Cortinovis Nicola - SM3800042

07/09/2025

Abstract

The project covers the implementation of a cloud-based storage system with a set of basic features that give each user a private storage space where they can upload, download, and delete files. We begin by discussing the choices of software that compose the system, then the discussion shifts towards its security and scalability, the latter supported by empirical tests.

Design and Deployment

The software choices followed simplicity as a guiding principle. The suggested choice of Nextcloud for self-hosted file sharing and collaboration seemed the most natural and easy to use. While Nextcloud out-of-the-box version came with severe limitations, mainly that of SQLite for the database back end, it officially supported a lot of software that could solve such issues (PostgreSQL) or improve other areas (Redis). Furthermore, Nextcloud stood out because it also includes some built-in essential security features.

The deployment of the system happens through Docker. Docker allows for the containerization of each component, enforcing isolation and avoiding dependency conflicts. By using a `docker-compose.yaml,` the entire stack config is described in a single portable file, allowing for easy setup, streamlining updates, and, most importantly, making the deployment consistent and repeatable.

Nextcloud

I chose the latest stable image of Nextcloud available on Docker Hub. In this stack, Nextcloud handles user authentication, authorization, and roles. By default, two roles are defined: *regular* and *admin*. Regular users have access to their own private space, while admin users can also manage regular ones by deleting them, resetting their password, changing space quota, etc.

Inside the compose file, Nextcloud is exposed at `http://localhost:8080.` Volumes are created to make the data persistent (both configs for Nextcloud and user data directories). I also put a dependence on the "health" conditions of the other services to avoid booting Nextcloud with a database or a caching system that is not usable. Finally, the Nextcloud service is given a `PHP_MEMORY_LIMIT` of 2 GB to give a small but decent amount of RAM to the process (the default is 512 MB) and allow the handling of heavier operations.

To strip Nextcloud of some features that I did not find relevant to the project, such as the weather, user status and the default files, a `setup.sh` script can be run to both compose the container and make Nextcloud a little more lightweight.

PostgreSQL

Nextcloud default database SQLite struggles outside of simple testing environments because it's only capable of handling one write at a time. Having noticed that, I decided to substitute it with PostgreSQL. It is officially supported by Nextcloud and avoids slowdowns and errors by allowing multiple users to work at at once, therefore making the system more reliable and scalable.

Inside the compose file we can find a dedicated volume, the information needed to spin up the database on first startup, and the credentials to access it later on. Finally, a simple health check ensures that PostgreSQL is ready to accept connections.

Redis

Redis was added to improve Nextcloud's performance and reliability. While Nextcloud can run without it, this leads to slower page loads and problems when multiple users access the same file at once. With the addition of Redis, Nextcloud is equipped with fast in memory caching and file locking, preventing conflicts, reducing database load, and overall enhancing scalability.

As for the compose file, again we can find a volume to ensure persistence. A similar health check is also present for the same motives.

Security

Given the purely didactic purpose of this project, usual good practices security are not followed in favor of ease of use and a quicker setup. For example, you can find the `.env` file in the GitHub repository and some credentials directly defined inside the `docker-compose.yaml`.

As mentioned earlier, Nextcloud is responsible for user authentication; therefore most security aspects are handled by it. At its core the system gives an authenticated user a client token, stored only on the client device, and used to make all HTTP requests during the session.

Password Policies

To force users to choose more robust passwords, Nextcloud allows you to customize how severe the requirements are, such as requiring upper and lowercase characters, numeric characters, and special characters, and a minimum number of characters. More interestingly, it can also check that the password is not a common one and it's not present in the list of breached passwords provided by https://haveibeenpwned.com/.

Furthermore, Nextcloud has a brute force protection that prevents attempts to guess passwords by making requests coming from suspicious IP addresses slower or even outright rejecting them. If needed, IPs can also be whitelisted to bypass this protection layer.

For this project, password policies are the default ones (10 characters password, cannot be common, and cannot have already been breached)

Additional security features

Nextcloud also offers the possibility of increasing security by turning on the following features:

- Server-side encryption ensures files are only accessible via the Nextcloud interface
- Two-factor Authentication adding an extra step to the user authentication

On top of that, there is a logging system that can track suspicious activity, which can be analyzed via the admin interface and acted upon if need be.

These features significantly increase security but may impact performance, especially server-side encryption. For the scope of this project, they were not enabled in order to keep the system lightweight and accessible.

