Cloud Basic Project

Author

Andrea Buscema

Data Science and Artificial Intelligence

University of Trieste

andrea.buscema@studenti.units.it

Tasks of the project

The objective of this project is to identify, deploy and implement a cloud-based file storage system. The system must allow users to upload, download and delete files, with each user having a private storage space. The system must be scalable, secure and cost-efficient. A solution between Nextcloud (https://nextcloud.com/) and MinIO (https://min.io/) was suggested for consideration.

First of all, it's necessary to explain what are Nextcloud and MinIO:

Nextcloud and MinIO are both cloud storage solutions, but they serve in different ways and have different features, which could influence the decision of choosing one versus the other.

Nextcloud

Nextcloud was primarily designed for file sharing and collaboration platform, if offers features like file synchronization, calendar, contacts, mail, and task management toold. It's often (and easily) seen as an alternative to services like Google Drive (https://www.google.com/drive/) or Dropbox (https://www.dropbox.com/), with a strong emphasis on **privacy and self-hosting**. It offers a wide range of features including end-to-end encryption, two-factor authentication, collaboration tools and integration with third-party applications. It's more user-friendly for non-technical users, for example it provides a web interface, desktop clients and mobile apps, making it accessible for a wide range of users. It's also easy to deploy for smaller to medium-sized installations (it can be scaled, but may require additional configuration and hardware for very large deployments). Lastly, has a large community and offers professional support services.

MinIO

MinIO it's an high-performance, distributed object storage system desiged for large-scale private cloud infrastructures. It is API-compatible with Amazon S3, making it suitable for enterprises looking for a private or hybrid cloud storage solution. MinIO focuses ore on the storage aspect rather than collaboration tools. It's specialized and optimized for machine learning, analytics and application data, and offers features like erasure coding and bitrot protection for data integrity. Instead of Nextcloud, MinIO focuses more on API access and is typically managed via command line, though it does have a basic web interface for management tasks (that, from my personal opinion, it can be always appreciate). MinIO was deisgned for large-scale deployment from the start, easily scalable and can handle petabytes of data and high levels of throughput. Also, has a strong community support, with enterprise support available. It's often favored in enterprise environments with large-scale storage needs.

Choice and reason

Regarding project tasks, the first choice fell on Nextcloud. The reasons are:

- file sharing and collaboration with user-friendly interface, including non-technical users;
- strong privacy controls and open-source software;
- small to medium-sized stogare needs and an easy-to-deploy solution.

In contrast, in real world use with big data projects, where primary requirements is high-performance, scalable object storage, MinIO might be the more suitable choice.

Obviously, the decision depends on specific requirements, the scale of deployments and the features most important regards different kinds of projects/works.

Understanding Docker and docker Compose

Docker (https://www.docker.com/) is a platform for developing, shipping and running apploications is isolated environments called containers. Containers package up an application with all its dependencies.

Docker Compose is a tool for defining and running multi-container Docker appications. With Compose, it will used a YAML file to configure applications's services, networks and volumes.

How to create a Docker Compose

To define services, like Nextcloud and its database and their configurations, is needed to create a 'docker-compose.yml' file. Then, is needed to define Nextcloud service and Database service (Nextcloud requires a database to store user data).

Running Docke Compose file

Use the command line to navigate to the directory containing the 'docker-compose.yml', run:

```
docker-compose up
```

This command pulls the necessary Docker images and starts the services defined in your compose file. Once the containers are running, it's possible to access Nextcloud through web browser at, i.e., `http://localhost:8080`.

Monitoring and Managing

To monitor logs and containers, it's possible use

`docker logs` and `docker ps`

Also, it's possible to use Docker Desktop to start, stop, and restart services. It's possible to keep an eye on resource usage (CPU, memory) with Docker's built-in tools (or, it's possible also with external tools like Portainer).

Documentation, Code and Presentation

In repository, will be available:

- Documentation where is explained and described the platform's architecture, including components, databases and their interactions, with a section on the security measures taken.
- Docker files and any code developed/modified for the cloud-based file storage system, with a README file with instructions on how to deploy and use the system developed.
- A short presentation summarizing the design and implementation.

Deployment Plan using Docker and Docker-Compose

After choosing between Nextcloud and MinIO, it was planned the deployment on laptop in a containerized environment using Docker and Docker-Compose.

Install Docker and Docker-Compose

First, it is necessary to install Docker and Docker-Compose. On MacOS, it is possible to install Docker Desktop, which includes Docker-Compose. (On Linux, it is needed to install Docker and Docker-Compose separately.)

To install Docker Desktop, it is necessary to download the installer from the official website: https://www.docker.com/products/docker-desktop

After downloading the installer, it is necessary to run it and follow the instructions. After the installation, it is possible to check if Docker and Docker-Compose are installed using the following commands:

```
docker --version
docker-compose --version
```

If Docker and Docker-Compose are installed, the commands will return the version of the installed software.

Otherwise, it is possible to install Docker and Docker-Compose using Homebrew:

```
brew install docker docker-compose
```

Create Docker-Compose File

After installing Docker and Docker-Compose, it is necessary to create a Docker-Compose file to define the services that will be used in the deployment. The Docker-Compose file is a YAML file that defines the services, networks, and volumes that will be used in the deployment.

The Docker-Compose file for the deployment of Nextcloud is as follows:

```
version: '3'

services:
  db:
    image: mariadb
    command: --transaction-isolation=READ-COMMITTED --binlog-format=ROW
    restart: always
    volumes:
```

```
      - db:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_PASSWORD: example
      MYSQL_DATABASE: example
      MYSQL_USER: example

  app:
    image: nextcloud
    ports:
      - 8080:80
    links:
      - db
    volumes:
      - nextcloud:/var/www/html
    restart: always
    environment:
      MYSQL_HOST: db
      MYSQL_DATABASE: example
      MYSQL_USER: example
      MYSQL_PASSWORD: example
```

Here is where it's possible to include choosen file storage system, database service (if required), and any other necessary service.

For what concerns database, it was used MariaDB, which is a community-developed fork of MySQL. It is included in the Docker-Compose file as a service, and it is linked to the Nextcloud service. The MariaDB service is defined with the image mariadb, and it is configured with the environment variables MYSQL_ROOT_PASSWORD, MYSQL_PASSWORD, MYSQL_DATABASE, and MYSQL_USER. The volumes section is used to define the volume that will be used to store the database data. The restart section is used to define the restart policy for the service. The command section is used to define the command that will be used to start the service. The links section is used to define the link between the MariaDB service and the Nextcloud service.

For what concerns Nextcloud, it is included in the Docker-Compose file as a service, and it is linked to the MariaDB service. The Nextcloud service is defined with the image nextcloud, and it is configured with the environment variables MYSQL_HOST, MYSQL_DATABASE, MYSQL_USER, and MYSQL_PASSWORD. The volumes section is used to define the volume that will be used to store the Nextcloud data. The restart section is used to define the restart policy for the service. The ports section is used to define the port that will be used to access the Nextcloud service. The links section is used to define the link between the Nextcloud service and the MariaDB service.

Deploy Services using Docker-Compose

After creating the Docker-Compose file, it is possible to deploy the services using Docker-Compose. To deploy the services, it is necessary to run the following command in the directory where the Docker-Compose file is located:

```
docker-compose up -d
```

The command will start the services in the background, and it will create the necessary networks and volumes. After the services are started, it is possible to check the status of the services using the following command:

```
docker logs
docker ps
```

The command will return the status of the services, and it will show if the services are running or not. If the services are running, it is possible to access the Nextcloud service using a web browser. The Nextcloud service can be accessed at the following URL:

```
http://localhost:8080
```

After accessing the Nextcloud service, it is possible to create an account and start using the service. The Nextcloud service can be used to store files, share files, and collaborate with others. It is also possible to install apps and plugins to extend the functionality of the service.

