# Cloud Computing Final Project Report - Cloud Based File Storage System

**Marta Lucas SM3800043 - Data Science and Artificial Intelligence, University of Trieste**

11/09/2025

## Introduction

The final project for the cloud course involves the deployment and implementation of a **Cloud-Based file storage system**. The system must allow users to upload, download, and delete files each within their own private storage space. The system should be scalable, secure, and cost-efficient.
This report includes design choice motivations, implementation details, security measures, a scalability and cost-efficiency analysis of the system.

### Nextcloud

For the file storage platform, **Nextcloud** was selected. Nextcloud is an open-source self-hosted cloud storage solution that allows users to store and share files in a private storage space.
It supports file uploads, downloads, and deletions, along with user and group management for administrators.
Nextcloud also offers several built-in security features, including two-factor authentication, server-side encryption, and OAuth2 support.
These features, along with its ease of deployment, made Nextcloud a better fit for this project compared to other alternatives.

### MariaDB

The system uses **MariaDB** as its database backend to store file metadata and configuration settings. It is a relational database management system that is fully open-source and known for better performance and scalability compared to the default SQLite option offered by Nextcloud.

Nextcloud and MariaDB were both deployed using **Docker Compose**, allowing for modular and reproducible setup in containerized environments.

### Docker and Docker Compose

Docker is a platform that allows for the creation, deployment, and management of containerized applications. Docker Compose is a tool that simplifies the process of defining and running multi-container Docker applications using a YAML file.
As previously mentioned, Nextcloud and MariaDB were deployed using Docker Compose, which allowed for easy configuration and management of the two services. The `docker-compose.yml` file defines the services, their dependencies, and the necessary environment variables. Two volumes were also created to persist data for both Nextcloud and MariaDB, making sure that data is not lost when the containers are stopped or removed.

To deploy the system, run `docker-compose up -d` in the directory containing the `docker-compose.yml` file.
This will start both services in the background. Once the containers are up and running, you can access the Nextcloud web interface by navigating to: `http://localhost:8080` .
First access can be done with the admin credentials specified in the `.env` file (see Security section for more details).
For a guided setup of the system and usage instructions, please refer to the README file in the project repository.

## Security

Security is fundamental for any cloud-based system, especially when dealing with sensitive data. In this project, some best practices have been intentionally relaxed, for example, the .env file containing credentials is included in the GitHub repository. In a real-world deployment, this should be excluded using a `.gitignore` file, and credentials should be stored in a secure, encrypted environment.

After registration, Nextcloud users can authenticate with their username and password. Once logged in, Nextcloud issues an access token, which is used by the client for all subsequent HTTP requests. This token should be stored securely on the client side and must not be shared or saved on any other system.
Additionally, user passwords are stored in encrypted form in the Nextcloud database, guaranteing that even in the event of a database breach, raw passwords remain protected.

Nextcloud provides several built-in security features, many of which are configurable through the admin interface or through the command line. These include:

- **Two-factor authentication (2FA)**: Adds another layer of protection by requiring a second form of verification during login.
- **Server-side Encryption**: Encrypts files at rest on the server to prevent unauthorized access, even if the storage backend is compromised;
- **OAuth2**: Allows secure authentication and token-based access using external identity providers (e.g., Google, GitHub);
- **Logging and monitoring**: Tracks user activity and system events to detect and investigate potential security incidents;
- **Password policies**: Enforces secure password creation through a configurable set rules
- **Brute-force protection**: Limits the number of login attempts to prevent brute-force attacks;

In this case only the last two security measures were enabled. This is because the other features, while valuable in production, introduce additional overhead that may hinder performance or usability in a lightweight demonstration setup.

The password policy can be directly fixed by running the `password_security.sh` script. You can verify whether these settings were activated by using the `test_password_security.sh` script, which attempts weak password resets and multiple failed logins to test.

In addition to the default settings of Nextcloud, which check for commonly used passwords and also ensure they are not found in the *haveibeenpwned* database of compromised passwords, this script will add some security measures to create stronger passwords and protect user accounts.
The password policy includes:

- Minimum password length set to 10 characters
- Inclusion of both uppercase and lowercase letters
- Inclusion of at least one number
- Inclusion of at least one special character

Furthermore, accounts will be locked after **5 failed login attempts** and must be unlocked by an admin. Also **password expiration** is set to **30 days**, after which users will be prompted to change their password.