Scalability and Cost Efficiency

The concept of scalability is central to any system that expects increasing workloads over time, which is usually the desired outcome. Scalability refers to the idea that adding more resources allows the system to continue handling these workloads efficiently.

Vertical scalability

Vertical scalability implies making a node more powerful by adding more CPU, RAM, or disk resources. The immediate approach to scaling up the system would be increasing `PHP_MEMORY_LIMIT` inside the `docker-compose.yaml`, or assigning more resources to Redis and PostgreSQL.

Nextcloud also allows tuning storage quotas per user. For this project, quotas were left as unlimited, but in a real deployment they would likely be restricted, and scaling up disk resources could then become necessary to accommodate more users.

For this kind of system, however, vertical scaling is not the preferred approach, since it only postpones bottlenecks. Vertical scaling is better suited to high-performance computing, such as scientific simulations, where large shared memory and low-latency communication provide significant advantages.

Horizontal scalability

Horizontal scalability improves the system by adding more nodes. In practice, this could mean running multiple Nextcloud containers, each serving a portion of the users.

For a cloud-based file storage system such as the one described in this project, horizontal scalability is the preferred approach given its simplicity and flexibility. It also brings some intrinsic benefits:

- **Elasticity**, the ability to grow or shrink dynamically depending on workload. For example, the system may experience heavier pressure during peak hours, in which case more containers can be added automatically and then removed when demand decreases.
- **Fault tolerance**, the ability to continue operating even if one or more components fail. For example, if a node goes down, its traffic can be redirected to healthy nodes, ensuring uninterrupted service for users.

Horizontal scaling does come with downsides, mainly the added complexity of managing multiple containers and keeping them consistent across all instances. It also requires a mechanism to decide which container handles which requests, usually delegated to a load balancer. These constraints are responsible for a trade-off in efficiency when compared to vertical solutions.

Cost efficiency

Vertical scaling can easily become inefficient from a cost-efficiency point of view. While in the beginning it may cost less, high-end servers with many CPUs and large memory banks grow disproportionately more expensive while still being limited by bottlenecks

such as memory bandwidth or I/O.

Horizontal scaling, instead, makes it possible to use multiple smaller nodes that typically offer a better price/performance ratio. This is especially true when the scaling happens via cloud services thanks to their ability to be provisioned on demand, which helps avoid spending on too many resources that may remain idle outside peak hours. If the horizontal scaling happens through the acquisition of multiple machines up-front, the benefits are only marginal, and the savings mainly come from reduced power consumption.

Testing

The system was tested using Locust, an open source load testing tool that allows simulating traffic on a system to measure performance, scalability, and reliability. Considering that it is an analysis tool, it was not inserted in the container, as it is not essential for the stack. Refer to the github repository README.md for the instruction on how to run the tests.

Locust creates virtual users that act like real Nextcloud users by sending various requests such as listing files (PROPFIND), uploading files (PUT), downloading files (GET) and deleting files (DELETE). To do so, we must first create these users inside Nextcloud. For our test we fixed the number of unique users to 100.

Each of these tasks is assigned a weight, representing the frequency of the request (higher implies more frequent), plus we also define a wait time interval, which is intended to simulate the time a human takes to think and then carry out one of these tasks. Finally ,we must set the number of maximum concurrent users and their growth rate. Locust then randomly chooses the Nextcloud users in which to login and starts performing the tasks. It can be the case that a user is logged in more than once, leading to that user being more active than normal.

Table 1 - Locust task definition

Task name	Method	Description	Weight
list_root	PROPFIND	List files/metadata in the user root	2
upload_file	PUT	Upload a file (disk write I/O)	6
download_file	GET	Download a file (disk read I/O)	4
delete_file	DELETE	Delete a file (metadata update)	3

I decided to create three different tests, all test do the same tasks described above, but the number of virtual users and their growth rate, size of file and wait time changes according to the type of test.

Table 2 - Locust tests definition

Test name	Users (`-u`)	Growth rate (`-r`)	File size (MB)	Wait time (s)	Duration
Easy	100	10	1	3.0 – 6.0	5m
Throughput	200	20	1	1.0 – 1.5	5m
Bandwidth	20	5	500	3.0 – 6.0	5m

Helper function

For all tests a helper function is defined to catch error 423 Locked and error 429 Too Many Requests during DELETE and GET operations:

- Error 423 Locked happens when the file has not been released yet because it is still in use and we try to access it to do some other operation
- Error 429 Too Many Requests is due to NextCloud limiting requests for safety reasons.