What I used is Docker Desktop for MacOS. It's user friendly and easy to use. It's possible to manage containers, images, networks, and volumes using the Docker Desktop interface. It's also possible to view the logs of the services, and to stop and remove the services using the Docker Desktop interface.

Explanation of the Deployment Plan with Docker and Docker-Compose

Deploying Nextcloud with Docker Compose involves some steps, each focusing on creating an isolated yet interconnected environment where Nextcloud can operate with its required services.

Structure of Docker Compose File

The file is structured in a specific format that docker Compose understands. It begins with a version declaration (version: '3.8' for example) that specifies the version of the Compose file format. It then contains definitions for different sections, such as services, networks, and volumes.

Defining Services

- **Database Service**: A database is essential for Nextcloud to store data. It is possible to define a service (often named `db` for simplicity) using a database image MySQL or MariaDB. It is possible also to set environment variables (like root password, database name, and user credentials) required for the database configuration.
- **Nextcloud Service**: This is the main application service that runs the Nextcloud image. The `image` tag is used to specify the Nextcloud image. The `ports` section maps the ports of the container to your host machine, allowing to access Nextcloud via a web browser. The `links` tag is used to link this service to the database service, allowing Nextcloud to access the database. Environment variables can be set to configure the connection to the database (like database name, user, and password). The `volumes` tag is used for data persistance. Without this, all data stored in Nextcloud would be lost when the container is stopped.

Accessing Nextcloud

After running the `docker-compose up -d` command, Nextcloud can be accessed at `http://localhost:8080` in a web browser. The initial setup involves creating an admin account and configuring the database settings. Once the setup is complete, Nextcloud is ready to use.

In Docker Desktop, just easily click on "Containers" and click on play button to start the container.

To stop the services, it is possible to click on the stop button in the Docker Desktop interface, or to run the following command in the directory where the Docker-Compose file is located:

```
docker-compose down
```

The command will stop the services and remove the networks and volumes that were created. After the services are stopped, it is possible to remove the containers and images using the Docker Desktop interface. It's also possible to remove the containers and images using the following commands:

```
docker container rm <container_id>
docker image rm <image_id>
```

The commands will remove the containers and images that were created. After the containers and images are removed, it is possible to remove the networks and volumes using the Docker Desktop interface. It's also possible to remove the networks and volumes using the following commands:

```
docker network rm <network_id>
docker volume rm <volume_id>
```

The commands will remove the networks and volumes that were created.

Tests and Checks

After the initial setup, it is possible to test the functionality of Nextcloud by uploading files, creating folders, and sharing files with others. It is also possible to install apps and plugins to extend the functionality of the service. Doing this can help to ensure that Nextcloud is working as expected and that it is ready to use. It also needed to check and make sure that Nextcloud is correclty connected to the database and that the data is being stored correctly.

Management and Maintenance

This part is crucial for ensuring the long-term stability and security of the setup.

General Management tasks include: - Regularly updating for both Nextcloud and the database; - Backup strategy, which includes regular backups of the database and Nextcloud data; - Regularly review user accounts and permissions to ensure that access levels are appropriate; - Monitoring disk usage and performance to ensure that the system is running smoothly.

General Maintenance tasks include: - Perform regular health checks, it can be easily done using built in tools under the admin settings to check the health and performance of the system; - Regularly check logs for any errors or warnings; - Security Checks, it is important to regularly review and update security settings to ensure that the system is secure. If the system is exposed to the internet, it is important to consider additional security measures such implementing SSL/TLS certificates and using a firewall. - Regularly update passwords and employ strong authentication methods. - Performance tuning, such as adjusting php settings, fine-tuning database parameters, or scaling your Docker resources.

Scalability Aspects

What was done as far as deployment plan is just a starting point. As the number of users and the amount of data grows, it may be necessary to scale the deployment to handle the increased load. First of all, it is needed to analyse that there are two main approaches to scaling: horizontal and vertical scaling.

Horizontal vs. Vertical Scaling

- **Vertical Scaling**: This involves increasing the capacity and the power of the server by adding more resources (CPU, RAM, etc.) to the existing server. This is often done by upgrading the hardware of the server. This approach has its limitations, like physical and cost limits.
- **Horizontal Scaling**: This involves adding more servers to the existing infrastructure. This approach is more flexible and scalable, as it allows to distribute the load across multiple servers. This approach is often used in cloud environments, where it is possible to add or remove servers as needed.

The system should be capable of scaling out in horizontal scaling, which means adding more instances of the same service (like Nextcloud service) to handle increased load. It is possible to specify the number of replicas in the Docker-Compose file, but for the true horizontal scaling, it is typically done using an more robust orchestration tool like Kubernetes or Docker Swarm in a production environment.

Load Balancing

When scaling out, it is important to consider how the traffic will be distributed across the different instances of the service. Load balancing is a technique used to distribute the incoming network traffic across a group of backend servers. This ensure that no single instance becomes a bottleneck.

Database Scaling

Since it is used MariaDB, it is possible to scale the database using techniques like replication or clustering to handle more significant amounts of data and requests.

Scalability Enhancements

1. **Service Replication**: Firstly, it was prepared the system for potential scaling. While Docker Compose itself isn't used for scaling in production, it was setting up to allows for an easier transition to tools like Docker Swarm or Kubernetes in the future.
2. **Database Scaling**: While actual database scaling won't be configured in Docker Compose, it is important to design with this in mind.

So, this is the updated version of docker-compose.yml incorporating these considerations:

```yaml
version: '3.8'

services:
  nextcloud:
    image: nextcloud
    ports:
      - "8080:80"
    volumes:
      - nextcloud:/var/www/html
    environment:
      - MYSQL_HOST=db
      - MYSQL_DATABASE=nextcloud
      - MYSQL_USER=nextcloud
      - MYSQL_PASSWORD=andf12
    depends_on:
```

```
      - db
    deploy:
      replicas: 1  # Placeholder for scalability, it can be adjusted as needed

  db:
    image: mariadb
    environment:
      - MYSQL_ROOT_PASSWORD=andf12
      - MYSQL_PASSWORD=andf12
      - MYSQL_DATABASE=nextcloud
      - MYSQL_USER=nextcloud
    volumes:
      - db:/var/lib/mysql
    deploy:
      replicas: 1  # Placeholder for scalability, also it can be adjusted as needed

volumes:
  nextcloud:
  db:
```

Notes:

The `replicas` key under `deploy` are a placeholder for scalability and they are used to specify the number of replicas for the service, but serves here as a reminder that for future scaling considerations.

Also, it is possible to consider Load Balancing and Caching Mechanisms to improve the performance and scalability of the system:

In this new version, it was included:

- **Load Balancing**: it was introduce a load balancer service (Nginx) to distribute traffic accross multiple instances of Nextcloud. This will help in handling increased load.
- **Caching Mechanisms**: it was introduce a caching service (Redis) to cache frequently accessed data and reduce the load on the database.

Security Enhancements

Security is a critical aspect of any deployment, especially when it comes to handling sensitive data. It was analysed to address some security enhancements that can be made to the deployment plan, such as secure file storage and transmission, user authentication, and unauthorized access prevention.

