Cloud Based Storage System

CLOUD COMPUTING EXAM

Marco Tallone
[SM3600002]

UniTS — 2024

Abstract

Cloud-based storage systems are becoming more and more popular, and the need for testing and deploying them in a scalable and efficient way is growing. This project aims to propose two different testing deployments for a cloud-based storage system: one based on Docker and the other based on Kubernetes.
Technical details about the two deployments are provided, and a comparison is made in terms of performance, scalability, and ease of use. Finally, a discussion about the deployment of the system in a real-world scenario is made together with some considerations about the two proposed solutions.

Contents

1 Introduction

This report presents and discusses the implementation of a cloud-based file storage system built using Docker [2] containers and Kubernetes [7] orchestration. The main objective of the project is to provide a scalable and reliable storage system that gives users the ability to manage personal files in a secure, private and efficient way.
The solution here presented makes use of the Nextcloud [15] software as the building block for the cloud storage system. Nextcloud is a free and open-source cloud storage service that allows users to store and share files, contacts, calendars and more.
Additionally, the project is divided into two main parts: the first part focuses on the deployment of the cloud storage system through Docker containers and using Docker Compose [3] to manage such containers. The second part focuses instead on the deployment of the system using Kubernetes, a container orchestration platform that allows for the automatic deployment, scaling and management of containerized applications.
For both scenarios, the scalability, security and cost-efficiency of the proposed solutions are discussed and compared. Moreover, some ideas for a real production deployment are presented, along with some possible future improvements to the system.
With the aim of maintaining a clear and concise report of the work, the following sections will only present the most noticeable aspects of the project, while further analysis and the detailed implementations will be available on the author's GitHub repository [23].

2 Docker Implementation

This section discusses the first proposed solution for the deployment of the cloud storage system, which is based on the use of Docker containers and on the Docker Compose tool to manage and orchestrate such containers. All the containers used in this deployment are taken from the official Docker Hub repository [4] and are configured to work together in a single network.
In particular, the whole project evolves around the Nextcloud software, which is used as the main building block for the cloud storage system. Therefore, before diving into the technical details of the deployment, a brief overview of the Nextcloud software is presented in the following section.

2.1 Nextcloud's User and File Management Features

Nextcloud is an ideal choice for this project due to its extensive range of functionalities tailored for cloud-based file management. It offers a user-friendly interface, robust security features, and seamless scalability.
Among the evaluated solutions, Nextcloud emerged in fact as the most compelling platform for several reasons. The primary factor influencing this decision was the ease of system implementation, particularly when utilizing Docker containers. After all, Nextcloud is supported by a vaste and fully detailed documentation that simplifies the deployment process. Additionally, since this software is open-source, it is supported and already adopted by a large number of users and developers, which ensures a high level of reliability and stability.
Among its features, Nextcloud offers file sharing, collaborative editing, and real-time synchronization. It also provides a wide range of plugins and extensions that can be used to enhance the system's capabilities.
Through the implemented web interface, users can easily log in and log out, upload, download, and manage personal files, as well as create and share folders, all while maintaining a high level of security and privacy. In other words, each user has full control over their data, but cannot access other users' data unless explicitly shared. It's also possible to assign to users different roles: for instance administrator users have advanced permissions and can not only manage the system but also tweak other users' permissions and setting, such as the available storage space. These features therefore satisfy the main requirements of a classical cloud-based storage system out of the box.
Moreover, Nextcloud provides a wide array of features beyond basic file storage, enabling the establishment of robust security measures and the integration of other software and services which will be discussed in detail in the following sections.