These are errors that tend to disappear by waiting a small amount of time. The helper retries the operation 5 times, then gives up and reports the failure.

The helper also handles 404s which might happen during DELETE and GET operations when no file is present. We do not want to track that as a failure since in this context it only means that the virtual users have not yet uploaded a file, so there is nothing to DELETE or GET.

Easy test results

The aim of this test was to check if the system was healthy and capable of managing a relaxed environment with a moderate amount of users, but long wait times and small sized files.

Table 3 - Easy test request statistics

Type	Name	# Requests	# Fails	Avg (ms)	Min (ms)	Max (ms)	Avg size (bytes)	RPS	Failures/s
DELETE	DELETE /files/[user]/1.0MB	1260	0	236.46	23	12579	252.88	4.20	0.00
GET	GET /files/[user]/1.0MB	1628	0	184.98	25	11685	896.81	5.43	0.00
PROPFIND	PROPFIND /files/[user]/	865	0	237.47	37	11039	105986.70	2.88	0.00
PUT	PUT /files/[user]/1.0MB	2510	6	274.50	2	12078	0.00	8.37	0.02
	Aggregated	6263	6	238.46	2	12579	14922.10		

Table 4 - Easy test 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)
DELETE	DELETE /files/[user]/1.0MB	51	57	65	76	98	120	6200	13000
GET	GET /files/[user]/1.0MB	51	57	64	76	97	120	5700	12000
PROPFIND	PROPFIND /files/[user]/	74	82	91	110	140	170	5800	11000
PUT	PUT /files/[user]/1.0MB	97	110	120	140	170	220	6300	12000
	Aggregated	74	83	95	110	140	190	6000	13000

Table 5 - Easy test failures statistics

Failures	Method	Name	Message
6	PUT	PUT /files/[user]/1.0MB	RemoteDisconnected('Remote end closed connection without response')

Figure 1 - Easy test response time evolution

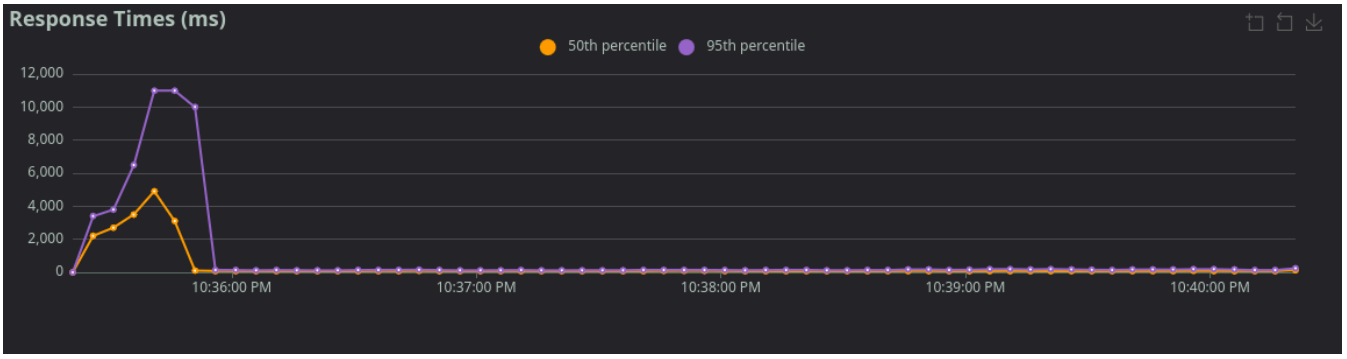
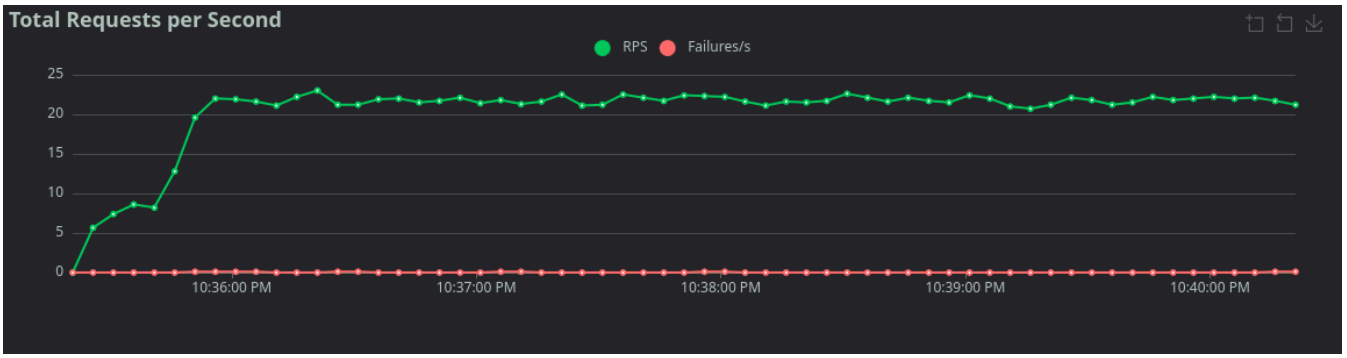