Implementing Secure File Storage and Tasmission

It is important to ensure that files are stored and transmitted securely, especially when dealing with sensitive data. This can be achieved by configuring Nextcloud to use HTTPS for secure communication, and can be done by obtaining and installing an SSL/TLS certificate for NGINX (It is possible to use Let's Encrypt for a free certificate).

Securing User Authentication

Nextcloud uses a username and password for user athentication, but by default, it is possible to enhance the security of admin/users authentication using settings, that are available in the Nextcloud admin settings.

Also, it is possible to enhance the security of user authentication by integrating OAuth2/OpenID Connect for single sign-on (SSO) and multi-factor authentication (MFA) for an extra layer of security.

Preventing Unauthorized Access

It is important to prevent unauthorized access to the Nextcloud service. This can be achieved by implementing a firewall to restrict access to the Nextcloud service, and by regularly reviewing and updating security settings to ensure that the system is secure. It is also possible to use a Web Application Firewall (WAF) to protect against common web application attacks.

Obtain an SSL Certificate

Using Let's Encrypt and Certbot (for production environments)

As mentioned, it is possible to use Let's Encrypt to obtain a free SSL certificate for the Nextcloud service. This can be done by installing Certbot, which is a tool for obtaining and renewing SSL/TLS certificates. Once Certbot is installed, is needed to run it to obtain a certificate for the domain. Certbot will automatically provide with two key files: the certificate file and a private key file.

Using a Self-Signed Certificate (for testing purpose)

It is possible to create a self-signed SSL certificate using OpenSSL, that can be done by running the following command:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/nginx-selfsigned.key -out /
```

This will create a self-signed SSL certificate and a private key file. The certificate will be valid for 365 days. The certificate file and the private key file can be used to configure NGINX to use HTTPS.

Update NGINX Configuration

Once the SSL certificate is obtained, it is possible to update the NGINX configuration file to use HTTPS, just adding an HTTPS server block in the configuration file, specify the paths for the SSL certificate and private key files, and redirect HTTP traffic to HTTPS (that is optional, but recommended, as community says).

```
server {
    listen 80;
    server_name yourdomain.com;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name yourdomain.com;

    ssl_certificate /etc/nginx/ssl/cert.pem;
    ssl_certificate_key /etc/nginx/ssl/key.pem;

    # Other NGINX configuration...
}
```

Adjust Docker-Compose File

Then, after creating a directory (for example, `./nginx/ssl`) and placed the SSL certificate and key files, it is needed to adjust the Docker-Compose file to mount the SSL certificate and private key files into the NGINX container:

```
# ...
services:
  load_balancer:
    image: nginx
    ports:
      - 80:80
      - 443:443
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl  # Mounting SSL directory
    # Rest of the configuration...
    #...
```

After this step, it is possible to redeploy Docker environment using `docker-compose up -d`.

Ensuring Data Encryption

Nextcloud provides server-side encryption to protect data. To enable server-side encryption, it is possible to go to the Nextcloud admin settings, in "Security" section, and enable the "Server-side encryption" option. This will encrypt the data stored in Nextcloud, and it will provide an extra layer of security.

After enabling encryption, it is important to ensure that the encryption keys are backed up and stored securely. Nextcloud will asks to log out and log back in to generate new encryption keys.

For these files that are already stored in Nextcloud before enabling encryption, it is needed to run a commando to encrypt them: `php occ encryption:encrypt-all`.

This enhancement, along with SSL/TLS for data in transit, greatly improves the overall security of the cloud system, and this is especially important when dealing with sensitive data, such as personal or financial information.

Other Security Enhancements

It is also possible to consider other security enhancements, such integrating OAuth2/OpenID Connect and multi-factor authentication (MFA) for an extra layer of security.

OAuth2/OpenID Connect: This can be used to enable single sign-on (SSO) and to allow users to log in using their existing credentials from another service, such as Google or GitHub. This can simplify the login process for users and improve the overall security of the system. It can be done by installing and configuring an OAuth2/OpenID Connect provider, such as Keycloak or Okta, and integrating it with Nextcloud.

Multi-Factor Authentication (MFA): This can be used to add an extra layer of security to the login process by requiring users to provide a second form of authentication, such as a code sent to their phone or a fingerprint scan. This can be done by installing and configuring an MFA provider, such as Google Authenticator or Duo, and integrating it with Nextcloud.

Also, it is possible to consider using a Web Application Firewall (WAF) to protect against common web application attacks, such as SQL injection and cross-site scripting. A WAF can help to prevent unauthorized access to the Nextcloud service and to protect the system from common web application attacks.

As said before, it is important to regularly review and update Nextcloud, MariaDB, and the Docker images. Regularly updating the software will help to ensure that the system is secure and that it is protected against known vulnerabilities. It is also important to regularly review user accounts and permissions to ensure that access levels are appropriate, and to monitor disk usage and performance to ensure that the system is running smoothly.

Cost Efficiency

Cost efficiency is an important consideration when deploying a cloud system, especially when it comes to choosing the right infrastructure and services. It is important to consider the cost of the infrastructure, the cost of the services, and the cost of the maintenance and management of the system.

Actually, with these implementation, the cost of the infrastructure in relative low, as it is using Docker and Docker-Compose, which are open-source and free to use. The cost of the services will depend on the specific services that are used. The cost of the maintenance and management of the system will depend on the complexity of the system and the level of expertise required to manage it.

Cost Implication of the deployed system

For the project, it was considered the optimization of the container resources by setting resource limits, also to prevents over-allocations and under-utilization of resources. Also, it was considered the managing of docker volumes and images, to prevent the accumulation of unused data and to reduce the storage costs. Other project-related costs are network costs, especially in the cloud environment. Beyond the specific use of the project, the optimisation of data transfer processes is important in order to minimise data input and output costs.

Cost Efficiency Enhancements

It is possible to consider some cost efficiency enhancements, such as:

- Optimization od Docker Images: it can be done using smaller base images to reduce the size and resource requirements of the containers, or regularly cleaning up unused images, containers, and volumes to freep up space and resources.
- Scalability and Auto-Scaling: It is possible to implement auto-scaling where possible, to automatically adjust the number of instances based on the current load (so scaling up during high demand and down during low demand). This can help to reduce costs by only using the resources that are needed.
- Monitoring and Cost Analysis: It is important to regularly monitor the system and to analyze the costs to identify areas where costs can be reduced. This can be done implementing monitoring tools to track resource usage and costs.
- Choosing the Right Services: It is important to choose the right services that meet the requirements of the system while minimizing costs. This can be done by comparing the costs of different services and choosing the ones that offer the best value for the money.

Implementing these cost efficiency enhancements can help ensure that the cloud systema remains economically sustainable, especially as it scales up to handle increased load and data.

Returning to the Deployment Plan

The deployment plan was designed to deploy Nextcloud in a containerized environment using Docker and Docker-Compose. The plan included the installation of Docker and Docker-Compose, the creation of a

Docker-Compose file to define the services, and the deployment of the services using Docker-Compose. The plan also included the initial setup of Nextcloud, the management and maintenance of the system, and the scalability aspects, it was considered security enhancements and cost efficiency.

So, the deployment plan was designed to be flexible and scalable, and it was considered the possibility of scaling the deployment to handle increased load and data. It was also considered the security enhancements to ensure the confidentiality and integrity of the data, and the cost efficiency enhancements to ensure that the system remains economically sustainable.

The deployment plan was designed to be a **starting point**, and it was considered the possibility of using more robust orchestration tools like Kubernetes or Docker Swarm in the future. It was also considered the possibility of integrating OAuth2/OpenID Connect and multi-factor authentication (MFA) for an extra layer of security, and the possibility of using a Web Application Firewall (WAF) to protect against common web application attacks.

Actually, from the Docker Desktop setup, it was set up Resources and Advanced settings, and it was set up the Docker Desktop to use 4 CPUs and 8 GB of memory, with Swap of 1 GB (Swap means that if the system needs more memory resources and the RAM is full, inactive pages in memory are moved to the swap space), Virtual disk limit of 64 GB (Due to filesystem overhead, the real available space might be less).

Monitoring Tools

By default, the Docker Desktop includes a built-in monitoring tool that provides information about the containers, images, networks, and volumes. It is also possible to use third-party monitoring tools, such as Prometheus and Grafana, to monitor the system and to analyze the performance and resource usage.

Prometheus and Grafana

Prometheus is an open-source monitoring and alerting toolkit that is designed for reliability and scalability. It is possible to use Prometheus to collect metrics from the Docker containers and to store them in a time-series database.

Grafana is an open-source platform for monitoring and observability that is designed to visualize and analyze the metrics collected by Prometheus. It is possible to use Grafana to create dashboards and alerts to monitor the performance and resource usage of the system.

For this project, it was added both Prometheus and Grafana to a Docker implementation.

To do that, it is needed to create a Docker Network that will allow the containers to communicate with each other:

```
docker network create monitoring
```

Then, it was created a file named `prometheus.yml` to configure it to scrape metrics:

```yaml
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

Run Prometheus in Docker:

```
docker run -d --name=prometheus -p 9090:9090 --network=monitoring -v ./prometheus.yml:/etc/prometheus/p
```

This command runs Prometheus in a detached mode, names the container `prometheus`, exposes it on port 9090 and mounts the configuration file into the container.

Then, start Grafana:

```
docker run -d --name=grafana -p 3000:3000 --network=monitoring grafana/grafana
```

This command starts Grafana, names the container `grafana`, and exposes it on port 3000.

After this, it is possible to access Grafana at `http://localhost:3000` in a web browser and log in using the default credentials (admin/admin). Then, it is possible to add Prometheus as a data source and create dashboards to visualize and analyze the metrics collected by Prometheus.

This is a basic setup, both Prometheus and Grafana offer a range of configurations and options to suit different requirements. Obviously, it is needed to ensure to secure Grafana and Prometheus instances, especially if they will be exposed to the internet.

In this project, it was integrated Prometheus and Grafana into docker-compose.yml file:

```yaml
services:
  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"
    networks:
      - your_network

  grafana:
    image: grafana/grafana
    volumes:
      - grafana_data:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=your_password
    ports:
      - "3000:3000"
    networks:
      - your_network

volumes:
  grafana_data:

networks:
  your_network:
    driver: bridge
```

This is a basic setup, and it is possible to add more advanced configurations and options to suit different requirements.

Nextcloud doesn't provide a Prometheus-compatible metrixs endpoint, it is needed to use an additional plugin or exporter to collect metrics from Nextcloud. So, it was considered to use an exporter that can translate Nextcloud's status into Prometheus-readable metrics, simply by adding a new service to the docker-compose.yml file:

```
services: # ...
# ...
node_exporter:
    image: prom/node-exporter
    ports:
      - "9100:9100"
# ...
```

This will expose the node exporter on port 9100, and it will collect metrics from the host system and make them available to Prometheus. Then, was also updated prometheus.yml to include the new target:

```
global:
# ...
scarpe_configs:
#...
    - job_name: 'node'
    static_configs:
      - targets: ['node_exporter:9100']
```

From here, it is possible to access Prometheus at `http://localhost:9090` and Grafana at `http://localhost:3000` in a web browser. In Grafana it is possible to add Prometheus as a data source and create dashboards to visualize and analyse the metrics collected by Prometheus, from the panel configuration screen, using a query editor using PromQL (Prometheus Query Language) where is possible, for example, using `node_cpu_seconds_total` to see CPU usage.

Grafana, as said before, offers various type of visualization like graphs, table, heatmaps, and more, where they are customizable. Once the dashboard is created and saved, it is possible to share it with others, and it is possible to set up alerts to be notified when certain conditions are met.

Grafana has extensive documentation and a large community, there are also pre-built dashboards, experiment with different visualizations.

In this project, it was firstly created 3 dashboards:

- **CPU Usage**: `node_cpu_seconds_total{mode="system"}`;
- **Memory Usage**: `node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes`;
- **Disk IO**: `rate(node_disk_io_time_seconds_total[2m])`.

Obviously, is possible to adjust time ranges and other settings to suit specific requirements. After writing the query, Grafana will automatically execute it and display the results.

Stress Test - Testing the infrastructure

For what concerns testing, it is possible to use a variety of tools and techniques to test the infrastructure. The test that was conducted was **load testing** and **performance testing**.

The choice of tool often depends on the specific requirements of the test scenario (e.g. necessary protocols, level of realism, reporting capability), the budget (if available) and, of course, the skills of the owner/team.

Tools for testing

To do this test, there are several tools that can be used to conduct load testing:

- **Apache JMeter**: Is a open-source tool designed for load testing and measuring performance. It's widely used for its versatility and can simulate heavy loads on servers, networks, or objects to test strength or analyse overall performance under different load types (https://jmeter.apache.org/).
- **LoadRunner (Micro Focus)**: Another widely used tool for performance testing, LoadRunner can simulate thousands of users concurrently, makit it ideal for complex applications. It supports a wide range of application environments, protocols and APIs (https://www.microfocus.com/en-us/products/loadrunner-professional/overview). There's a free trial available, but it's a paid tool.
- **BlazeMeter (Acquired by Broadcom)**: A cloud-based load testing service compatible with Apache JMeter. It provides an easy-to-use interface for running JMeter tests at scale, and offers real-time reporting and analytics (https://www.blazemeter.com/). It's a paid tool, but there's a free trial available.
- **Gatling**: An open-source load testing tool designed for an easy use. It's written in Scala and can be used to simulate heavy loads on web applications (https://gatling.io/).
- **Artillery.io**: A modern, powerful, and easy-to-use load testing toolkit. It's used to test backend systems such as HTTP APIs, WebSocket services and more (https://artillery.io/).
- **K6**: An open-source load testing tool and SaaS for engineering teams. It's used to be developer-friendly with a strong on automation. It's also used to test APIs, microservices and websites (https://k6.io/).
- **Locust**: An open-source load testing tool that allows to define user behavior with Python code, making it highly flexible and programmable. It is designed to create distributed and scalable load tests for web applications (https://locust.io/). One of the main advantages of Locust is that it's easy to use and can be used to create complex test scenarios.

Testing

Conducting testing on the system is essential to understand how it behaves under various levels of stress, particularly in terms of user load and I/O operations.

For this project, it was used **Locust** to conduct testing. The main reason for choosing Locust is that, first of all, it's open-source and free to use, and it's also easy to use and can be used to create complex test scenarios. Also the script can be written in Python, which is a language that is familiar to me.

Another appreciated feature of Locust is that provides a real-time web interface for starting tests, monitoring their progress, and viewving results. This interactive approach, from a personal point of view, is more intuitive and user-friendly than other tools. Then, during the research of the tools, it was found that Locust it's lightweight and doesn't demand extensive resources, making it efficient for various environments, including local machines for development or testing. So, at the end, it seems be the best choice for this project (also, it's seems to be the most appreciated tool in the community).