2.2 Deployment Details with Docker Compose

The deployment of the cloud storage system using the Docker Compose tool is summarized in a single `docker-compose.yaml` file. This file contains the personalized configuration for all the services needed to set-up and run the system, which was built by following the documentation guidelines provided by the official Nextcloud repository on Docker Hub and GitHub [13].
In detail the following services have been used for the deployment:

- **Nextcloud** ([16], version `27.1-fmp`): the image which deploys the main service of the cloud storage system, which provides the web interface for users to manage their files. Additionally, a secondary image is used to run background tasks and cron jobs.
- **PostgreSQL** ([19], version `16.3-alpine`): the image which deploys the database service for the Nextcloud software.
- **Redis** ([22], version `7.25-alpine`): the image which deploys the Redis service, used as a cache and for file locking, improving the overall performance of the system.
- **Nginx** ([18], version `1.27.0-alpine`): the image which deploys the web server service for the Nextcloud application.
- **Caddy** ([1], version `2.8.4-alpine`): the image which deploys the reverse proxy service for the Nextcloud application, providing an additional layer of security and load balancing.

In addition to these, the configuration manifests also set up a network for the services to communicate with each other, and 3 volumes to store the persistent data of the system, in particular these are associated respectively to the Caddy reverse proxy, the Nextcloud application and the PostgreSQL database. Figure 2.1 shows a simplified diagram of the deployment architecture, with the services and volumes used in the system.
Overall, these services work together to provide a fully functional cloud storage system that is secure, reliable, and scalable. Additional details and environmental variables configurations for these services can be found in the `docker-compose.yaml` file and in the `.env` file contained in the project repository.
The docker Compose tool provides a simple and effective way to deploy the system: it's in fact possible to start the whole system with a single command (`docker-compose up -d`) and to stop it with another single command (`docker-compose down`). Again further installation details are available in the project repository.

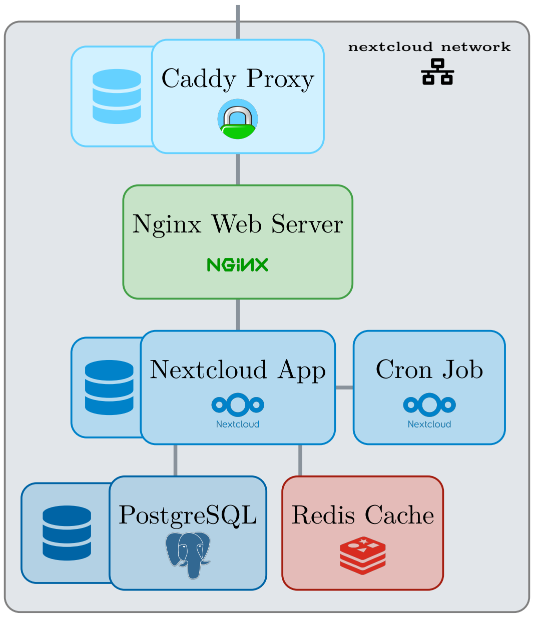

Figure 2.1: Simplified diagram of the deployment architecture with Docker containers and volumes.

2.3 Load Testing and Scalability

To evaluate the performance of the system and its ability to scale as the number of users grows, a load test was conducted using the `Locust` tool [10]. `Locust` is an open-source load testing tool that allows users to define and run load tests on web applications such as the Nextcloud one. It's in fact possible to write a test in Python that simulates the behavior of multiple users accessing the system at the same time, while a interactive web interface provides useful information about the test itself.
In the specific case of this project, a custom `locustfile.py` test was written to simulate the behaviour of 50 users interacting with the deployed Nextcloud instance. In particular, during the test, a new user was spawned at each second and then, once logged in, it was instructed to perform one among the requests reported in Table 2.1 to the system with a waiting time between 1 and 3 seconds between each request. The requests were chosen to simulate a realistic scenario in which users upload, read and download files of different sizes and, in fact, as it's possible to see from the table, the requests are associated with different probabilities. This was made possible thanks to the `@task` decorator provided by the `Locust` tool, which allows to define how many times a specific request is performed with respect to the others. These ratios are of course arbitrary and subject to eventual change, but are chosen in an attempt to reflect a realistic scenario. Moreover, the file sizes selected are also arbitrary but have been chosen to assess the system's I/O performance with different file sizes.
The test was conducted on a local machine equipped with an Intel Core i7-8565U CPU and 8 GB of DDR4 RAM, running Arch Linux as the main operating system. Figure 2.2 shows the results of the test, which lasted for about 2 minutes.
Looking at the results, it's possible to see that the system was able to handle the load test without any major issues. At full capacity, in fact, it was able to handle an average of 30 requests per second with at most 1.5 failures per second. Indeed, the overall failure rate remained below 5% throughout the test. The response times were also quite good, with the 50^{th} percentile reaching a peak of ~ 2000 ms during the login phase of the last users but then quickly dropping to ~ 200 ms for the rest of the test.
Of course, these results only constitute a test given the hardware at disposal and are subject to the specific ratio and user spawning rate chosen for the system.