Figure 2 - Easy test request and failures evolution



Throughput test results

The aim of this test was to stress the system's ability to handle a high number of requests per second. This was achieved by simulating a larger user base with shorter wait times and frequent small file operations.

Table 6 - Throughput test request statistics

Type	Name	# Requests	# Fails	Avg (ms)	Min (ms)	Max (ms)	Avg size (bytes)	RPS	Failures/s
DELETE	DELETE /files/[user]/1.0MB	6105	468	476.48	25	137028	266.92	20.35	1.56
GET	GET /files/[user]/1.0MB	8237	593	439.79	27	135372	649.91	27.45	1.98
PROPFIND	PROPFIND /files/[user]/	4069	282	652.71	48	134851	173226.08	13.56	0.94
PUT	PUT /files/[user]/1.0MB	12287	900	492.40	52	137141	31.29	40.95	3.00
	Aggregated	30698	2243	496.37	25	137141	23201	102.32	7.48

Table 7 - Throughput test 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)
DELETE	DELETE /files/[user]/1.0MB	54	59	67	81	200	380	11000	137000
GET	GET /files/[user]/1.0MB	55	61	68	80	180	360	8000	135000
PROPFIND	PROPFIND /files/[user]/	100	110	120	140	210	380	10000	135000
PUT	PUT /files/[user]/1.0MB	110	120	140	170	300	450	5800	137000
	Aggregated	86	97	110	140	250	410	8100	137000

Table 8 - Throughput test failures statistics

Failures	Method	Name	Message
468	DELETE	DELETE /files/[user]/1.0MB	HTTPError('500 Server Error: Internal Server Error for url: DELETE /files/[user]/1.0MB')
593	GET	GET /files/[user]/1.0MB	HTTPError('500 Server Error: Internal Server Error for url: GET /files/[user]/1.0MB')
282	PROPFIND	PROPFIND /files/[user]/	HTTPError('500 Server Error: Internal Server Error for url: PROPFIND /files/[user]/')
900	PUT	PUT /files/[user]/1.0MB	HTTPError('500 Server Error: Internal Server Error for url: PUT /files/[user]/1.0MB')