First, it is needed to have Locust installed. It can be done using pip:

```
pip install locust
```

Define Testing Parameters

Before starting the test, it is needed to define the testing parameters for load testing. The objective is to understand how the system behaves under different levels of stress in terms of user load and I/O operations.

Is possible to do several tests:

- **Concurrent Users**: Test the maximum number of users that the system can handle concurrently. This can be done by gradually increasing the number of concurrent users, starting with 50 and then increase to 100 and 200, and monitoring the system's response time and error rate;

- **Hatch Rate**: Test the system's ability to handle a sudden increase in users. This can be done by gradually increasing the hatch rate, starting with 10 users per second and then increase to 20 and 30, and monitoring the system's response time and error rate;
- **Types of Requests**: Test the system's ability to handle different types of requests, such as file uploads, file downloads, and file sharing. This can be done by simulating different types of requests and monitoring the system's response time and error rate, such as the time it takes to upload a file, the time it takes to download a file, and the time it takes to share a file.

Example of a Locust Test Script

To use Locust, it is needed to create a test script that defines the user behavior and the tasks that the users will perform. The test script is written in Python. Here there is an example of a simple test script that simulates a user uploading a text file to Nextcloud:

```
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
    wait_time = between(5, 9)

    @task
    def upload_file(self):
        self.client.post("/upload", files={"file": open("test.txt", "rb")})
```

This script defines a user that will upload a file to Nextcloud. The `wait_time` attribute is used to define the wait time between the tasks, and the `@task` decorator is used to define the task that the user will perform. The `self.client.post` method is used to send a POST request to the `/upload` endpoint with a file named `test.txt`.

After configuring credentials (if needed), that can be done by sending a POST request to the login endpoint with the user's credentials, and then using the obtained session for subsequent requests, it is possible to run the test using the following command:

```
locust -f path/to/test_script.py
```

This will start the Locust web interface at `http://localhost:8089`, where it is possible to start the test, monitor the progress, and view the results.

Example of Script for Load Testing Nextcloud

For the load testing of this project, it was created a test script that simulates a user uploading a file to Nextcloud, and then downloading the file. The script is written in Python and uses the Locust library to define the user behavior and the tasks that the users will perform.

This is an example of a test script that simulates a user uploading a file to Nextcloud and then downloading the file:

```
from locust import HttpUser, task, between

class NextcloudUser(HttpUser):
    wait_time = between(5, 9)

    def on_start(self):
```

```
        self.login()

    def login(self):
        response = self.client.post("/login", data={"username": "user", "password": "password"})
        if response.status_code != 200:
            print("Login failed")

    @task
    def upload_file(self):
        self.client.post("/upload", files={"file": open("test.txt", "rb")})

    @task
    def download_file(self):
        self.client.get("/download/test.txt")
```

Testing the Infrastructure

For this project, it was created a bash file and a python file:

```
#!/bin/bash

# This script is used to test the nextcloud-docker project.

# define some variables
URL="http://localhost:8080/ocs/v1.php/cloud/users"
USERNAME="ab_andrea"
PASSWORD="H#8@$rkBvFVj9t"

# Function to create users
create_users() {
  for i in {1..5}
  do
    echo "Creating user user$i"
    docker exec -u www-data -e OC_PASS="password$i" nextcloud-docker-nextcloud-1 php occ user:add --pass
  done
}

# Function to delete users
delete_users() {
  for i in {1..5}
  do
    echo "Deleting user: user$i"
    docker exec -u www-data nextcloud-docker-nextcloud-1 php occ user:delete "user$i"
  done
}

# create users
create_users

#Run the locust test
locust -f test_two.py --host http://localhost:8080
```

```
# cleanup
echo "Cleaning up"
delete_users

echo "Test and cleanup complete."
```

```
import os
from locust import HttpUser, task, between
import random
import logging

logging.basicConfig(level=logging.INFO)

class NextcloudUser(HttpUser):
    wait_time = between(1, 5)

    def on_start(self):
        # Initialize user credentials
        user_index = random.randint(1, 5)  # Randomly pick a user between user1 and user5
        self.username = f"user{user_index}"
        self.password = f"password{user_index}"
        self.auth = (self.username, self.password)
        self.login()

    def login(self):
        # Perform login for each user
        self.client.post("/login", {
            "user": self.username,
            "password": self.password
        })

    @task(1)
    def upload_file(self):
        # Simulate file upload
        with open( "test.txt", "rb") as file:
            response = self.client.put(f"/remote.php/dav/files/{self.username}/test.txt",
                            data=file,
                            auth=self.auth)
            logging.info(f"Upload Response: {response.status_code} - {response.text}")

    @task(1)
    def download_file(self):
        # Simulate file download
        response = self.client.get(f"/remote.php/dav/files/{self.username}/test.txt",
                        auth=self.auth)
        logging.info(f"Download Response: {response.status_code} - {response.text}")


# this test will simulate login, upload and download of a file
```

After running the test (`./test.sh`), it was possible to access the Locust web interface at `http://localhost:8089` in a web browser, and start the test, monitor the progress, and view the results.

From Logs, it was possible to see the response of the requests, and it was possible to see the response time

