Cloud Project Report

Matteo Munini

September 2025

Cloud Basic Task

1 Introduction

The goal of this project is to design and deploy a scalable, secure, and efficient cloud storage system, utilizing *Docker* for containerization, proposing *Nextcloud* for storage and user management, and *Locust* has been chosen for executing performance testing on the service. The platform allows users to upload files, download and manage operations, ensuring reliability and scalability under varying workloads.

2 Management of the authentication and authorization for user

Nextcloud defines the central framework of the cloud solution in this project, and it offers advanced functionalities for user identity management, access control, and file handling. As a self-hosted open source platform, it provides secure, flexible, and reliable tools for managing user data. By extending the innate features of Nextcloud, the system, in this project, incorporates customized user management and file operation processes in order to support usability and efficiency.

2.1 Authentication and Authorization of an user

Strong authentication and well-defined authorization are vital to ensure both security and a smooth user experience. Within this system, users are able to:

- Register, authenticate, and terminate sessions through the NextCloud Web interface.
- Modify credentials and personal settings, giving them control over their own accounts.

The admin user holds elevated privileges, while standard users retain limited storage-focused rights (operating under role-based permissions). This structure setup protects resources while providing flexibility in how users access the platform.

In the GitHub repository, to simulate a common scenario, a script is provided:

- `script_sign_in_new_user.sh` creates 200 users by defining password, email, username, and signing in to the system under a user group named "common" with limited and fixed characteristics.

3 Management of File operations

The system provides comprehensive support for file management tasks, including uploading, downloading, and deleting files within private storage. These operations can be executed by a user through the Nextcloud web interface (in real world scenario).

A user could also interact from the terminal, indeed thanks to this opportunity it has been possible to efficiently run a simualtion.

In the GitHub repository, to handle more easily the simulation of a common scenario, two script are provided:

- `script_user_tries_upload_files.sh` creates synthetic test files of different sizes for performance testing that will be used in the simulation run with Locust. The files have dimension 1KB, 1MB and 1GB.
- `delete_after_test.sh` deletes the users created for simulation.

4 Scalability

MariaDB runs with READ-COMMITTED isolation to minimize lock contention and support concurrent access. Each service is containerized with defined resource limits (4GB RAM, 2 CPU cores). APCu caching reduces database load by storing frequently accessed data in memory.

4.1 Theoretical modification to ensure Scalability with condition of increased load and traffic

In theory, managing scalability and sustaining performance under heavier traffic requires a combination of strategies that work together:

- Horizontal Expansion: Running the application across several servers or containers allows requests to be spread evenly, preventing overload on any single machine. A load balancer ensures fair distribution and makes it possible to add or remove capacity as demand fluctuates.

- Vertical Upgrades: Increasing the processing power, memory, or storage of individual machines can provide an immediate performance boost. However, this method has physical and cost limitations, which makes it more of a short-term measure compared to horizontal expansion.
- Effective Caching: Using multiple layers of caching—such as in-memory systems like Redis or Memcached for rapid data retrieval, and reverse proxies for frequently requested web content—reduces pressure on databases and application servers, improving both speed and efficiency.
- Database Efficiency: Refining queries, applying proper indexing, and monitoring query execution times helps the database serve more requests in less time. For larger systems, replication, partitioning, and pooling connections are essential techniques to maintain responsiveness under high concurrency.
- Background Task Handling: Moving time-consuming or non-essential processes into background job queues (using tools like Kafka, RabbitMQ, or Celery) allows the main application to stay responsive, while heavy operations are processed asynchronously.
- Content Delivery Networks (CDNs): Offloading static files such as images, videos, and scripts to geographically distributed servers reduces latency and frees core infrastructure to focus on dynamic workloads.
- Monitoring with Elastic Scaling: Real-time monitoring of key performance metrics—like latency, throughput, and error rates—combined with automated scaling policies ensures the system can adapt to unexpected traffic surges without manual intervention.

5 Security Considerations

5.1 Protected File Storage and Transfer

User data is safeguarded by allocating each regular user a private storage space, fully isolated from other accounts to maintain confidentiality. Administrators can adjust the storage space of each user dynamically, ensuring balanced use of system resources and preventing misuse. For enhanced protection, **server-side encryption** can be enabled through the Nextcloud interface, securing files at rest. In combination with **TLS/HTTPS protocols**, this ensures that data in transit is protected from interception or tampering.

5.2 Robust Authentication Mechanisms

The platform employs a **credential-based login system**, with passwords required to meet complexity rules such as including uppercase letters, numbers, and special symbols. Also, I have setup as constraint that each password expires in 30 days, so each user has to dynamically modify his/her own password. This reduces the risk for credentials to be discovered, since validity of those is limited. To strengthen security, the system supports **two-factor authentication (2FA)**, requiring users to provide an additional verification factor beyond the password. Furthermore, the **Nextcloud Registration app** supports secure self-service account creation with **email verification**, mitigating the risk of unauthorized or fraudulent accounts. Indeed, in this project I have set a single only trusted domain, by simulating a real world scenario where this system is built for internal industry usage.