Figure 3 - Throughput test response time evolution

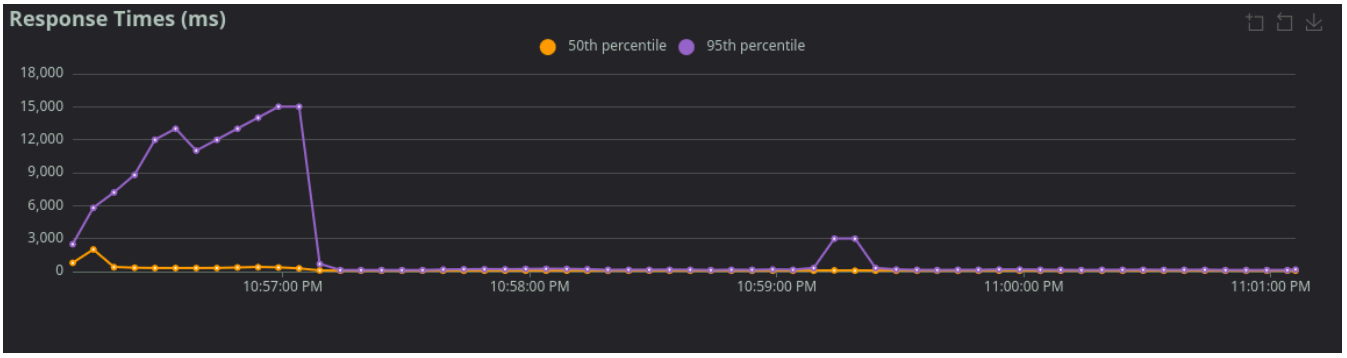
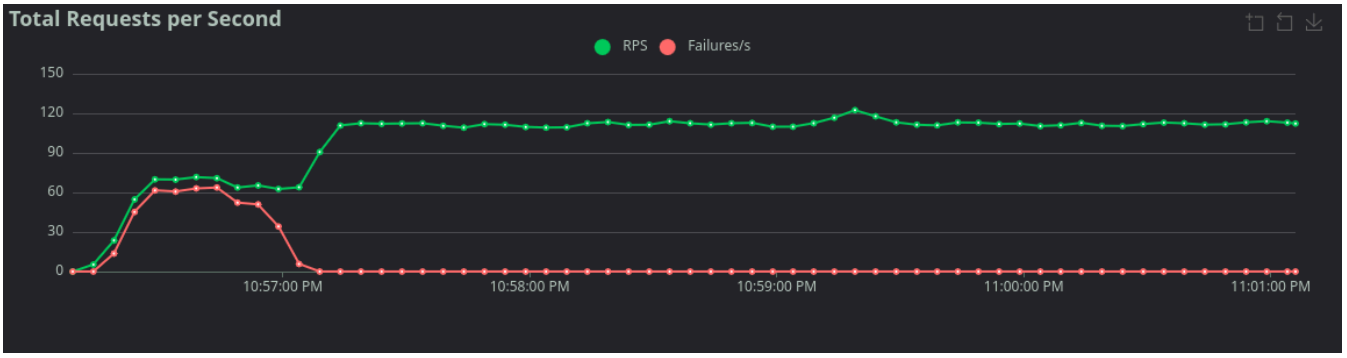


Figure 4 - Throughput test response time evolution



Bandwidth test results

The aim of this test was to examine the system's behavior when managing very large files. In this scenario, the number of users was kept low, but each operation involved files of 500MB. The purpose was to mainly stress disk I/O.

Table 9 - Bandwidth test request statistics

Type	Name	# Requests	# Fails	Avg (ms)	Min (ms)	Max (ms)	Avg size (bytes)	RPS	Failures/s
DELETE	DELETE /files/[user]/500.0MB	50	0	1358.43	45	9672	254.90	0.17	0.00
GET	GET /files/[user]/500.0MB	82	0	818.50	46	8773	254.96	0.28	0.00
PROPFIND	PROPFIND /files/[user]/	36	0	1286.12	72	8576	119734.69	0.12	0.00
PUT	PUT /files/[user]/500.0MB	83	6	45990.45	3983	53898	0.00	0.29	0.02
	Aggregated	251	6	15930.47	45	53898	17307.18	0.86	0.02

Table 10 - Bandwidth test request 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)
DELETE	DELETE /files/[user]/500.0MB	170	500	810	2400	5400	8900	9700	9700
GET	GET /files/[user]/500.0MB	120	150	230	1200	2300	4600	8800	8800
PROPFIND	PROPFIND /files/[user]/	530	750	1200	2400	4600	6000	8600	8600
PUT	PUT /files/[user]/500.0MB	48000	49000	50000	51000	52000	53000	54000	54000
	Aggregated	920	2500	39000	48000	50000	52000	53000	54000

Table 11 - Bandwidth test request statistics

Failures	Method	Name	Message
6	PUT	PUT /files/[user]/500.0MB	RemoteDisconnected('Remote end closed connection without response')