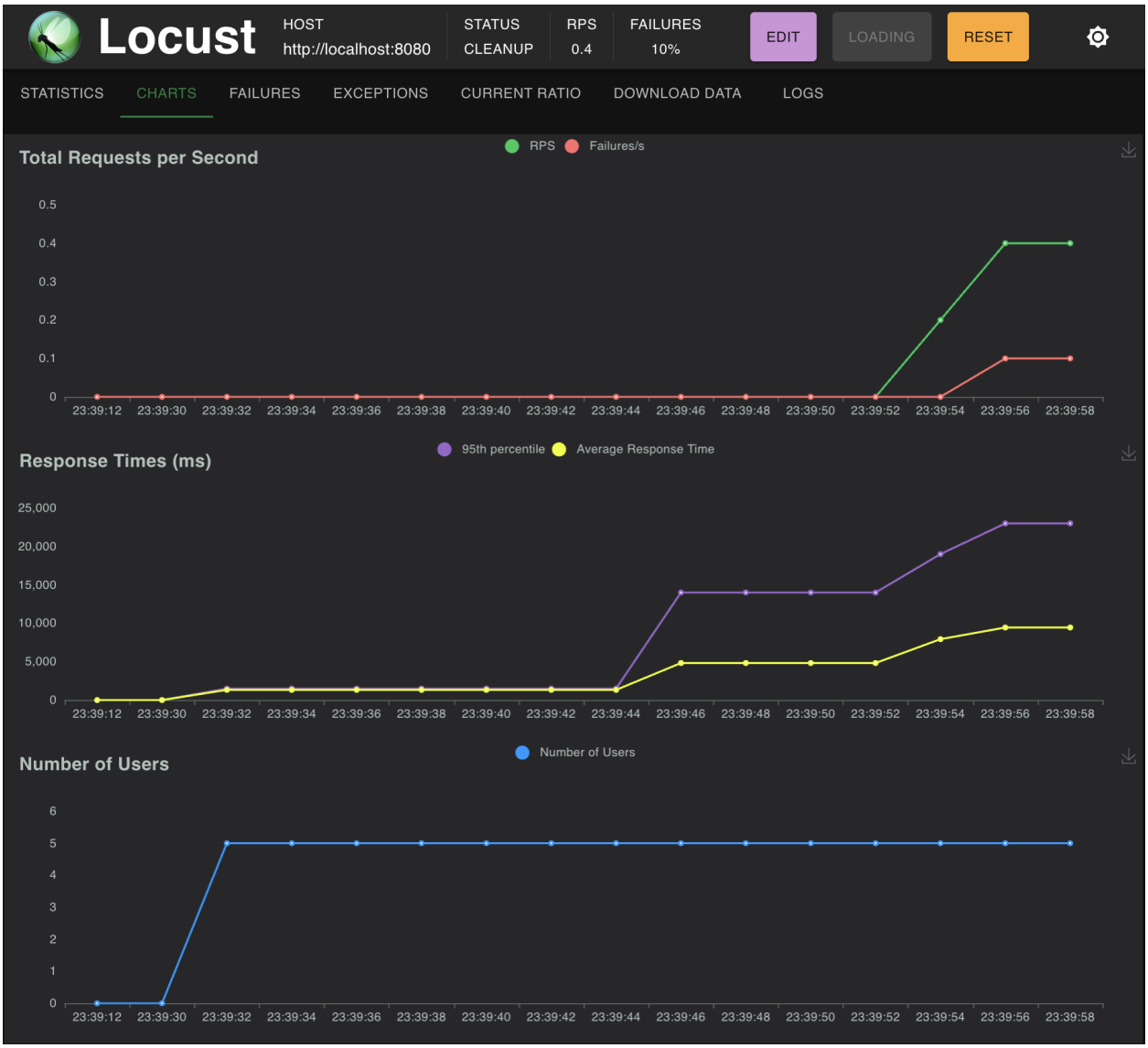
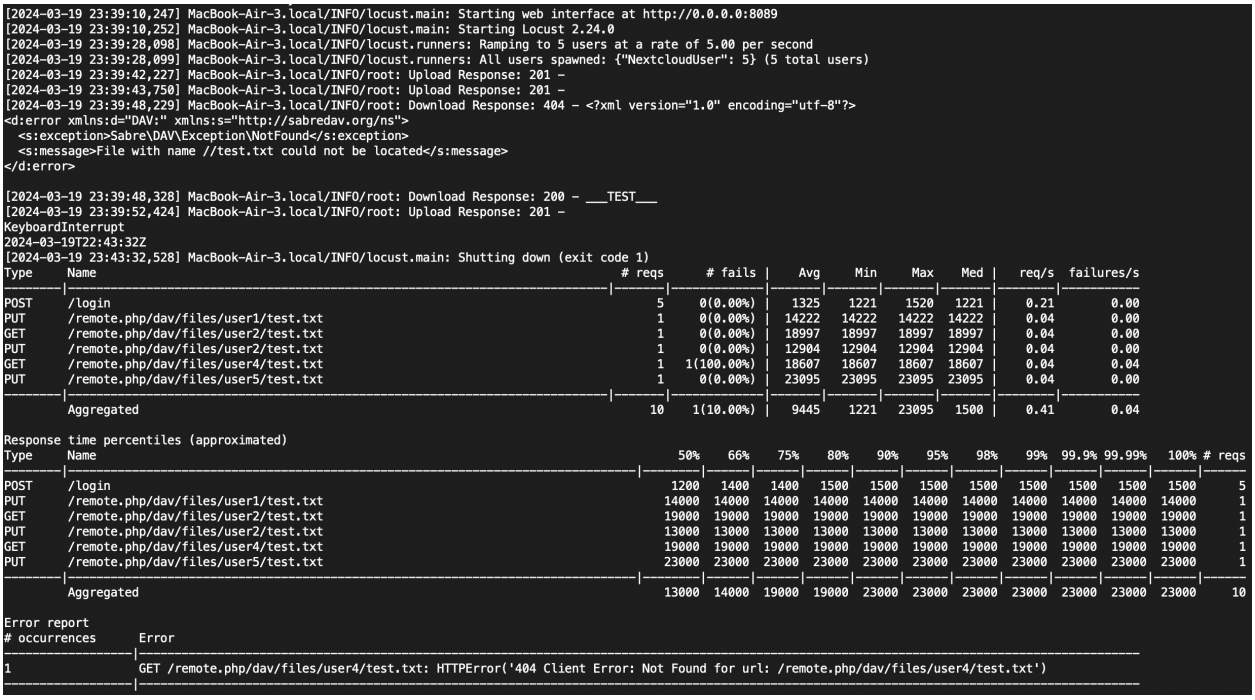


Figure 1: test_1

and the error rate.



```
[2024-03-19 23:39:10,247] MacBook-Air-3.local/INFO/locust.main: Starting web interface at http://0.0.0.0:8089
[2024-03-19 23:39:10,252] MacBook-Air-3.local/INFO/locust.main: Starting Locust 2.24.0
[2024-03-19 23:39:28,098] MacBook-Air-3.local/INFO/locust.runners: Ramping to 5 users at a rate of 5.00 per second
[2024-03-19 23:39:28,099] MacBook-Air-3.local/INFO/locust.runners: All users spawned: {"NextcloudUser": 5} (5 total users)
[2024-03-19 23:39:42,227] MacBook-Air-3.local/INFO/root: Upload Response: 201 -
[2024-03-19 23:39:43,750] MacBook-Air-3.local/INFO/root: Upload Response: 201 -
[2024-03-19 23:39:48,229] MacBook-Air-3.local/INFO/root: Download Response: 404 - <?xml version="1.0" encoding="utf-8"?>
<d:error xmlns:d="DAV:" xmlns:s="http://sabredav.org/ns">
  <s:exception>Sabre\DAV\Exception\NotFound</s:exception>
  <s:message>File with name //test.txt could not be located</s:message>
</d:error>

[2024-03-19 23:39:48,328] MacBook-Air-3.local/INFO/root: Download Response: 200 - ___TEST___
[2024-03-19 23:39:52,424] MacBook-Air-3.local/INFO/root: Upload Response: 201 -
KeyboardInterrupt
2024-03-19T22:43:32Z
[2024-03-19 23:43:32,528] MacBook-Air-3.local/INFO/locust.main: Shutting down (exit code 1)
Type     Name                                                                          # reqs      # fails |    Avg     Min     Max    Med |   req/s  failures/s
--------|-----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
POST     /login                                                                             5     0(0.00%) |   1325    1221    1520   1221 |    0.21        0.00
PUT      /remote.php/dav/files/user1/test.txt                                               1     0(0.00%) |  14222   14222   14222  14222 |    0.04        0.00
GET      /remote.php/dav/files/user2/test.txt                                               1     0(0.00%) |  18997   18997   18997  18997 |    0.04        0.00
PUT      /remote.php/dav/files/user2/test.txt                                               1     0(0.00%) |  12904   12904   12904  12904 |    0.04        0.00
GET      /remote.php/dav/files/user4/test.txt                                               1   1(100.00%) |  18607   18607   18607  18607 |    0.04        0.04
PUT      /remote.php/dav/files/user5/test.txt                                               1     0(0.00%) |  23095   23095   23095  23095 |    0.04        0.00
--------|-----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
         Aggregated                                                                        10    1(10.00%) |   9445    1221   23095   1500 |    0.41        0.04

Response time percentiles (approximated)
Type     Name                                                                                50%    66%    75%    80%    90%    95%    98%    99%  99.9% 99.99%   100% # reqs
--------|-----------------------------------------------------------------------------|--------|------|------|------|------|------|------|------|------|------|------|------
POST     /login                                                                             1200   1400   1400   1500   1500   1500   1500   1500   1500   1500   1500      5
PUT      /remote.php/dav/files/user1/test.txt                                              14000  14000  14000  14000  14000  14000  14000  14000  14000  14000  14000      1
GET      /remote.php/dav/files/user2/test.txt                                              19000  19000  19000  19000  19000  19000  19000  19000  19000  19000  19000      1
PUT      /remote.php/dav/files/user2/test.txt                                              13000  13000  13000  13000  13000  13000  13000  13000  13000  13000  13000      1
GET      /remote.php/dav/files/user4/test.txt                                              19000  19000  19000  19000  19000  19000  19000  19000  19000  19000  19000      1
PUT      /remote.php/dav/files/user5/test.txt                                              23000  23000  23000  23000  23000  23000  23000  23000  23000  23000  23000      1
--------|-----------------------------------------------------------------------------|--------|------|------|------|------|------|------|------|------|------|------|------
         Aggregated                                                                        13000  14000  19000  19000  23000  23000  23000  23000  23000  23000  23000     10

Error report
# occurrences      Error
------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------
1                  GET /remote.php/dav/files/user4/test.txt: HTTPError('404 Client Error: Not Found for url: /remote.php/dav/files/user4/test.txt')
------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------
```