You can modify these settings by editing the script or manually adjusting the values in the Nextcloud admin interface . The same goes for the other types of security measures mentioned above. For production deployments, it is strongly recommended to enable at least 2FA, encryption and OAuth2.

## User Management

Admin users can create and manage user accounts through the Nextcloud web interface. Each user is provided with a private storage space where they can upload, download, and delete files. Admins can also assign users to groups, set storage quotas, and monitor users activity.

In this project, a script named `create_user.sh` automates the creation of multiple user accounts for testing purposes. It generates 100 test users with a space quota of 4GB, which will be used for load testing with Locust.
The usernames are in the format `test_userX`, while display names are in the format `Test User X`, where `X` is the numeric index of the user (from 1 to 100). The password follows the format `Test_passwordX!`, which complies with the password policy set in the previous section.
Deletion of users can be done manually through the admin interface or by using the `delete_user.sh` script, which removes all the prevously generated test users.

It's recommended to run the `delete_files.sh` script before clearing users, as it will delete all files uploaded by the test users, helping to free up storage and maintain a clean environment

## Locust Testing

Locust is an open-source load testing tool that allows you to define user behavior in Python code and simulate concurrent users. It provides a web-based interface for monitoring test progress and analyzing results in real time.

To run tests effectively, all test users must already exist on the Nextcloud platform, this can be done through the aforementioned `create_user.sh` script. Since some scenarios involve file uploads, you should also first run the `create_test_files.sh` script. This generates three test files of different sizes: 1KB, 1MB, and 1GB.

The tasks to be performed during user behavior simulations are defined in the .py files in the `locust` directory. Common elements across all files include:

- Authentication (HEAD): Verifies user credentials and server availability.
- Search (PROPFIND): Lists the contents of the user's root directory.

- Deletion (PUT + DELETE): Uploads a temporary file and then immediately deletes it, checking that the file exists before deletion and avoiding errors.
- Read (GET): Retrieves the contents of the Readme.md file, which is included by default in every new user's storage.
- Upload (PUT): Uploads files of various sizes, depending on the load scenario.

Tasks are assigned weights using the @task(X) decorator, where X determines how frequently a task is performed relative to others. This allows the simulation to reflect realistic usage patterns, where some actions (like file uploads) are more common than others. You can observe task weights in the Python filess.

The `on_start` and `on_stop` methods, also shared across all test scripts, manage user setup and cleanup.

The `on_start` method initializes each virtual user by assigning them a random username and password from the test pool and sets up an empty list `uploaded_files` to track the paths of uploaded files during the test. This ensures each user simulates realistic, independent behavior.
The `on_stop` method is triggered when the user session ends. It iterates over all uploaded file paths and deletes them, preventing test data from accumulating on the server.

Each uploaded file is assigned a unique name using a random index between 0 and 100000 (for example: file1KB_101 ), this should avoid confilcts and file locking issues.
Initially, using the same filenames led to 423 Locked errors, like HTTPError('423 Client Error: Locked for url: PUT /upload'), due to concurrent write attempts. Hence the introduction of randomized file names and the cleanup with the `on_stop` method.

Each .py script represents a different load testing scenario:

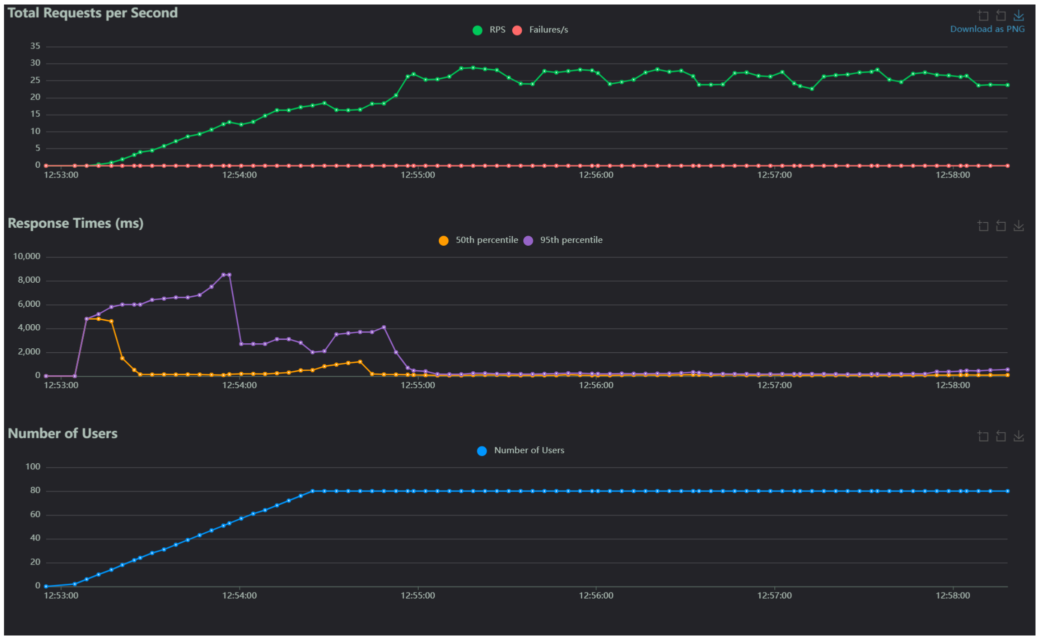
- `locust_tasks_light.py` : Light load with 1KB file uploads, with tasks occurring every 2 to 4 seconds.
- `locust_tasks_medium.py` : Medium load with 1KB and the addition of 1MB file uploads, with task requests every 1 to 2 seconds.
- `locust_tasks_heavy.py` : Heavy load with a mix of 1KB, 1MB, and 1GB file uploads, with a pause of 2 to 4 seconds between tasks.

Only the weights of upload tasks vary across these scripts to simulate different load intensities. All other task weights remain constant.

Information regarding locust test execution can be found in the project's README file.

## Test Results: Light Load

For the first test (light load), 80 users were simulated over a period of 5 minutes. The spawn rate was set to 1 user per second. The plots below show requests per second, response time percentiles, and the number of users over time:



As we can observe from the plots, the number of requests per second (RPS) increases steadily during the initial ramp-up period and then stabilizes between 25 and 30 RPS. The 90th percentile response times are higher at the beginning of the test touching 8000 ms, likely due to the additional overhead of spawning new users and establishing connections. Once all virtual users are active and the system reaches a steady state, response times stabilize and remain consistently low, below 100 ms, suggesting that the system handles the low load efficiently.

## Request Statistics

| Type | Name | # Requests | # Fails | Avg (ms) | Min (ms) | Max (ms) | Avg Size (bytes) | RPS | Failures/s |
|---|---|---|---|---|---|---|---|---|---|
| HEAD | /remote.php/dav | 932 | 0 | 147.93 | 18 | 25160 | 0 | 2.5 | 0 |
| PROPFIND | /remote.php/dav/files/[user]/ PROPFIND | 1341 | 0 | 268.94 | 29 | 33780 | 14299.31 | 3.6 | 0 |
| GET | /remote.php/dav/files/[user]/Readme.md | 1328 | 0 | 200.89 | 29 | 32107 | 206 | 3.57 | 0 |
| DELETE | /remote.php/dav/files/[user]/Testfile.md DELETE | 408 | 0 | 422.37 | 114 | 5937 | 0 | 1.1 | 0 |
| DELETE | DELETE /cleanup | 2237 | 0 | 1658.42 | 156 | 6599 | 0 | 6.01 | 0 |
| PUT | PUT /upload | 2646 | 0 | 438.05 | 87 | 35855 | 0 | 7.11 | 0 |
| **All** | **Aggregated** | **8892** | **0** | **653.01** | **18** | **35855** | **2187.24** | **23.88** | **0** |

Note: the DELETE /cleanup request is the operation run with the `on_stop` method and therefore separate from the DELETE operation performed as a test during runtime.

From these statistics, we notice that no requests failed during the simulation. The average response time across all request types was 653.01 ms. Upload requests (PUT) had an average response time of 438.05 ms, which, while higher than most other operation, is still manageable.