Request	**`@task` Ratio**	**Probability**
Read a file	10	32.3%
Download a file	5	16.1%
Upload a 1 KB file	10	32.3%
Upload a 1 MB file	5	16.1%
Upload a 1 GB file	1	3.2%

Table 2.1: Different requests and their respective probabilities during the load test. The `@task` ratio indicates how many times a specific request is performed with respect to the others.

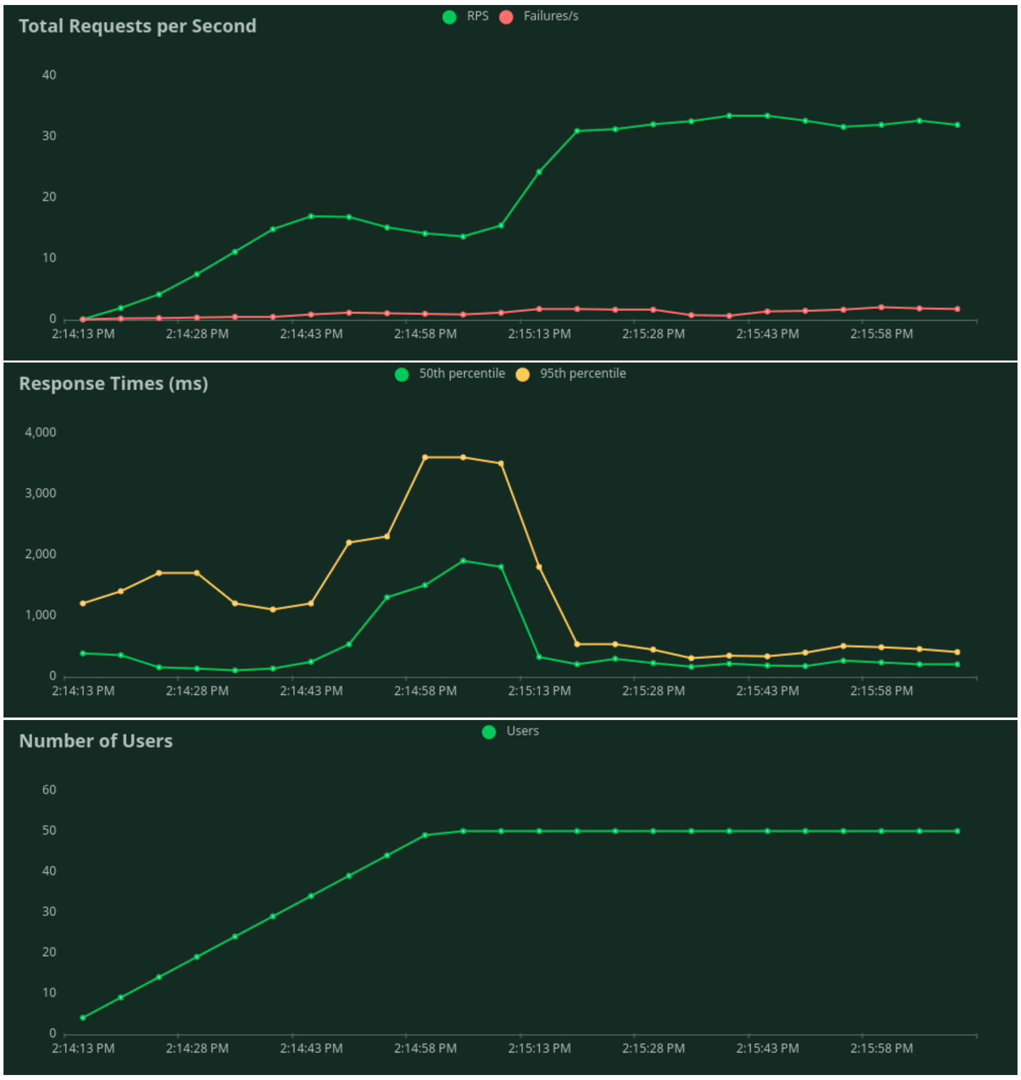

Figure 2.2: Results of the load test conducted with the Locust tool. The top plot shows the number of requests per second (●) and the respective failures (●), the middle plot shows the response times (● 50^{th} percentile, ● 90^{th} percentile) and the bottom plot shows the number of users over time (●).

2.4 Security Assessment

Nextcloud comes equipped with a wide range of security features that, although not strictly necessary for the core functionalities of the system, are essential in real-world scenarios to guarantee the safety and privacy of the users' data. These features can be enabled from the web interface by accessing the Nextcloud instance as an administrator. In particular, most of these can be found under `Administration settings` → `Security`. Among the different security features available, the following are the most notable:

- **Server-side encription** (SSE): this feature allows to encrypt the files stored on the server side, so that even if the server is compromised the data remains safe. This security measure is unnoticed by the user, who can still access and manage the files as usual, since these are decrypted on-the-fly.
- **Password policies and brute-force protection**: Nextcloud offers multiple features to strengthen user authentication and password security. Among these we find for instance the possibility to set a minimum password length, to enforce the use of special or upper/lower case characters, to set a maximum number of login attempts before blocking the user, and to set a time limit for password expiration. Additionally, it's also possible to enable user password history so that the user cannot reuse the same password multiple times and there's also the possibility to validate the password against the `haveibeenpwned.com` database of known data breaches [21].
- **Two-factor authentication** (2FA): this feature adds an additional layer of security to the user authentication process, requiring the user to provide a second piece of information beyond the password. This can be a security code sent via email or SMS, a code generated by a mobile app, or a physical security key.