5.3 Unauthorized Entry

Multiple layers of defense are applied to minimize the possibility of intrusion:

- Since users are divided into administrators, standard and other groups of users, each with distinct access privileges, this makes unauthorized access, by principle of least-privilege access.
- Using Nextcloud's File Access Control, administrators can define detailed rules for file visibility, sharing, and modification, providing fine-grained oversight.
- Built-in mechanisms detect repeated failed login attempts and introduce delays, reducing the effectiveness of automated or malicious login attacks.
- Built-in brute force protection, so the IPs could be added to the white list.

6 Trade off between cost and efficiency

If the project were to be deployed online, a range of expenses would need to be considered, like:

- hosting
- domain registration
- storage
- costs associated with third-party services.

The exact expenses will depend on the selected hosting provider and specific service configurations, making it important to evaluate these factors carefully during the initial phase of the project. If the design of the project is not well done either the excessive cost would shrink the budget and so limit the deployment of the service or it would make the solution not economically sustainable in the long term.

6.1 Optimization for cost efficiency

There are cloud providers that offers flexible pricing models and tools that can help to minimize operational expenses while maintaining performance. Migrating to a cloud platform such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP) would be a possible solution

To ensure cost efficiency, the system should be designed with scalability and resource optimization in mind. AWS Cost Explorer, Azure Cost Management, Google Cloud Billing are tools that allows the management to keep track of any kind of expenses and this allows an overall control over the system. These provides offer different tools that minimize operational expenses, like:

- AWS Lambda service, EC2
- AWS Glacier, Azure Blob Storage Archive
- AWS Aurora Serverless, Google Firestore
- AWS s3

7 Deployment of the Project

There is a ReadMe in the GitHub folder BASIC where it is explained how to deploy the service on a local machine. Here, a quick summary. However, firstly it is important to say that the code has been defined and tested on Windows OS and so, working on WSL which has some limitations. So, in other OS some customization would not be necessary.

- Requirements: Docker and Docker Compose. If using Windows, it is important to have also Docker Desktop.
- Cloning: Clone the repository and move to BASIC folder and then move into `nextcloud_docker` directory.
- Run Docker: Type the following command `docker-compose -f docker-compose.yaml up -d` and this will read the service definitions from the manifest, then it will creates and start the defined containers. It will run them in the background.
- Access Server UI: Once the container are created and up, then the user can access to the instance (here used NextCluoud) by opening a web browser, like Chrome, DuckDuckGo, Mozilla FireFox ... and navigate at `http://localhost:8080`. There, the user will need to login using credentials provided in the manifest. (for admin role, the username is admin and psw is admin too.)
- Locust Simualation: The user needs to create the syntethic users before runnign the experiment, using the bash file provided. Then, by running this code, the service definitions from the manifest will be read and, the

defined containers will be started and run. It will run them in the background. `docker-compose -f docker-compose-performance-test.yaml up -d`. By opening a web browser, the user will customize through Locus UI the experiment, by navigating at `http://localhost:8089`.

It could be that, conditional on host machine that will be used or personal goal, the user needs to make some personal modification to meet specific requirements. The files to be reviewed are:

- docker-compose.yaml for docker container setup
- docker-compose-performance-test.yaml for the locust configuration of the test
- `run_test_locust.py` for the loading operations to do

To monitor the behaviour of the service it could be used, loggin as Admin, the built-in dashboard of NextCloud. It allows the user to look at data like usage, performance of different kind, and status.

Since everything is built up with Docker, the user can monitor the activity by using the *stats* command, which in real time shows CPU, memory, and network utilization for each container.

7.1 Choice of Cloud Provider to migrate this project

For my little experience, I would use the services offered by AWS, since tools like EC2 and S3 allows NextCloud service to scale seamlessly with growing users and data, supported by the service of Auto Scaling to avoid over-provisioning. To improve the availability of the service, on AWS the deploy can be across multiple AWS Regions and Availability Zones for a fault-tolerant architecture. Since the service is related with storage of users' data, the access and data can be protected with IAM service, so to have also encryption at rest/in transit, that meet standards like GDPR, HIPAA, ISO. For the performance point of view, tools like CloudFront (CDN) and Elastic Load Balancing could be used to improve global access and to reduce latency respectivelly, to have a smoother collaboration in industry activity. Finally, the very rich ecosystem offered by AWS, like different kind of database's solutions (Aurora, RDS, ...), monitoring tools like CloudWatch, data analysis services ... could make easier and more robust the future improvements of the service.