Figure 2: test_1-2

Let's analyse the results:

- From **Upload Response** I've got a 201 status code, which indicates that file upload requests were successful. The HTTP 201 status code means "Created" and is a standard response for an HTTP POST request that results in the creation of a resource, in this case, a file.
- From **Download Responses** there is a mix of '404' and '200' status codes for file download requests:
- – The 404 status code means "Not Found," indicating that the requested file could not be found. This likely means that the download request was made for a file that either doesn't exist or wasn't successfully uploaded first.
- – The 200 status code means "OK," which indicates a successful file download. The response body "_ _ _ TEST _ _ _" suggests that the content of the downloaded file is correct.
- For **Response Times**, The logs don't give specific response times, but if we look at the charts from the Locust interface:
- – The "Response Times" graph shows a trend in how long requests are taking. A significant increase in response time could indicate a performance bottleneck.
- – The "Total Requests per Second" graph shows the throughput of your system — how many requests it's handling per second.
- For **Error Rate** the statistics show an error rate, which is the ratio of failed requests to total requests. An error rate above 0% means some requests are failing.
- The "Number of Users" graph shows the load ramp-up, which in this test has quickly reached the target number of users.

So, for this easy test, it can be said that the system is correctly executing file upload operations, but there are some issues with file download operations. The 404 errors are likely due to a timing issue, where the download task is executed before the file is uploaded for certain users. Since tasks in Locust are executed asynchronously and independently for each user, there's no guarantee the the upload will always happen before the download for each user instance.

The importance of analysing each results is to understand the system's behavior BUT also to go through the logs and understand the reasons behind the results and understand how to improve the system. For example, to address this specific issue, it can be develop some points:

- Adjust Task Weights, to ensure that the upload task is always executed before the download task;
- pre-upload the file for each user before starting the test, to ensure that the file is available for download;
- Add a check to the download task to ensure that the file exists before attempting to download it.

Performance testing

Performance testing is used to determine how a system performs in terms of responsiveness and stability under a particular workload. It is used to identify performance bottlenecks and to ensure that the system meets the performance requirements. For this project, it was implemented a test script for testing the performance and scalability of the system under increasing load, using Locust for generating user traffic and Docker for managing the instances.

First, it was implemented for terminal execution and used Grafana to monitor system performance and resource usage. In this case, after defining `create_users()` and `delete_users()` (as seen before, just so as not to overload the system after testing), it was set up an incremental load testing loop, where the number of users was increased by 10 every 20 seconds, with a maximum of 100 users, so it starts with `INITIAL_USERS` and increases by `INCREMENT` every `INCREMENT_INTERVAL` seconds, until it reaches `MAX_USERS`. After each test iteration, there is a pause (`seep 10`) to allow the system to stabilize before the next iteration.

```
# Incrementally run the load test
current_users=$INITIAL_USERS
while [ $current_users -le $MAX_USERS ]; do
  echo "Running test with $current_users users..."
  $LOCUST_CMD -u $current_users -r $HATCH_RATE --run-time $STEP_DURATION

  # Increase the number of users for the next increment
  let current_users+=$USER_INCREMENT

  # Short delay to let the system stabilize before the next increment
  echo "Waiting for system to stabilize..."
  sleep 10
done
```

Or, it can be implemented also to see the results in Locust web interface, where it is possible to see the response time, error rate, and other metrics in real-time (check the `locust_load.sh` and `test_incr.py` files).

Test conducted

Test 1

The first test was conducted to test the system's ability to handle an increasing number of users and loading files. The test was conducted using the Locust tool to simulate different user traffic and to monitor the system's performance and resource usage using Grafana.

Figure 3: test_1

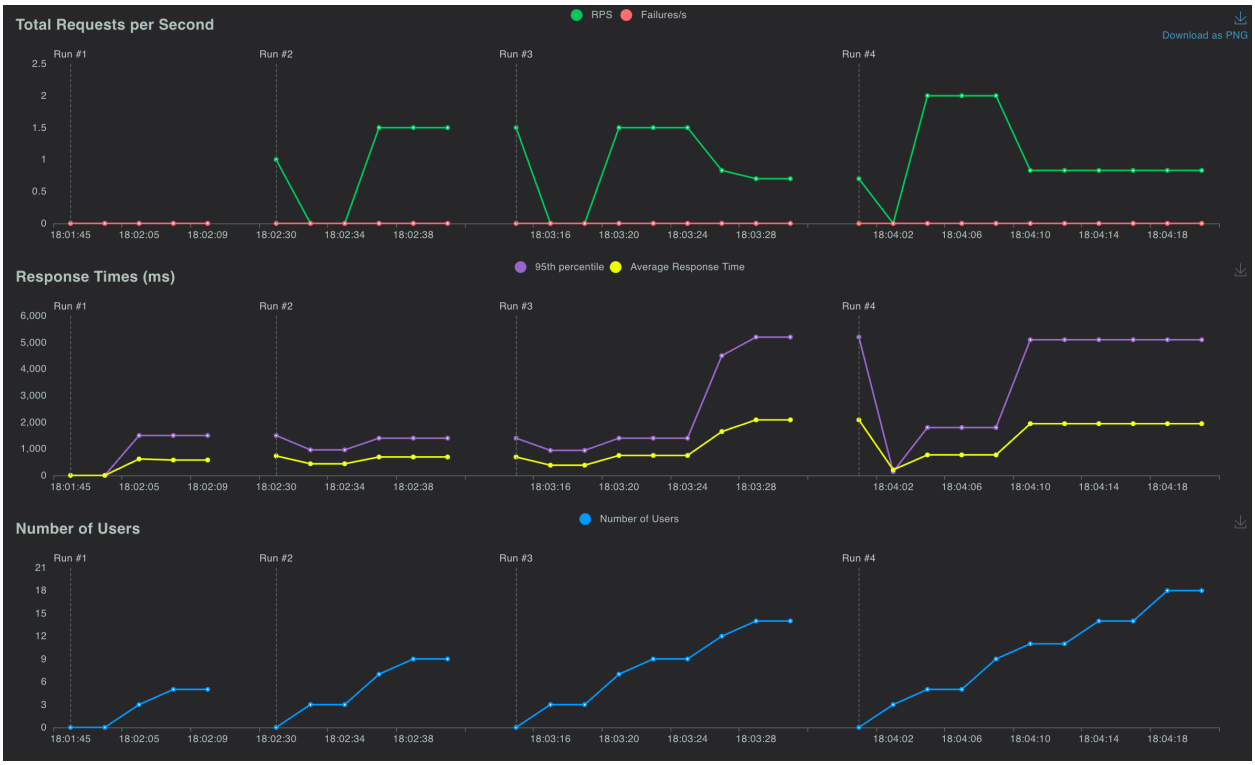


Figure 4: test_1

So, the test was splitted in 4 parts, where progressively the number of users and the duration of the test with each run was increased. The test can be described as a type o ramp-up test, and it's designed to gradually increase the load on the system to observe hiw the system's performance metrics (like response times, error rates, etc.) change under increasing load. This is a common testing pattern to find out how the system behaves under escalating stress and to identify the performance limits and scalability issues. The purpose of this test is to:

- Monitor how system performance metrics evolve as the load increases;
- Determine at what point the system's performance starts to degrade;
- Identify bottleneck and performance thresholds.