## Response Time Statistics

| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| HEAD | /remote.php/dav | 34 | 37 | 42 | 48 | 62 | 100 | 4000 | 25000 |
| PROPFIND | /remote.php/dav/files/[user]/ PROPFIND | 54 | 59 | 65 | 74 | 98 | 150 | 5200 | 34000 |
| GET | /remote.php/dav/files/[user]/Readme.md | 56 | 61 | 68 | 78 | 110 | 320 | 2900 | 32000 |
| DELETE | /remote.php/dav/files/[user]/Testfile.md DELETE | 200 | 220 | 250 | 330 | 550 | 2000 | 4100 | 5900 |
| DELETE | DELETE /cleanup | 1700 | 1800 | 1900 | 2100 | 2300 | 2500 | 4300 | 6600 |
| PUT | PUT /upload | 150 | 170 | 200 | 290 | 590 | 1600 | 5500 | 36000 |
| **All** | **Aggregated** | **140** | **180** | **430** | **1400** | **1900** | **2200** | **4500** | **36000** |

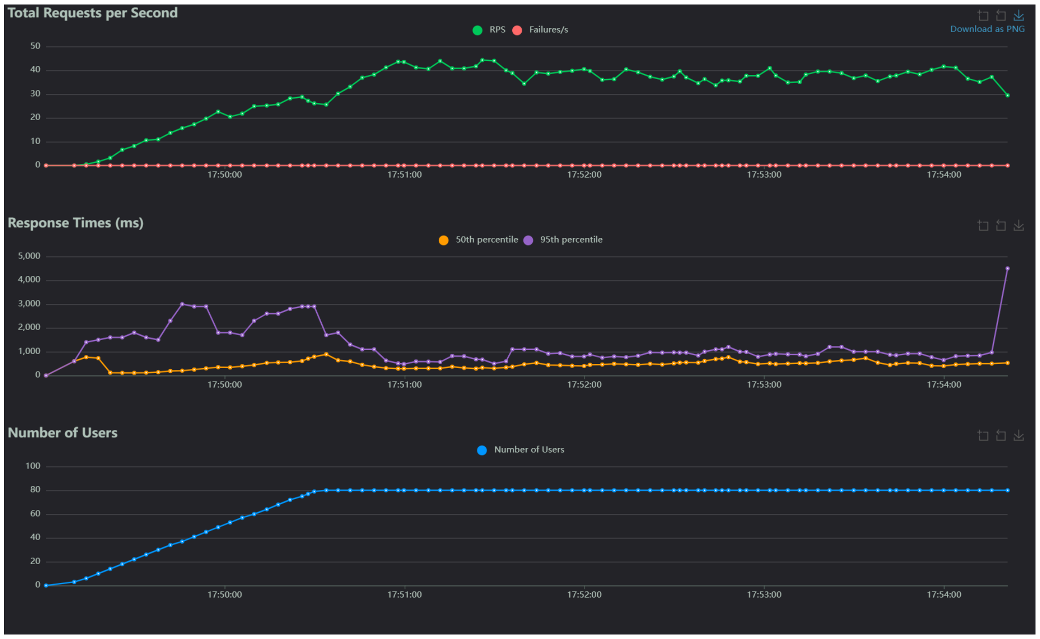
Looking at the percentile values, we see that for UPLOAD requests, the median response time (50th percentile) is 150 ms, and the 90th percentile is 590 ms. This suggests that the majority of upload requests are processed rapidly and only a small percentage experience higher latency, which could be attributed to temporary resource contention.

In the aggregated statistics, the median response time is 140 ms, and the 90th percentile is 1900 ms. The elevated values in the upper percentiles are most likely influenced by the DELETE /cleanup requests, which are more time-consuming, as shown by their own percentiles.

Overall, the system performs well under light load, with fast and consistent response times, no request failures, and good scalability during the initial phase.

## Test Results: Medium Load

For the second test (medium load), 80 users were simulated over a period of 5 minutes. The spawn rate was also set to 1 user per second. The results are displayed below:



We can observe, that in this case the number of RPS also increases steadily during the initial period and then stabilizes between 35 and 40 RPS, which is slightly higher than in the light load scenario. This is expected due to the shorter wait times between tasks, resulting in a higher frequency of requests.

Initially, the 90th percentile response times peak near 3000 ms, and then stabilize and remain around 1000 ms, which is still an acceptable value. This suggests that the system can handle a moderate increase in load reasonably well, although with additional latency compared to the light load test.

