DETI - 2023/2024

Reliable Deployment

Report 2

Diogo Oliveira Magalhães	102470
Leonardo Almeida	102536
Pedro Henrique Figueiredo Rodrigues	102778

Gestão de Infraestruturas de Computação

Professor: João Paulo Barraca

June 7, 2024

Contents

Chapter 1

Introduction

In the previous work, we successfully deployed the Wesago application and the static website in a Kubernetes cluster. We started by analyzing the components of the application, the interactions between them, and the dependencies that exist. We also designed a deployment strategy that takes into account the requirements and limitations of the application and the cluster environment. To achieve the final goal, several Kubernetes resources were used including Deployments, StatefulSets, Services, ConfigMaps, Secrets, PersistentVolumeClaims, Ingresses, and Jobs. In Figure 1.1 we can see the deployment strategy that was implemented.

After this initial deployment, our goal was to improve the robustness of the application to make it more resilient and reliable. This report presents the work done to achieve this goal which consists of the following factors:

- **Automatic Deployment:** With a single click, the application should be deployed in a Kubernetes cluster. This should be able to deploy the application from scratch, including creating and pushing the images to the container registry.
- **Healthchecks**: Regular health checks can be implemented to monitor the health of individual instances. This can help to identify failing instances before they impact the overall system. Unhealthy instances can then be automatically restarted or replaced.
- **Redundancy**: The application should be able to handle failures of individual instances. This can be achieved by running multiple instances of the same component. To further improve redundancy, deploying pods across different nodes in the cluster will help to reduce the impact of node failures on the system.
- **Autoscaling**: The system can be improved by implementing autoscaling. This will allow the system to automatically adjust the number of instances based on the load and help to reduce costs and improve performance.
- **Monitoring and Observability**: Monitoring and observability tools can be implemented to provide insights into the system's performance and health, including the use of metrics and logs. This can help to identify bottlenecks and issues before they impact the system.

- **Disaster Recovery**: A disaster recovery plan can be implemented to ensure that the system can recover from catastrophic failures. This can include regular backups of data and configurations, as well as a plan for restoring the system in the event of a failure.

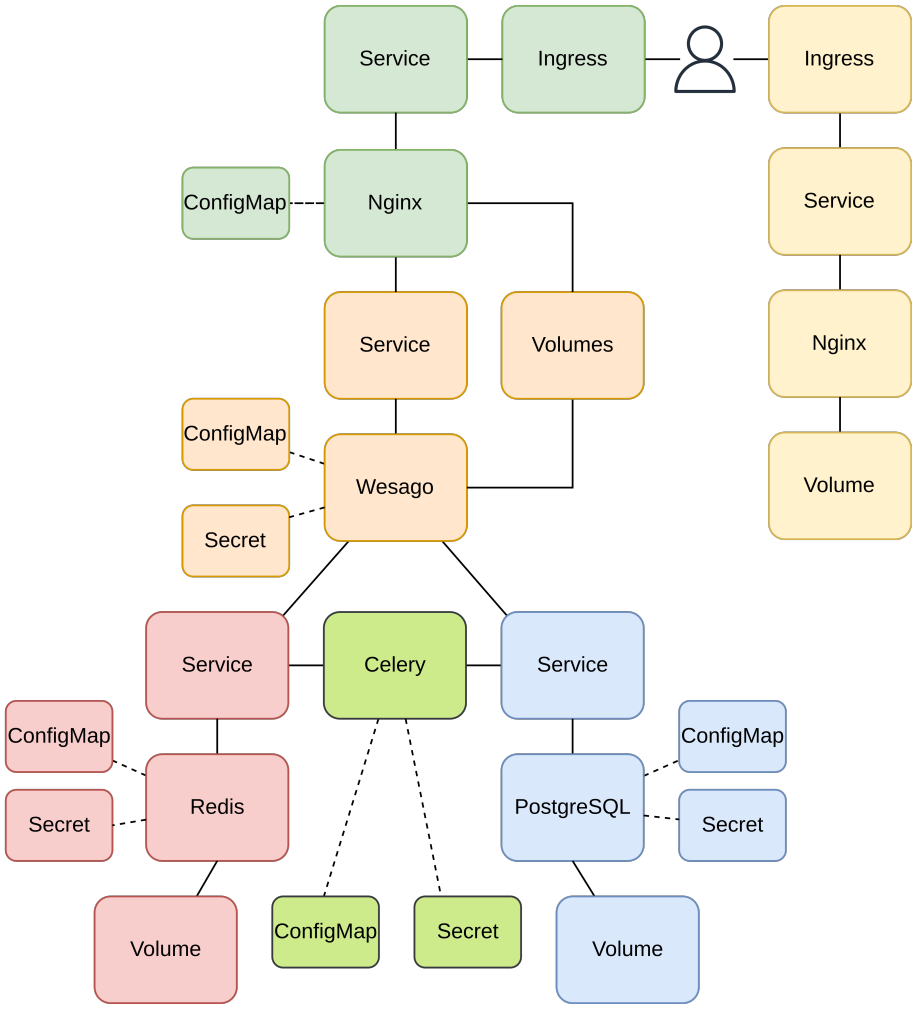

Figure 1.1: Initial deployment strategy

In the following chapters, we will present the work done to achieve these goals and the challenges faced during the implementation.