Figure 5 - Bandwidth test response time evolution

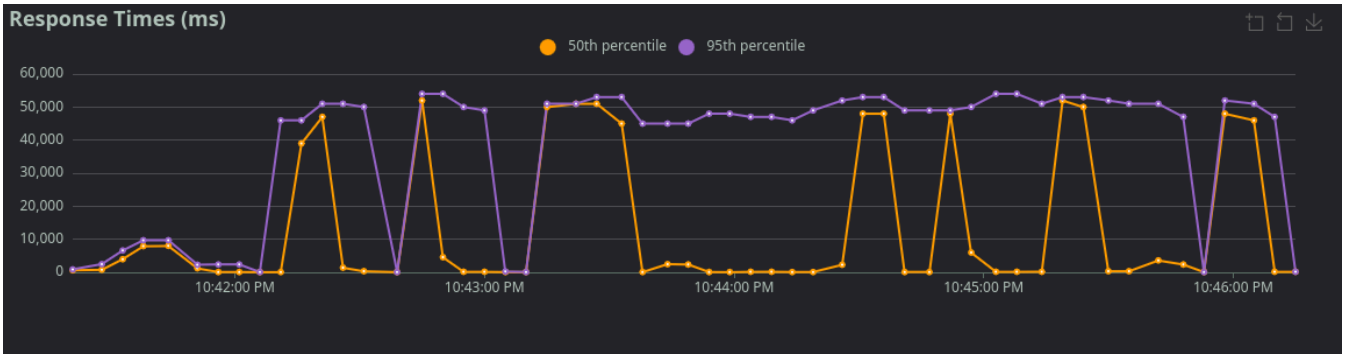
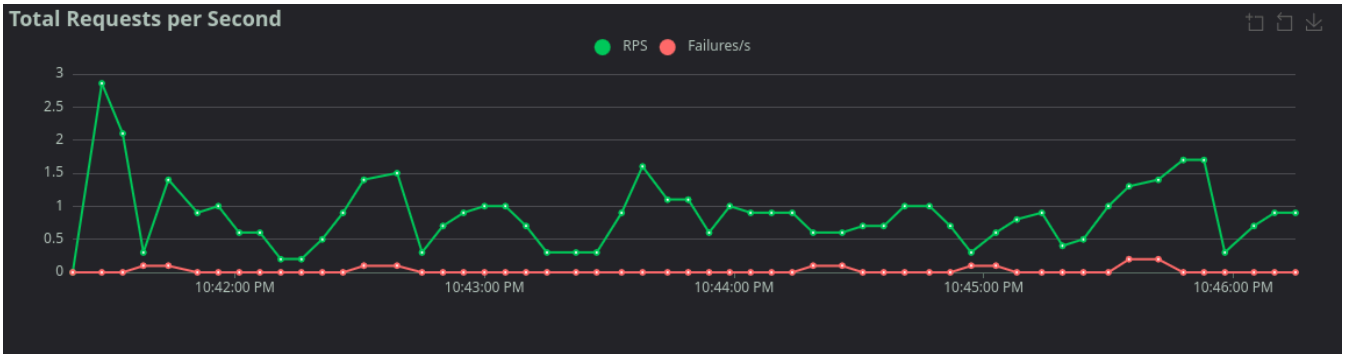


Figure 6 - Bandwidth test response time evolution



Conclusions

From the results described above, when the stack is in a relaxed environment, such as the one simulated in the easy test, it remains healthy and stable, with only 6 failures over more than 6000 requests. These failures happen only during PUT operations and are of type `RemoteDisconnected('Remote end closed connection without response'),` meaning that the server dropped the connection before completing the request. Still, the rarity of the event makes it not too alarming. Regarding response time, when the system is still spinning up, we can observe that the 95th percentile is relatively high, but this behavior is expected due to the user still spawning.

When the environment changes to one where the number of requests per second is increased, for example due to the users having a low waiting time, the system begins to struggle. While the average response time is still reasonable, the system takes longer to spin up, so the very heavy task linger longer. Furthermore, the failure rate is very high in the beginning, showing a lot of `HTTPError('500 Server Error: Internal Server Error for url: DELETE /files/[user]/1.0MB')` indicating server side bottlenecks. In fact, during the throughput test, the container exhausted the CPU resource, mainly due to the Nextcloud container.

Finally, in another environment with fewer users but heavier files, the system is well behaved. The bandwidth test shows, as expected, that the bulk of the response time is due to PUT operations, which are also the most frequent and therefore regularly happen throughout the test. Given the very high response time, the request per seconds are very low. Coincidentally, the failures are the same as the ones observed during the easy test, however, while still small, here they're more significant due to the lower number of requests (251).

Overall, the system performs reliably for everyday, low-intensity usage but struggles when pushed either in request volume or data size, suggesting that scaling up (possibly horizontally) and further fine-tuning of the system would be required.