Annalisa Paladino
SM3800021

Cloud-Based File Storage System

Introduction

This report outlines a strategy to implement a cloud-based file storage system, this report aims to present a robust framework that ensures a seamless, secure, and scalable user experience.
This project proposes a containerized deployment approach, utilizing Docker and Docker Compose, to facilitate ease of deployment and system manageability. Nextcloud has been chosen to guarantee the system's performance, reliability, and cost-effectiveness.
Moreover, for the test Locust has been used.

Manage User Authentication and Authorization

Nextcloud supports by default user authentication and authorization through its integrated features. In fact, users can sign up, log in and log out smoothly.
Nextcloud allows the creation of different roles between the users, there is an admin and multiple regular users: each user is provided with a private storage space, in this particular case 1GB has been assigned to each user.
This distinction allows the admin to have a greatest control over other users, in fact the admin can add, remove and modify regular users.

Manage File Operations

Nextcloud allows users to manage their files within their private storage space effectively. Specifically, users have the capability to upload files to this area, ensuring that they can securely store their data in the cloud. Additionally, they can easily retrieve or download these files from their private storage whenever necessary, providing convenient access to their information. Furthermore, users have the control to delete any files from their private storage, allowing for the management and organization of their stored data according to their needs.

Address Scalability

Design the system to handle a growing number of users and files.
At the base of a scalable system there is its ability to accommodate growth in number of users and data volume without compromising on performance or reliability. The goal is to ensure the system remains efficient, reliable, and responsive as demand escalates, the proposed system is a Nextcloud with MariaDB.
By dividing the system into a set of small services it’s possible to overcome problems related to the growth of users, this way each service can be scaled independently, allowing for more precise resource allocation based on demand for specific functionalities.
Moreover, the implementation of stateless application servers where possible could be a possible solution, this allows any server to respond to any request, making it easier to

distribute traffic across multiple servers without worrying about local state. It simplifies horizontal scaling since adding more servers increases capacity without additional complexity. The use of distributed databases that can scale horizontally could be an option, partitioning data across multiple nodes (sharding) can help distribute the load equally and improve response times.
Leverage cloud-based resources that automatically scale up or down based on real-time demand might be useful, in fact services like AWS Auto Scaling or Kubernetes' horizontal pod autoscaler can adjust resources dynamically, ensuring that the system meets current demand without over-provisioning.

Discuss theoretically how you would handle increased load and traffic.
To handle increased load and traffic some techniques can be introduced, for example the use of load balancers to equally distribute incoming requests across the servers can prevent any single server from becoming a bottleneck, thus maintaining system responsiveness under heavy load.
Implementing caching at various levels to reduce the load on the backend minimizes the need to compute or retrieve the same information repeatedly, significantly reducing response times and server load. Deploy Content Delivery Networks (CDN) to serve static content (images, CSS, JavaScript) closer to the users reduce latency by caching content in distributed servers, ensuring faster access for users.
Continuously monitor system performance metrics (CPU usage, memory usage, response times) to identify trends and predict when to scale can help, auto-scaling can be used to adjust the number of active servers automatically based on current load.
Moreover, regularly analyze and optimize database queries to reduce latency and increase throughput can improve the performance.

Address Security

Implement secure file storage and transmission.
To implement security measures for file storage and transmission it's possible to use strong *encryption* standards, such as AES-256, to encrypt files before they are stored. This ensures that even if data is accessed by unauthorized parties, it remains unintelligible without the corresponding decryption key.
To secure data transmission an option could be using protocols such as TLS (Transport Layer Security) to encrypt the data exchanged between the client and the server, this protects the data from being spied on, changed, or faked.
Moreover, the encryption of keys in a dedicated hardware security module or a managed key service allow to regularly rotate keys and enforce strict access controls to minimize the risk of key compromise.

Discuss how you would secure user authentication.
To secure user authentication it's possible to implement *Multi-Factor Authentication* (MFA), this method requires users to provide two or more verification factors to gain access to their accounts, MFA significantly reduce the risk of unauthorized access due to compromised

passwords. Enforcing strong password policies that require *complex passwords*, including a mix of letters, numbers, and special characters is also a good practice.

Discuss measures to prevent unauthorized access.
Nextcloud allows the creation of different roles between the users, there is an admin and multiple regular users, this allows to assign permissions based on the principle of least privilege, it ensures that users have access only to the resources necessary for their role, minimizing the potential impact of compromised accounts.
It's possible to use intrusion detection systems (IDS) and intrusion prevention systems (IPS) to monitor and control incoming and outgoing network traffic, moreover some regular security checks should be done.