Besides the security measures provided by the Nextcloud service, another important aspect to consider, especially when deploying the system in a real-world scenario, is the communication between the end-users and the server itself. In fact, the data exchanged between the client and the server must be encrypted to prevent malicious actors from intercepting and reading the sensitive information. To achieve this, the Nextcloud instance is deployed with the Caddy reverse proxy, which provides an additional layer of security by enabling HTTPS encryption and automatically managing the SSL certificates.

2.5 Production Deployment and Cost-Efficiency Considerations

The deployment presented so far is of course a simplified version of the system suitable for testing and development purposes. In a real-world scenario, some additional considerations must be taken into account to ensure the system's reliability, security, and scalability. In particular, not only the requirement of the system will be higher (with respect to the simple load test conducted above) due to the supposed increase in the number of users, but also the need for high availability and fault tolerance will be crucial. To achieve these goals, the following preliminary aspects regarding the overall scalability of the system must be considered.
First of all, if we insider the current deployment, we can imagine that some of the containerized services could be scaled either horizontally, i.e. by adding more instances of the same service, or vertically, i.e. by increasing the resources allocated to the service. For instance, starting from the database instances, it would make sense to scale the redis container vertically to allocate more capacity to the cache system, while scaling horizontally the PostgresSQL container both to handle more requests and to ensure fault tolerance by replicating the data in multiple instances. On the other hand both the Nextcloud and the Nginx containers could be scaled horizontally so that the incoming increased traffic could be balanced among multiple instances. Related to this, the Caddy container could be in fact scaled vertically so that it could load balance the incoming traffic among multiple Nginx instances and manage the SSL certificates.
Of course these are just preliminary considerations, in a real-world scenario scaling the system comes with a cost which must be carefully evaluated. Therefore, we can now examine what are the different practical options for a real organization to deploy the system in a production environment.

