user + "/" +
```

The tasks considered were:

- propfind
- uploading and deleting of a small file (picture.png - 4.2 kB)
- uploading and deleting of a medium file (dataset.dat - 1.9 MB)
- uploading and deleting of a large file (lotr.mp4 - 702 MB)

The files used for testing can be found in the repository and the tests can be replicated (last one is a `zip` file since it was too heavy to put on the repo).

To run the tests you have to open a terminal, navigate to the directory where the 'locustfile.py' is located and run the following bash command:

```
locust -f locustfile.py --host=http://localhost:8081
```

This command attempts to run Locust with the selected test script (locustfile.py) targeting the host located at http://localhost:8081. Then open a web browser and navigate to

- `http://localhost:8089/`

The default port for Locust is 8089 so keep in mind when assigning a port with the `docker-compose.yaml` file to assign a different one from this to avoid conflicts.

The parameters used were:

- Number of users: 10
- Ramp up: 2

Each test was performed individually by commenting out the other tests so capture the actual plot of each task.

In the repository, inside the directory "files_test" you can find the extensive reports for each performed test with all the numbers, here just the report plots will be analyzed for simplicity.

## First Task: Propfind

The first task tested was 'propfind', which sends a "PROPFIND" HTTP request to "/remote.php/dav". PROPFIND is an HTTP method used to retrieve properties of a resource without retrieving the resource itself. If an exception occurs during the request (such as an HTTP error), it logs the error message along with the user's information into a file named "output.txt".

From the graph we can see that the 95th percentile of response time stabilizes to a value of approx. 60 ms.

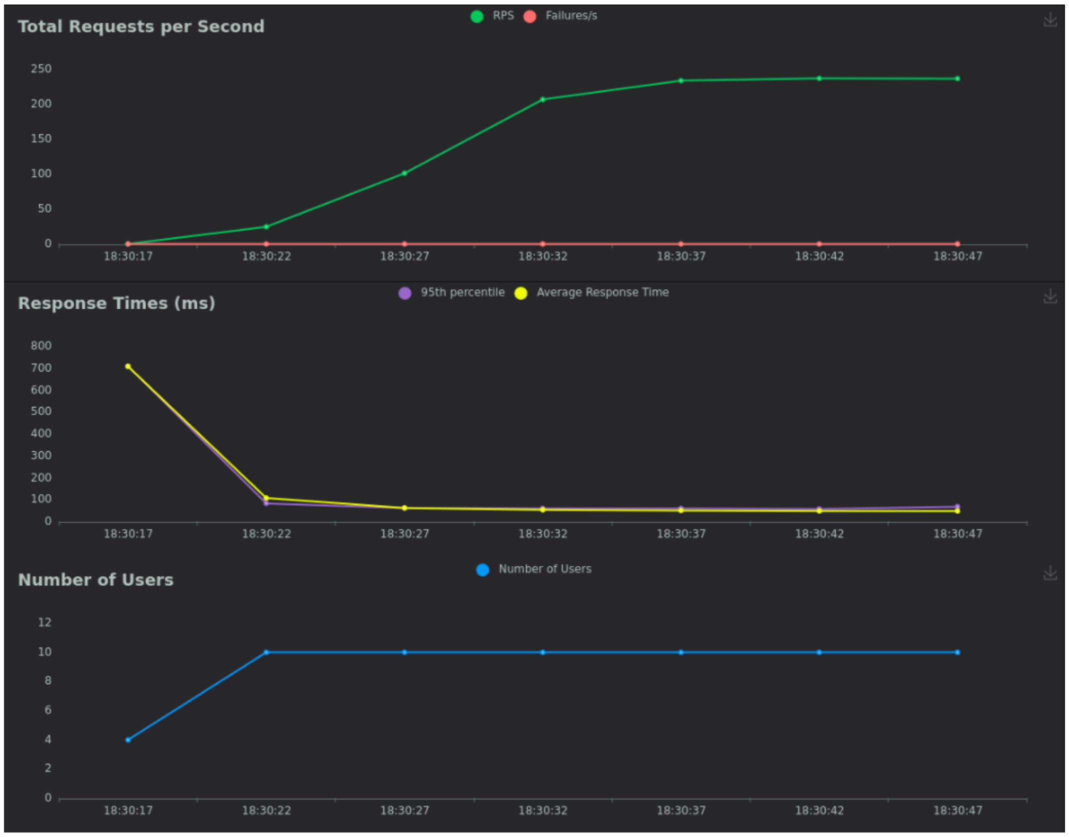

Figure 1: Graph of propfind task

## Second Task: Upload small

The second task uploads a small file (picture.png) to the cloud using the HTTP PUT method, then accesses the file five times with GET requests, and finally deletes the file using an HTTP DELETE request. If any error occurs during the PUT request, it logs the error message along with the user's information into a file named "output.txt".