8 Testing the service

The system was tested to observe its behavior under load and during input/output operations. Several actions were carried out, including uploading files of different sizes (from 1 KB to 1 GB), uploading images, listing files, and reading documents, in order to verify core functionality. To simulate realistic usage, the Locust tool was employed, gradually increasing the number of concurrent users up to 15. As shown in the image below (Figure 1), the system managed up to 15 users with a limited number of errors, suggesting a reasonable level of responsiveness under medium load conditions. Throughout the tests, indicators such as requests per second and response times were monitored, providing useful data on efficiency and reactivity. Overall, the results are encouraging while naturally subject to the limitations of the test environment, local machine, and indicate that the solution can deliver adequate performance for moderate usage scenarios.

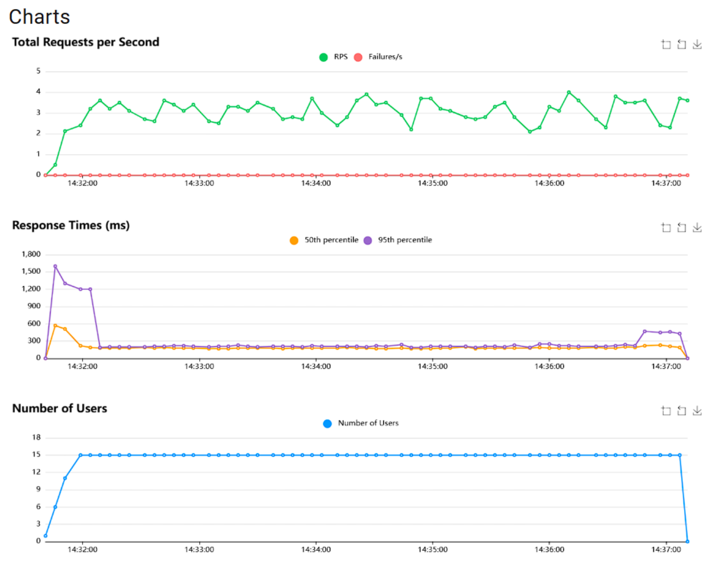

Figure 1: System performance results

9 Conclusion

In conclusion, the development and deployment of the cloud-based file storage system reached a functional stage and was successfully tested on a local machine. The results indicate that the selected technologies and methodologies are suitable for building such a solution, while also highlighting areas where further refinement would be possible. The project offered practical exposure to cloud computing, containerization, and system deployment, which helped broaden my understanding and improve my skills in these fields. Overall, the experience has been valuable from both a technical and a learning perspective, even though the system remains a prototype with room for future improvement.

Cloud Advance Tasks

10 Same service but on Kubernetes

This section outlines an advanced deployment of a cloud-based storage system with Nextcloud, using Kubernetes and Helm for orchestration and simplified management. A single-node Kubernetes cluster was set up locally, using Minikube solution, to run a complete solution with persistent storage, load balancing, and ingress control. The architecture includes MariaDB for the database, Redis for caching, MetalLB as a load balancer, and NGINX Ingress for external access. It has been provided thinking at a real worls scenario, however even if everything works fine this service did not work on WLS (NGINX Ingress). It has been provided another solution customized for the issue with WLS. As reported above, Minikube was used as the Kubernetes distribution, offering a lightweight yet realistic environment for testing and validation. The driver used is with Docker. This means that Minikube will run the Kubernetes node as Docker containers instead of full-built VM.

This makes the cluster lighter, faster, and easier to manage. This setup allows developers to explore scaling, troubleshoot issues, and refine configurations before moving to production. The strategy applies infrastructure-as-code principles through manifests, Helm charts, and declarative files to ensure reproducibility and maintainability.

11 Deployment

The requirements for running this service on local machine are the installation of Minikube, Helm and Kubectl (Kubernetes command line tool). This code have been defined and tested on a host machine that has Windows as OS (WSL downloaded) and so, every command or operations run has been executed on Ubuntu 24.04 WSL. Some operations done here, maybe not necessary for other OS. At the beginning, the user needs to clone this GitHub repository. Then, open the terminal and move to the directory where it has saved the repository. Everything needed is on the `deploy_nextcloud` directory.

On the *ReadMeAdvanceProject1* there are all the steps and commands to run in order to build up the service.

12 Description

Since the system was deployed on Minikube, the *replicaCount* parameter was limited to 1, and the Nextcloud image was pinned to version 27 to ensure stability and reproducibility. The Nextcloud deployment was managed through a customized *values.yaml* manifest.

Authentication credentials were handled via Kubernetes Secrets. A secret named *sys-cloud-secret* was referenced for the Nextcloud administrator account, while additional secrets (*mariadb-secret* and *redis-secret*) stored sensitive database and caching credentials. This choice prevents the use of plaintext passwords in the configuration and follows Kubernetes best practices for security.