Observations and Comments

The test shows how well the system scales as more users are added. If the response times remain low and the error rate doesn't increase significantly, it means the the system scales well. If not, it means that the system has reached its performance limits and that there are bottlenecks that need to be addressed.

Test 2

In the second test, it was continued to test the system's ability to handle an increasing number of users and loading files. As before, the test was conducted using the Locust tool to simulate different user traffic and to monitor the system's performance and resource usage using Grafana. Instead of the first test, in this case, it was tested the system to reach failures and to understand the system's performance limits and scalability issues.

Figure 5: test_2

The test that was conducted is a stressed ramp-up test, in which not only the number of users was increased but also the time they are active. Thus, the server is being pushed to see how it handles the sustained load over a longer period. This test revealed that with the increase in the number of users and the prolonged duration of the test, the system starts to show malfunctions.

Observations and Comments

- If we look at DiskI/O charts, we can see that operations have been increasing over time (like the first test). This is expected as more users are making more requests, leading to more read and write

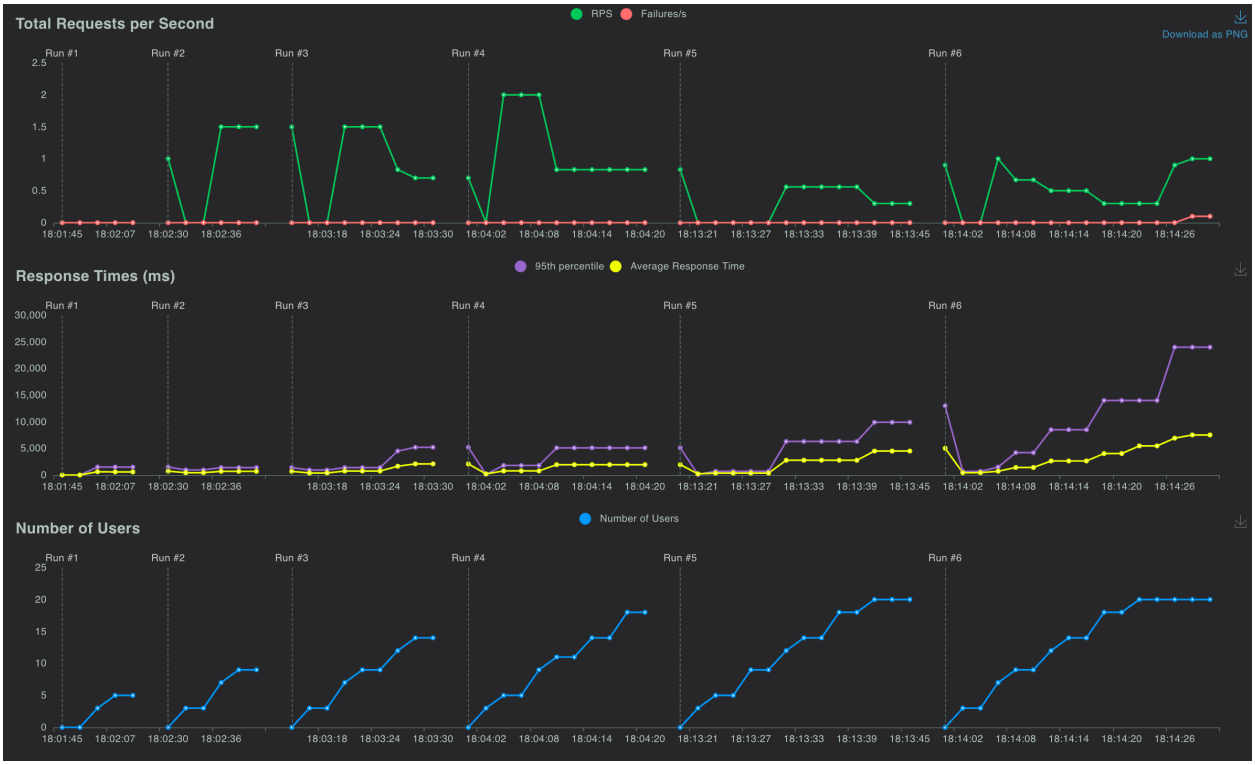


Figure 6: test_2

operations on the disk.

- The CPU Usage graph shows spikes that correlate with the testing intervals, this suggests that the CPU is under more load as more users are active.
- The Memory Usage graph is relatively stable with a slight upward trend, and this is a good sign, indicating that memory is not a bottleneck at this stage.
- In the Disk Utilization, the graph is fairly flat, which could imply that overall disk capacity is not being heavily used or that the metrix might not be capturing the write and delete operations effectively (keep in mind that that the file is small and the system is not under heavy load).
- The Network Traffic seems negligible, which may suggest that network bandwidth is not a limiting factor during these tests.
- There is a noticeable increase in response times as the number of users grows, with the 95th percentile response time showing significant spikes, indicating that while most requests are handled in a timely manner, a small percentage are taking much longer, which could point to intermittent performance issues or bottlenecks.
- The failures recorded are for the DELETE method, which means that the server was unable to respond to the file deletion requests. This could be due to the server being overloaded, a configuration issuem, or a problem with how sessions are managed.

Last but not least

In conclusion, the series of ramp-up tests conducted to assess the performance of the Nextcloud system under increasing loads have provided valuable insights into its scalability and reliability. Initial tests with 5 to 20 users for corresponding durations of 5 to 20 seconds showed the system's ability to handle incremental loads without significant degradation in performance. Metrics such as CPU usage, memory usage, and disk I/O scaled predictably with the increase in load.

However, when extending the duration to 30 seconds with a corresponding increase in the number of users, the system began to exhibit signs of stress. Specifically, we observed a 11% failure rate predominantly associated with DELETE operations, indicating potential issues with handling file deletion under sustained high loads. Additionally, response time metrics showed an increased latency, particularly at the 95th percentile, signaling that a subset of requests was experiencing considerable delays.

Throughout the tests, CPU and memory utilization did not reach full saturation, suggesting that the bottlenecks may not be due to hardware resource limits but could be related to software configuration, session management, or specific application-level constraints. Notably, the network traffic remained consistently low, indicating that network bandwidth was not a limiting factor.

The test results underscore the importance of fine-tuning server configurations, investigating application-level optimizations, and ensuring robust session management to improve overall system robustness. They also highlight the need for further analysis into the DELETE method failures to pinpoint the exact causes and implement the necessary fixes.

In the context of these findings, the system demonstrates adequate performance up to a moderate number of simultaneous users but begins to falter as the load intensifies. This provides a clear directive for focused optimization and enhancement of the system's capacity to handle high-traffic scenarios, which is imperative for maintaining service quality and reliability as user demand escalates.