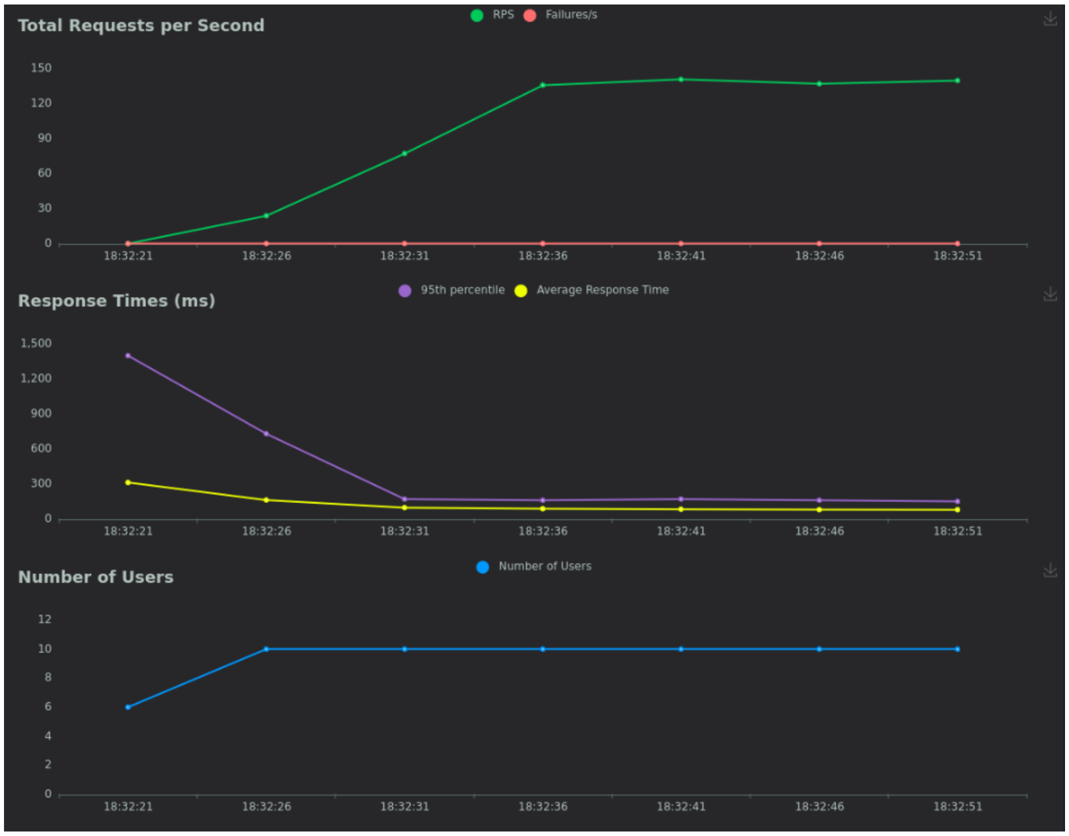

Figure 2: Graph of upload small task

From this graph we can see that the 95th percentile of response time stabilizes to a value of approx. 160 ms, while the average response time is 80 ms.

## Third Task: Upload medium

The third task does the same thing as the second one, with the exception that the considered file is different (dataset.dat) to take into consideration the performance with a file of a couple MBs.

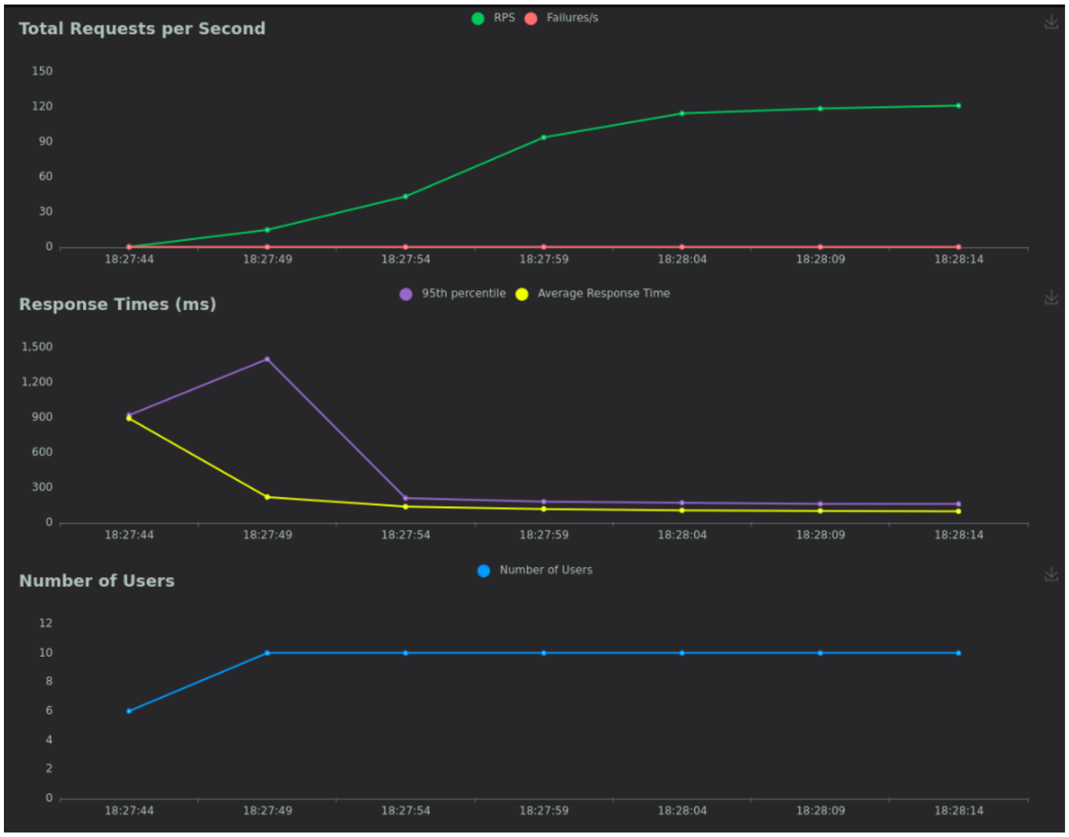

Figure 3: Graph of upload medium task

From the graph we can see that the 95th percentile of response time also stabilizes to a value of approx. 160 ms like in the small case, but now the average response time is higher (100 ms).