Supposing initially that the given organization has access to a data center infrastucture, the most straightforward solution would be to directly deploy the system on the organization's servers, i.e. **on-premises deployment**. This solution comes with some advantages, in particular:

- **Full control over the infrastructure**: the organization has full control over the hardware and software used to deploy the system, which allows for a high level of customization and optimization. This also implies that the organization can choose the most suitable location for the servers, which can be crucial for ensuring low latency and high availability.
- **Cost efficiency**: deploying the system on the organization's servers can be more cost-effective in the long run, as there are no monthly fees associated with cloud providers. Additionally, the organization can optimize the resources allocated to the system based on the actual usage, which can lead to significant cost savings.
- **Security and privacy**: by deploying the system on the organization's servers, the organization can ensure a high level of security and privacy, as the data remains within the organization's network and is not shared with third-party providers.

However, we might also consider the drawbacks of an on-premises deployment:

- **Initial setup and maintainence cost**: deploying the system on the organization's servers requires an initial investment in hardware and software, which can be significant depending on the size of the organization and the requirements of the system. This is also relevant in terms of a cost-efficiency analysis, since in my test scenario I deployed the free version of the Nextcloud software, but in a real-world scenario the organization might consider to buy the *Nextcloud Enterprise* [14], which is a pre-packaged version of the service offering various service tiers and plans.
- **Scalability and flexibility**: deploying the system on the organization's servers can limit the scalability and flexibility of the system, as the organization is responsible for managing and maintaining the infrastructure. This can be a significant drawback in case of sudden increases in traffic or storage requirements, as the organization might not have the resources or expertise to scale the system accordingly.

Given these considerations, a second possibility, relevant in the more probable scenario in which an organization doesn't have its own data center, would be to deploy the system on a **cloud provider**, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure. This solution actually comes with multiple sub-options which we can analize in detail.
For instance, in the case of the organization not having at disposal the necessary hardware resources, the **Infrastructure as a Service** (IaaS) solution would be the most suitable. In this case, the organization can rent virtual servers from the cloud provider and deploy the system on these servers. The advantages of this solution are:

- **No initial investment**: the organization does not need to invest in hardware, as the cloud provider takes care of the infrastructure. This can be particularly advantageous for small and medium-sized organizations that do not have the resources to set up their own data center.
- **Scalability and flexibility**: the organization can easily scale the system based on the actual usage, as the cloud provider offers a wide range of services and resources that can be provisioned on-demand on a "*pay-as-you-go*" basis.

Of course, in this case the organization will still have to manage and set-up the system's software with the desired level of customization and optimization.
On the other hand, the drawbacks of the IaaS solution are:

- **Limited control over the infrastructure**: the organization has limited control over the underlying infrastructure, as the hardware and software are managed by the cloud provider. This can be a concern in terms of security and privacy, as the organization's data is stored on servers managed by a third-party provider.

- **Cost inefficiency**: while the IaaS solution can be cost-effective in the short term, it can become more expensive in the long run as the organization scales the system. This is because the cloud provider charges for the resources used, hence the organization is subject to the provider's pricing model.
- **Vendor lock-in**: deploying the system on a cloud provider can lead to vendor lock-in, as the organization becomes dependent on the provider's services and technologies. This can limit the organization's ability to switch to another provider or to bring the system back on-premises.

Another possible solution would be to deploy the system on a **Platform as a Service** (PaaS) solution, such as Heroku or Google App Engine. In this case, the organization can deploy the system on a platform provided by the cloud provider, which takes care of the underlying infrastructure and provides a set of services and tools to develop, deploy, and manage the system. The advantages of the PaaS solution are:

- **Simplified deployment**: the organization can deploy the system with minimal effort, as the platform provides a set of tools and services to automate the deployment process. This can be particularly advantageous for organizations that do not have the expertise to manage the infrastructure themselves. In this case, the organization basically can focus on simply managing the system and the data.
- **Scalability and flexibility**: the organization can easily scale the system based on the actual usage, as the platform offers a wide range of services and resources that can be provisioned on-demand. This can be significant in case of sudden increases in traffic or storage requirements.