## Request Statistics

| Type | Name | # Requests | # Fails | Avg (ms) | Min (ms) | Max (ms) | Avg Size (bytes) | RPS | Failures/s |
|---|---|---|---|---|---|---|---|---|---|
| HEAD | /remote.php/dav | 916 | 1 | 346.24 | 21 | 10456 | 0 | 2.11 | 0 |
| PROPFIND | /remote.php/dav/files/[user]/ PROPFIND | 1360 | 0 | 423.45 | 31 | 28176 | 30067.17 | 3.13 | 0 |
| GET | /remote.php/dav/files/[user]/Readme.md | 1312 | 0 | 402.79 | 35 | 27506 | 206 | 3.02 | 0 |
| DELETE | /remote.php/dav/files/[user]/Testfile.md DELETE | 503 | 0 | 717.43 | 107 | 9540 | 0 | 1.16 | 0 |
| DELETE | DELETE /cleanup | 4115 | 0 | 2245.52 | 741 | 14818 | 0 | 9.48 | 0 |
| PUT | PUT /upload | 6137 | 3 | 631.7 | 85 | 22943 | 0 | 14.13 | 0.01 |
| **All** | **Aggregated** | **14343** | **4** | **1038.79** | **21** | **28176** | **2869.81** | **33.04** | **0.01** |

The average response time across request types was 1038.79 ms, which is higher than under light load but expected given the increased task frequency.
PUT requests averaged 631.7 ms, reflecting the presence of two different types of file to be uploaded . This still remains within an acceptable range for the medium load conditions.

## Failure Statistics

| # Failures | Method | Name | Message |
|---|---|---|---|
| 1 | HEAD | /remote.php/dav | RemoteDisconnected('Remote end closed connection without response') |
| 3 | PUT | PUT /upload | RemoteDisconnected('Remote end closed connection without response') |

The test produced 4 failures out of over 14,000 requests. All were caused by `RemoteDisconnected` , which typically occurs when the server closes the connection before sending a response. This can happen due to network issues, server overload, or timeouts. In this case, given the extremely low failure rate, the impact on the test is negligible.

## Response Time Statistics

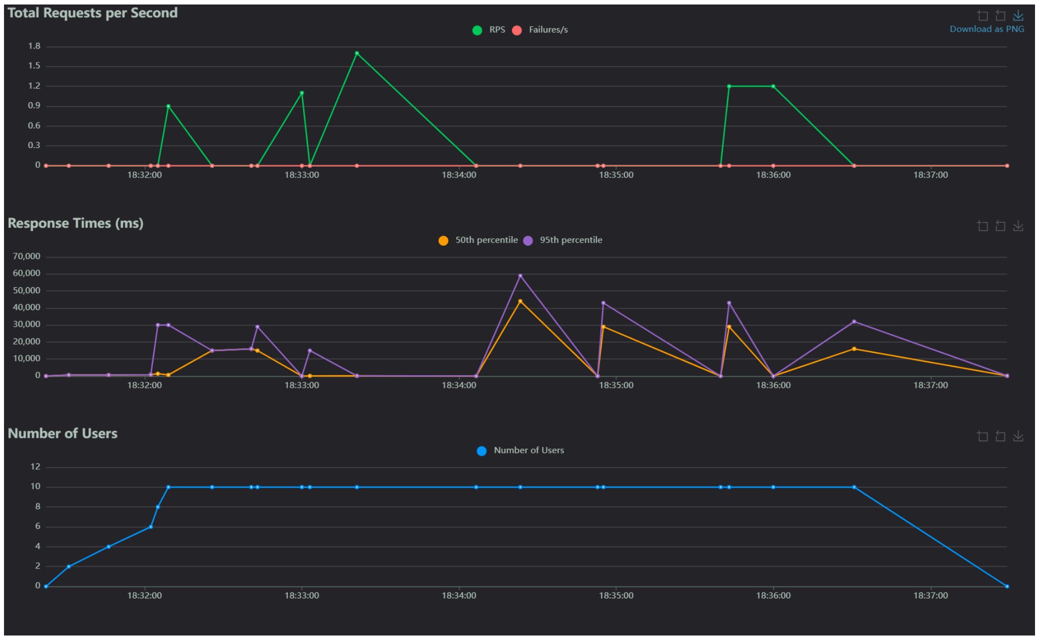
| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| HEAD | /remote.php/dav | 270 | 310 | 380 | 460 | 580 | 720 | 2200 | 10000 |
| PROPFIND | /remote.php/dav/files/[user]/ PROPFIND | 300 | 370 | 430 | 500 | 630 | 760 | 3800 | 28000 |
| GET | /remote.php/dav/files/[user]/Readme.md | 290 | 350 | 420 | 490 | 630 | 810 | 2600 | 28000 |
| DELETE | /remote.php/dav/files/[user]/Testfile.md DELETE | 630 | 720 | 790 | 910 | 1100 | 1300 | 1900 | 9500 |
| DELETE | DELETE /cleanup | 2000 | 2200 | 2400 | 2700 | 3300 | 4000 | 6300 | 15000 |
| PUT | PUT /upload | 550 | 610 | 680 | 780 | 930 | 1100 | 3400 | 23000 |
| **All** | **Aggregated** | **610** | **770** | **1200** | **1800** | **2400** | **2900** | **4900** | **28000** |