## Fourth Task: upload large

The fourth task once again uploads, access and deletes a file (lotr.mp4), but now the file is around 700MB. This task was tested with different parameters since the laptop used to run the tests couldn't handle the rates tested before. This time the test was done setting - users: 5 - rate: 2

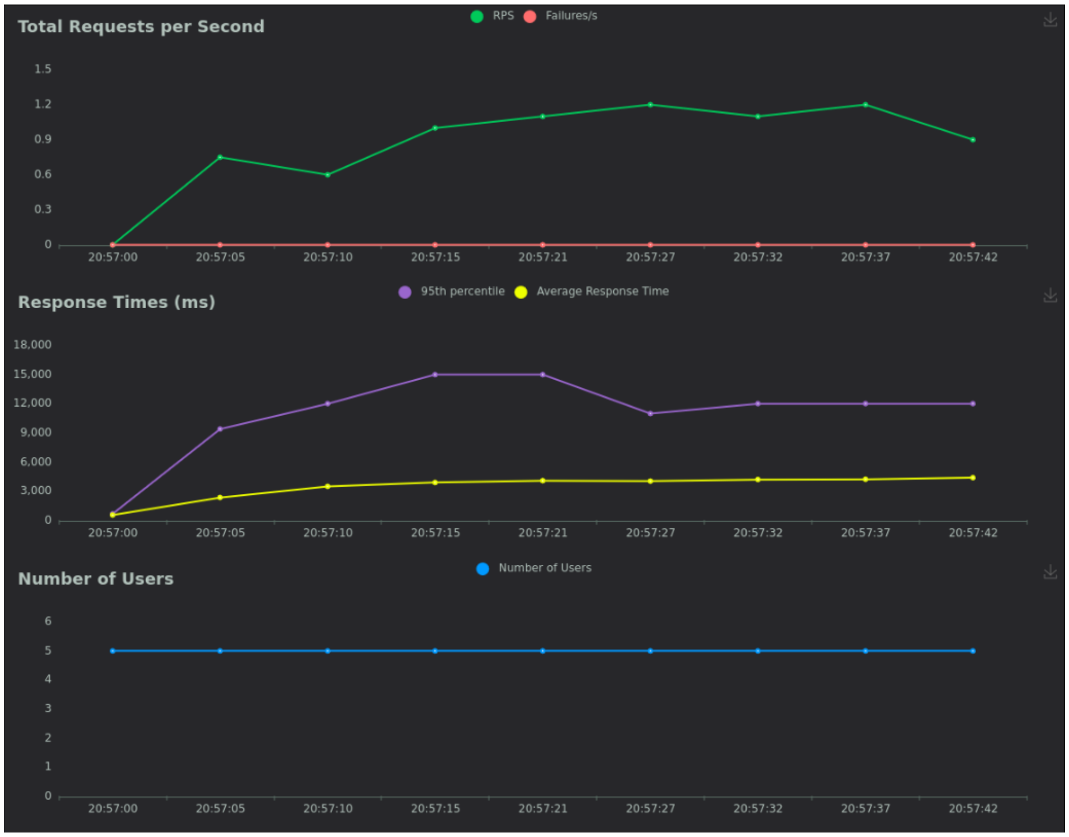

Figure 4: Second graph of upload large task

From the last graph we can see that the 95th percentile of response time stabilizes to a value of approx. 1200 ms and on an average response time of 4400 ms, in line with the previous results and the weight of the file used.

## Conclusions

The proposed solution to the given exercise was implementing a docker-composed instance of Nextcloud. Many tests were conducted to assess the performance of this cloud, like handling small, medium or large files. In cases where the files considered are heavy it would be useful to consider a way to scale the system to handle the high workload, possibly scaling the system horizontally since vertical scaling would cost more and be overall less versatile.