The cons of this approach are, once again, mostly similar to the IaaS solution previously discussed with minor differences.
Finally, a third possible solution would be to deploy the system on a **Software as a Service** (SaaS) solution, such as Nextcloud itself. This is the least demanding solution in terms of management and maintenance, since the organization doesn't have to worry about anything and just use the service by paying for it. Clearly, this comes with the advantages:

- **No management and maintenance**: the organization does not need to manage nor maintain the system, as the service provider takes care of everything (hardware, software, data management, security, etc.).
- **Scalability and flexibility**: the organization can seamlessly scale the system based on the actual usage, as this is managed by the service provider.

Once again, the disadvantages in this case are mostly similar to the IaaS and PaaS solutions, with some additional considerations:

- **Cost inefficiency**: the SaaS solution can be the most expensive option in the long run when compared to the previous solutions.
- **Limited customization and optimization**: the organization has limited control over the system and possibly also no knowledge of the underlying infrastructure.
- **Security and privacy concerns**: the organization's data is stored on servers managed by a third-party provider, which can raise concerns in terms of security and privacy.

3 Kubernetes Implementation

In this section, a secondary deployment strategy for the cloud-based storage system is discussed. In particular, this more advanced approach is based on the use of the well-known Kubernetes platform and the Helm package manager [6]. The requirements were to configure a single-node Kubernetes cluster on a local machine and deploy the cloud-based storage system on it. This was achieved by setting-up, thanks to Vagrant [24] and Libvirt [9], a virtual machine running a `Fedora/39-cloud-based` image [5] and installing on it the necessary software. Additionally the system has also been tested on a minikube cluster [12] for comparison.

3.1 Deployment Details with Kubernetes

The deployment of the cloud-based storage system on the Kubernetes cluster was performed using Helm. In detail, the official Helm chart for the Nextcloud service was used following the instructions provided in the official repository [17] and implementing a custom `values.yaml` file. More precisely, the chart deploys Nextcloud pods containing in total 3 containers: one for the Nextcloud service, one for background tasks and crone jobs (just as in the above deployment) and one containing a nginx web server instance. The custom values were also set to use a PostgreSQL database and a Redis cache running in separate independent pods just like in the Docker deployment. Additionally, a Nginx Ingress Controller [8] was deployed to attempt exposing the Nextcloud service outside the cluster. Moreover, in an attempt to simulate a real-world loadbalancer, the MetalLB project [11] was used to provide a Layer 2 load balancer for the Nginx Ingress Controller and the Nextcloud services.
While figure 3.1 provides a representation of the deployment just described with one replica of the Nextcloud service, further details about the implemented code and the deployment process can be found in the documentation provided in the project repository. The major differences with the Docker deployment are the use of Helm and the Kubernetes platform. The services are not now deployed as simple standalone containers, but are encapsulated in Pods and managed by the Kubernetes orchestrator. This allows for a more complex and scalable deployment, but also requires a more complex setup and configuration. In addition, the access to the services is now managed by a Kubernetes `Ingress` Controller resource rather than through a reverse proxy such as was done with the Docker deployment above.
For what concerns credentials and environmental variables, in this case these were implemented as Kubernetes `Secrets` and `ConfigMaps` rather than as simple variables in a `.env` file as in the above case.
There are also differences in terms of the persistent storage management which will be discussed in the next section.