Discuss Cost-Efficiency

Consider the cost implications of your design.
Cloud services offer scalability and flexibility but come with variable costs based on consumption. Continuous maintenance, including software updates, security patches, and system monitoring, incurs costs. Operational efficiency and automation level directly impact these expenses. The cost of storing and managing data can escalate quickly, especially with increasing data volumes, efficient data management practices are essential to keep these costs in check. Data transfer costs can be significant, especially for systems that involve high data ingress and egress volumes.

Discuss how you would optimize the system for cost efficiency.
For cost efficiency the system could be optimized by using auto-scaling capabilities to dynamically adjust resources based on demand, this ensures you're not paying for inactive resources during low usage periods and can handle peak loads effectively.
The system can be designed to minimize unnecessary data transfers to reduce networking costs.
Containers and microservices can improve resource utilization and reduce overhead by allowing multiple services to share the same underlying infrastructure without interference, this approach also facilitates more granular scaling and can lead to cost savings.
A continuous monitoring over resource usage and cost trends with some tools and services that provide detailed billing insights can help identify and eliminate wasteful spending, also some budget alerts can be implemented to proactively manage costs.
Last but not least, when designing the system it's crucial to have in mind cost-efficiency from the start, this includes choosing the right algorithms, minimizing dependency on external services, and using asynchronous processing where appropriate to reduce computing resource requirements.

Deployment

Provide a deployment plan for your system in a containerized environment on your laptop based on docker and docker-compose.
To set up the environment Docker and Docker Compose have to be installed and running on the laptop. Then create a Dockerfile, it specifies the environment, dependencies, and

commands needed to run the service, the docker file put together tge Nextcloud and the MariaDB instances. Define a docker-compose.yml file, this file describes the application's services, networks, and volumes. Include each service in the application, specifying the image to use, ports to expose, and volumes for persistent storage.
For the deployment use `docker-compose up -d` to start the application. Docker Compose will pull required images, create containers, and start the services defined in the compose file. To ensure that all services are running as expected it's possible to use `docker ps` and `docker-compose logs` to check the status and logs of the containers.
Then it's possible to access to Nextcloud with `http://localhost:8080` URL.

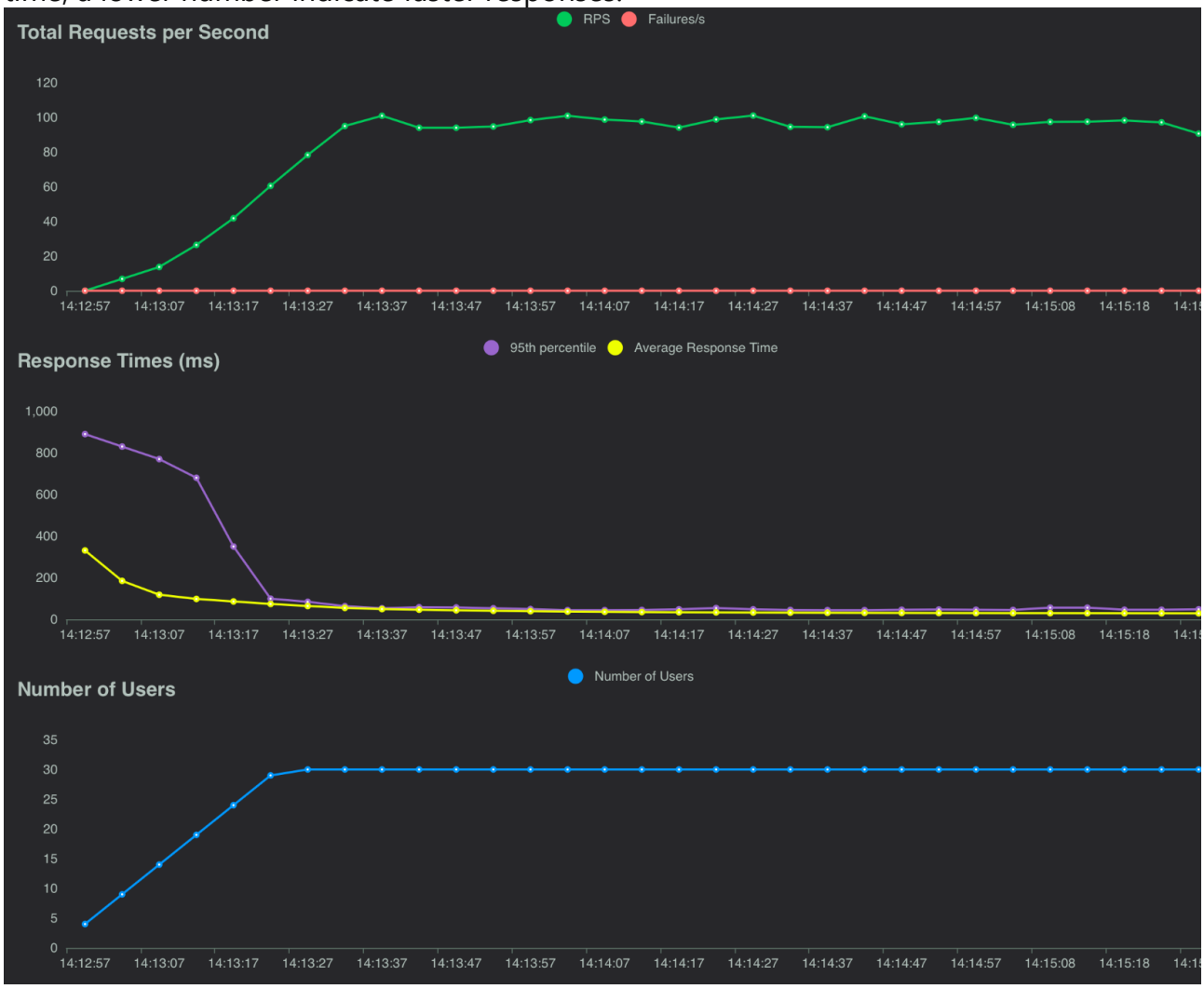
Discuss how you would monitor and manage the deployed system.
To monitor the health and performance of the containers it's possible to use `docker logs` for container logs and `docker stats` for real-time metrics.
Implementing a tool like Portainer offer a more comprehensive management and monitoring solution, in fact Portainer provides a web-based UI for container management, including detailed statistics, log viewing, and container control.
For advanced monitoring, the combination of Prometheus and Grafana can be useful for detailed insights into the application's performance and health: Prometheus for metrics collection and Grafana for metrics visualization.