Looking at the percentile values, we see that for UPLOAD requests, the median response time is 550 ms, and the 90th percentile is 930 ms. This is not suprising, given the increased number of upload requests in this scenario.

In the aggregated statistics, the median response time is 610 ms, and the 90th percentile is 2400 ms. Generally, we can say the system performs adequately under medium load, with some increase in response times and a few request failures, but still maintaining a good level of service.

## Test Results: Heavy Load

The third test (heavy load), simulated only 10 users over a period of 5 minutes. The spawn rate was set to 1 user per second.
The reduced number of users was necessary for the system to function, as it could not handle more concurrent users while uploading large files (1GB). Below is the plot with the results:



As we can see, the RPS is noticeably lower than in the previous tests: below 2 RPS.
The plot is less smooth with many spikes and drops, this is probably due to the fact that with only 10 users and large file uploads, the system experiences more variability in request handling.

The 90th percentile response times are significantly higher and variable, peaking around 60000 ms at times. The graph remains spiky and does not stabilize around a consistent value. This indicates that under heavy load the system struggles to maintain low latency.

## Requests Statistics

| Type | Name | # Requests | # Fails | Average (ms) | Min (ms) | Max (ms) | Average size (bytes) | RPS | Failures/s |
|---|---|---|---|---|---|---|---|---|---|
| HEAD | /remote.php/dav | 15 | 0 | 20824.67 | 29 | 44194 | 0 | 0.04 | 0 |
| PROPFIND | /remote.php/dav/files/[user]/ PROPFIND | 16 | 0 | 9546.62 | 32 | 44195 | 4947.13 | 0.05 | 0 |
| GET | /remote.php/dav/files/[user]/Readme.md | 28 | 0 | 7352.60 | 34 | 28772 | 206 | 0.08 | 0 |
| DELETE | /remote.php/dav/files/[user]/Testfile.md DELETE | 7 | 0 | 136.75 | 123 | 147 | 0 | 0.02 | 0 |
| DELETE | DELETE /cleanup | 39 | 0 | 313.83 | 107 | 995 | 0 | 0.11 | 0 |
| PUT | PUT /upload | 46 | 0 | 19437.61 | 93 | 58741 | 0 | 0.13 | 0 |
| **All** | **Aggregated** | **151** | **0** | **10452.42** | **29** | **58741** | **562.4** | **0.43** | **0** |

The average response time was 10452.42 ms (10.45 seconds), which is significantly higher than in the previous tests, highlighting the strain on the system.

Notably, the upload requests were around 19437.61 ms (19.44 seconds), which is expected, as 1GB file uploads naturally require more time to complete. This is a substantial increase compared to the medium load test, indicating that large file uploads have a pronounced impact on performance.

### Response Time Statistics

| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| HEAD | /remote.php/dav | 24000 | 29000 | 29000 | 44000 | 44000 | 44000 | 44000 | 44000 |
| PROPFIND | /remote.php/dav/files/[user]/ PROPFIND | 1000 | 1400 | 16000 | 16000 | 30000 | 44000 | 44000 | 44000 |
| GET | /remote.php/dav/files/[user]/Readme.md | 600 | 1600 | 15000 | 16000 | 29000 | 29000 | 29000 | 29000 |
| DELETE | /remote.php/dav/files/[user]/Testfile.md DELETE | 140 | 140 | 140 | 140 | 150 | 150 | 150 | 150 |
| DELETE | DELETE /cleanup | 150 | 160 | 170 | 740 | 810 | 920 | 1000 | 1000 |
| PUT | PUT /upload | 15000 | 29000 | 29000 | 32000 | 44000 | 44000 | 59000 | 59000 |
| **ALL** | **Aggregated** | **680** | **1400** | **15000** | **29000** | **30000** | **44000** | **50000** | **59000** |