Persistent data storage was enabled by linking the chart to an existing PersistentVolumeClaim (*nextcloud-pvc*) with a fixed capacity. This ensures that user files and application data remain intact across pod restarts or redeployments. The database integration was configured to use an external MariaDB instance, with all required connection parameters and credentials injected from the *mariadb-secret*. It has been necessary, to avoid conflicts, that the internal SQLite database option was explicitly disabled.

Redis was deployed in standalone mode with authentication enabled. Its persistence was disabled to simplify the initial setup, reflecting the prototype nature of the environment. The configuration ensured that only one Redis master pod was created, avoiding issues related to replicas in the single-node cluster. It has beend used to speed up file access, database queries, metadata lookups and to reduce the load on MariaDB by handling temporary data in memory.

MetalLB was integrated for function of load balancer, mapping hostname to a routable local IP address. Trusted domains included nextcloud.local, local addresses, and a wildcard entry (*) to provide flexibility during development and testing.

Finally, readiness and liveness probes were defined with an initial 60-second delay to allow for Nextcloud's startup time. These probes enable Kubernetes to automatically detect failures and restart unhealthy pods. All resources were deployed under the dedicated namespace **cloud**, improving organization and isolation from other workloads in the cluster.

The greatest difference with the previous deployment of the NextCloud service stands on the fact that now all the services needed are no more single and isolated containers, but everything is inside a Kubernetes ecosystem, meaning there are Pods orchestrated by k8s. The set up is more complex and more time consuming, however it allows to build something that is more complex, scalable, and that could adapt to new external requirements.

As required, it has been provided resources named PersistentVolume (PV) and PersistentVolumeClaim (PVC) to provide persistent storage for both the MariaDB database and the Nextcloud service. Specifically, hostPath volumes were created by mapping directories on the host machine (*/mnt/data/mariadb* and */mnt/data/nextcloud*) to persistent volumes within the Kubernetes cluster. In this way of deployment of the service the user data and database records survive pod restarts and redeployments, which is critical for a storage service. However, this solution remains a very basic setup, suitable only for a local testing environment. It comes with several inherent limitations: persistent volumes must be manually created before the deployment, meaning dynamic provisioning is not possible; and because hostPath volumes are bound to a specific node, the pods consuming them must also run on that node. This constraint makes the deployment less flexible and prevents scalability across multiple nodes. Such a configuration works for this prototype, single-node Minikube cluster, but it is not appropriate for production usage. In a real-world scenario, a distributed storage backend (e.g., Ceph, GlusterFS, or a cloud provider's block storage) would be required to overcome these limitations and ensure high availability, scalability, and data durability.

13 Steps that should be taken to have the service in high availability

The idea to move this deployment to a like real-world scenario keeping in mind the task of increasing the availability would require at least the achievement of these points:

- A more robust load balancer is needed, and from my little experience it could be interesting and economical feasible to use a load balancer offered by cloud providers (for example AWS ELB/ALB, GCP Load Balancer, or Azure Load Balancer). They would provide high availability, health checks, SSL termination, and automatic scaling of load balancing capacity. For on-premises environments, more robust solutions such as HAProxy or Traefik could be integrated to handle complex routing, failover, and security requirements. These improvements would ensure that external traffic is distributed efficiently across multiple replicas and nodes, minimizing downtime and improving user experience under high load.

- The deployment would need to move from a single-node setup to a multi-node Kubernetes cluster. This would allow multiple replicas of the services to run across different nodes, supported by a more advanced load balancing system. Running multiple replicas not only improves fault tolerance but also increases throughput by serving more requests in parallel. The addition of a Horizontal Pod Autoscaler (HPA) would further enhance scalability by dynamically adjusting the number of replicas based on CPU, memory, or custom application metrics, ensuring efficient resource usage under fluctuating workloads.

- Monitoring and reliability could be further improved by enabling the built-in Prometheus metrics. By collecting detailed metrics on application performance, resource usage, and request handling, Prometheus can serve as the foundation for proactive monitoring and alerting. Indeed, also with the advance of ML in forecasting and anomaly detection this could helps to built a more dynamical and resilient service.

- A production-ready solution would require replication and failover (also stronger security features such as data encryption, even if it is not the argument of discussion in this section). Service like Ceph or GlusterFS could be used.

14 Conclusion

The Kubernetes deployment can be seen as a more complex alternative to the Docker-based setup. While it requires additional resources and effort to configure, it offers greater scalability and resilience. Services can be scaled more easily, and the orchestration capabilities of Kubernetes provide stronger management of workloads compared to a simpler Docker setup.

At the same time, the added complexity is not always justified. For smaller environments with limited resources, a Docker-based deployment may be more practical, as it is easier to manage and less demanding in terms of infrastructure. Kubernetes, on the other hand, has higher requirements and a steeper learning curve.