Choose a cloud provider that could be used to deploy the system in production and justify your choice.
For the system deployment AWS (Amazon Web Services) is a possible solution for multiple reasons: it provides auto-scaling capabilities that can automatically adjust the number of compute instances based on the application's load; it can host applications closer to the end-users, reducing latency and improving user experience; it relies on high standards of security; it has a pay-as-you-go pricing model, which is very useful for optimizing costs.

Test your infrastructure

Consider the performance of your system in terms of load and IO operations
To test the performance of the system in terms of load and IO operations it could be helpful to use Locust, an open-source, scalable, user load testing tool written in Python. It's designed to help developers simulate millions of simultaneous users visiting a website or using an application to test its performance under stress. Unlike traditional load-testing tools that use a graphical user interface and static test scripts, Locust is code-driven, allowing to write test scenarios in Python, offering greater flexibility and realism in simulating user behavior.
For reference, tests were executed on a MacBook Pro M1.
A first test has been executed for 2 minutes with 30 users and a spawn rate equal to 10 for a small file.
In the plot the green line (RPS) represents the number of requests that the server is handling per second, this is a measure of throughput and is important for understanding the load your application can handle. The red line represents failures per second, ideally this line should be at zero, indicating no failed requests. The purple line represents the 95th percentile of response times, this means that 95% of the requests are completed within this

time frame, the decrease of this line suggest improved response times as the test progresses or that the system is handling the load well. The yellow line shows the average response time, a lower number indicate faster responses.



From this plot it's possible to observe that there is no failures, moreover the system can handle many request, this show that the system is responsive and has a robust behavior.
The POST request has a median response time of 750 ms, which is considerably higher than other requests, this might be due to authentication processes that take longer to execute.
The PROPFIND request have a low median response time of 21 ms, which increases slightly on average.
The PUT requests, specifically for uploading a file, have a median time of 45 ms, which is reasonable for file upload operations but with a wide range going up to 1273 ms.
The GET requests for downloading the same file across various users have very consistent and low median response times (17-19 ms), suggesting that the system handles file retrieval operations effectively.

Then a big file (created with `dd if=/dev/zero of=big-file.txt bs=1M count=512`) has been tested for 1 minute with 30 users and a spawn rate equal to 10. The results show that some failures are present (but still close to 0) and the RPS heavily decreased wrt the small file, the reason is that it takes more time to be processed. Moreover, the response time increased a lot, this indicate that the responses are lower.

The response times for both successful and failed requests are exceptionally high. Notably, the POST requests have median and maximum response times of 50,000 ms and 63,000 ms, respectively, which are beyond acceptable limits for user-facing operations.