Response times were also significantly affected. For UPLOAD requests (PUT), the median response time (50th percentile) is 15000 ms (15 seconds), and the 90th percentile is 44000 ms (44 seconds). This indicates that while half of the upload requests are completed within 15 seconds, a significant portion experiences much longer wait times, with 10% taking up to 44 seconds.

Aggregated statistics show a median response time of 680 ms, but the 90th percentile jumps to 30000 ms (30 seconds). This wide range indicates that while some requests are handled quickly, others are severely delayed, likely due to the heavy load and large file sizes.

In summary, the system experiences significant latency and performance degradation under heavy load . While no requests failed, indicating correctness, the response time variability and long delays suggest that it is not currently optimized for scenarios involving larger data volumes.

## Scalability

As observed from the stress tests, as concurrent users, request frequency and upload file sizes increase the system's performances degrade. To address this, it’s essential to consider scalability strategies that ensure the system remains responsive under higher loads.

In cloud computing there are two main approaches to scalability: vertical and horizontal scaling.

Vertical scaling involves upgrading the existing server's hardware resources, such as increasing CPU, RAM, or storage capacity. This approach is relatively straightforward and can provide immediate performance improvements. However, it has limitations, as there is a maximum capacity that a single server can reach, and it may lead to downtime during the upgrade process.
Furthermore in the case of failure of the single server, the entire system would be unavailable.

Horizontal scaling, on the other hand, involves adding more servers to distribute the load and increase capacity. This is usually the preferred approach for scalable cloud-native systems. In the context of Docker and Nextcloud, this could mean running multiple instances of the Nextcloud container behind a load balancer, which distributes incoming requests across all available instances.

Horizontal scaling offers several advantages:

- Improved fault tolerance: If one server fails, others can continue to handle requests, ensuring high availability.
- Better resource utilization: Workloads can be distributed across multiple servers, preventing any single server from becoming a bottleneck.
- Flexibility: New servers can be added or removed dynamically based on demand.
- Reduced downtime: Updates and maintenance can be performed on individual servers without affecting the entire system.

For these reasons horizontal scaling seems like the best approach to ensure the system can handle increased loads while maintaining performance and reliability.

## Cost Efficiency

Currently, the system runs locally on a personal machine using Docker Compose, which incurs no direct cloud infrastructure costs. This approach is appropriate for development, testing, and small-scale demonstrations.

Cost efficiency is further maintained by:

- Setting per-user storage quotas to prevent resource overuse;
- Using lightweight containers;
- Removing test users and files after each run to reclaim storage.

In a production environment, however, this setup wouldn't suffice. To deploy the system some kind of scaling of the infrastructure would be necessary.
In this case, horizontal scaling would be the preferred approach, also from a cost-efficiency perspective.

The investment required for setting up and maintaining such a system overall would be lower than the one needed for vertical scaling, which would require more expensive, high-end hardware.

Some additional cost-saving measures could include:

- Storing underaccessed files in cheaper, long-term storage solutions;
- Monitoring resource usage with tools like Prometheus and Grafana to optimize resource allocation;
- Leveraging a more optimized open-source database solution, which could upgrade performance with no additional cost.

To summarize, a combination of horizontal scaling and cost-saving strategies could make the system efficient and affordable as it grows.

## Conclusions

In conclusion, this project successfully designed and implemented a cloud-based file storage system using Nextcloud and MariaDB, deployed with Docker Compose. The system provides essential file operations within private user spaces, along with password policies, security measures and user management capabilities.
Load testing with Locust under varying conditions revealed that the system performs well under light and medium loads, maintaining adequate latency and reliability. However, under heavy load, performance degraded significantly, indicating the need for further optimization.
To address these challenges, horizontal scaling with multiple Nextcloud instances behind a load balancer is recommended. This approach offers improved fault tolerance, better resource utilization, and flexibility to handle increased demand.