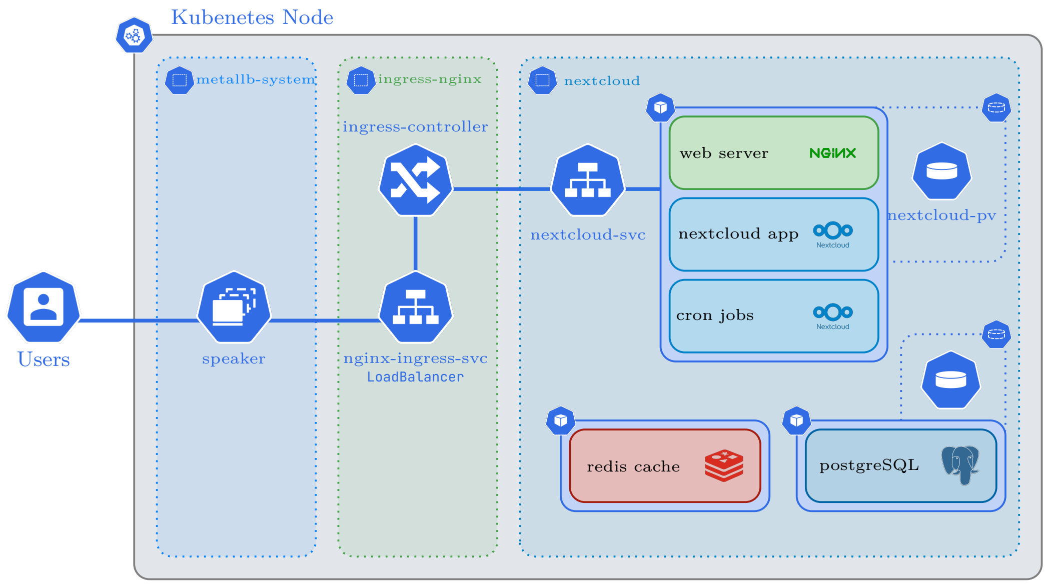

Figure 3.1: Kubernetes Deployment of the Cloud-Based Storage System

3.2 Back-End Storage on Kubernetes Deployment

Avoiding data loss in case of pod restarts or failures is a critical aspect of the deployment of stateful applications on Kubernetes. In the case of this cloud-based storage system, `PersistentVolume` (PV) and `PersistentVolumeClaim` (PVC) resources were used to provide persistent storage for the PostgreSQL database and the Nextcloud service. In particular, what has been done is to map a folder on the host-machine to a persistent volume in the Kubernetes cluster by using a local path provisioner.
Of course, this is a very basic setup suitable only for this kind of testing environment and in fact comes with some inherent limitations. For example, the data is not replicated at all, implying that in case of failure of the host machine, the data would be lost. Moreover, the persistent volumes are required to exist on the host machine before the deployment of the services, meaning that dynamic provisioning is not possible. Additionally, for the case of this Vagrant set-up, since the PV are bound to a specific node, the services using such volumes must be scheduled on that node, limiting the scalability of the deployment. This solution works for this simple single node set-up but it's of course not scalable and hence not suitable for real production environments...
To avoid such limitations in a production environment, a more complex storage solution would be required, such as a distributed storage system.

3.3 High Availability and Scalability Considerations

A few considerations could be made for what concerns having the service in high availability and improving the overall scalability of the deployment.
For sure, the first step would be to extend the current set-up to be working in a multi-node cluster, rather than a single-node one. This would however require for the deployment of multiple replicas of the services and the use of a more complex load balancing system. Pods should then be scheduled on multiple nodes rather than being tied to a single one as in this test deployment. At least for what concerns the Nextcloud service, the provided helm chart gives the possibility to tweak the `replica` parameter to have multiple replicas of the service. Additionally, the chart also allows to efficiently set-up a Horizontal Pod Autoscaler (HPA) to automatically scale the number of replicas based on CPU, memory usage or other metrics.
In this context might result helpful the usage of the optional Prometheus metrics exporter [20] provided by the Helm chart itself to monitor the Nextcloud service and the deployment of multiple `probes` that can also be configured through the very same helm chart. For the case of this project I experimented configuring startup, readiness and liveliness probes to guarantee correct functioning and failover capabilities to the Nextcloud pods.
As anticipated above, a more complex storage solution would be required to provide high availability and scalability for the data storage. In particular, an important aspect would be surely to have data replication and failover capabilities. Another aspect to keep in mind is also the one regarding data encryption and security, which would be crucial in a real-world deployment. For the current implementation, the only encryption settings available are the security measures that can be set-up inside the Nextcloud application as explained in the previous sections.
Finally, for what concerns the load balancing system, the MetalLB project used in this deployment is a good starting point, but in a real-world deployment a more complex and efficient load balancer would be required, although in most cases this would be provided by the cloud provider itself.

3.4 Comparison with Docker Deployment

Finally, it's possible to compare the Kubernetes deployment with the Docker one discussed in the previous section. The Kubernetes deployment is more complex and requires more resources to be set-up, but it also provides a more scalable and resilient solution. In detail, as an advantage the proposed Kubernetes deployment can be easily scaled up by either adding more nodes to the cluster or by increasing the number of replicas of the services as explained in the previous section. Moreover, the Kubernetes platform provides a more advanced orchestration system that can manage the services in a more efficient way, expecially thanks to the use of the HPA mentioned above.
While keeping this in mind however, we shall also pay attention to possible drawbacks of the Kubernetes deployment. First of all, in some cases the added complexity might

not be worth the benefits, especially if we connect to the real-world deployment considerations made above: consider for example the case of a small company with limited resources. In this case, the Docker deployment might be more suitable as such organization might not have the resources to set-up a Kubernetes cluster and hire dedicated personnel to manage it. Additionally it's worth mentioning that, although it might not be really concerning for real-world deployments, Kubernetes has minimum requirements in terms of resources that must be met to run properly as it infact requires machines to be equipped with at least 2GB of RAM and 2 CPUs (see `Vagrantfile` in project repository for details).

4 Conclusions

In conclusion, this project proposed two different testing deployment for a cloud-based storage system. The first one was based on Docker, while the second one was based on Kubernetes. The two deployments have been extensively tested and compared in terms of performance, scalability, and ease of use. It has been discussed how the Kubernetes solution, while being more complex to set up, results to be more scalable and probably more suited for a large production environment with some adjustments to the presented project. However, the Docker solution still remains a valid choice for small-scale deployments or for testing purposes, expecially for its set-up semplicity. Considerations about the deployment of the system in a real-world scenario have been made and of course apply in the same way to both proposed solutions.

References

[1] Caddy. Caddy docker image, 2024. URL: https://hub.docker.com/_/caddy.

[2] Docker. Docker, 2024. URL: https://www.docker.com/.

[3] Docker. Docker compose, 2024. URL: https://docs.docker.com/compose/.

[4] Docker. Docker hub, 2024. URL: https://hub.docker.com/.

[5] Fedora. Fedora/39-cloud-base, 2024. URL: https://app.vagrantup.com/fedora/boxes/39-cloud-base.

[6] Helm. Helm, 2024. URL: https://helm.sh/.

[7] Kubernetes. Kubernetes, 2024. URL: https://kubernetes.io/.

[8] Kubernetes. Nginx ingress controller, 2024. URL: https://docs.nginx.com/nginx-ingress-controller/installation/installing-nic/installation-with-helm/.

[9] Libvirt. Libvirt, 2024. URL: https://libvirt.org/.

[10] Locust. Locust, 2024. URL: https://locust.io/.

[11] MetalLB. Metallb, 2024. URL: https://metallb.universe.tf/.

[12] Minikube. Minikube, 2024. URL: https://minikube.sigs.k8s.io/docs/.

[13] Nextcloud. Docker examples, 2024. URL: https://github.com/nextcloud/docker/tree/master/.examples.

[14] Nextcloud. Enterprise, 2024. URL: https://nextcloud.com/enterprise/.

[15] Nextcloud. Nextcloud, 2024. URL: https://nextcloud.com/.

[16] Nextcloud. Nextcloud docker image, 2024. URL: https://hub.docker.com/_/nextcloud.

[17] Nextcloud. Official helm chart, 2024. URL: https://github.com/nextcloud/helm/tree/main/charts/nextcloud.

[18] NGINX. Nginx docker image, 2024. URL: https://hub.docker.com/_/nginx.

[19] PostgreSQL. Postgresql docker image, 2024. URL: https://hub.docker.com/_/postgres.

[20] Prometheus. Prometheus, 2024. URL: https://prometheus.io/.

[21] Have I Been Pwned. Have i been pwned, 2024. URL: https://haveibeenpwned.com/.

[22] Redis. Redis docker image, 2024. URL: https://hub.docker.com/_/redis.

[23] Marco Tallone. Cloud storage system, 2024. URL: https://github.com/marcotallone/cloud-storage-system.

[24] Vagrant. Vagrant, 2024. URL: https://www.vagrantup.com/.