Chapter 2

Automatic Deployment

To ensure that the application deployment can be easily replicated, we implemented an automatic deployment process that allows the deployment of the application with a single click. An important part of this process is the modularity of the deployment workflow, which allows the addition of new components easily since each component has its deployment script that is executed independently. This modularity also allows the deployment of individual components, which can be useful for debugging and testing purposes. The deployment process consists of the following steps:

- **Building the Docker Images:** The first step is to build the Docker images for each component of the application. This is done by executing the run script using the "images" option. After creating each image it also pushes them to the container registry.
- **Kubernetes Infrastructure:** This includes creating the necessary Kubernetes resources for the application, such as Deployments, StatefulSets, Services, ConfigMaps, Secrets, PersistentVolumeClaims, Ingresses, etc.
- **Job Execution:** This step includes running the necessary jobs to initialize the application, such as database migrations, static file generation, and other one-time tasks.

With automatic deployment, the application can be deployed at any time if necessary, and the deployment process can be easily replicated in different environments. This is especially useful for testing and development purposes, as well as for disaster recovery scenarios.

Figure 2.1 illustrates in a more detailed way the strategy adopted for the automatic deployment. As the diagram shows, we have 3 main folders: one with the components of the main application, another with the website components, and a third one containing the monitoring components. Each component has its own run script that creates the resources according to the manifest files. Then each folder that aggregates the components has a run script that orchestrates the deployment of all the components. Finally, there is another run script in the root folder that orchestrates the deployment of all the components of the application. This root script is also responsible for building the Docker images and pushing them to the container registry.

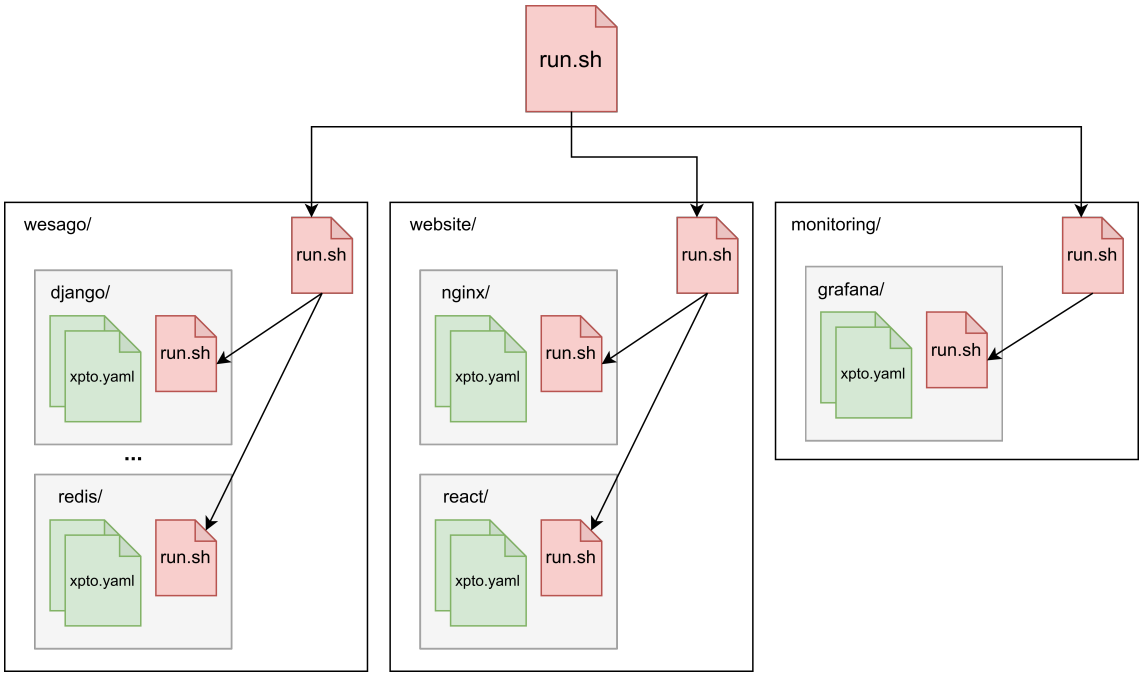

Figure 2.1: Automatic deployment strategy

Chapter 3

Healthchecks and Auto-Healing

Kubernetes has several built-in mechanisms for monitoring and restoring applications running in the cluster. If a container of a pod crashes, Kubernetes will automatically detect it and restart the pod to ensure that the application becomes available as soon as possible. However, this mechanism is not enough to ensure the reliability of an application, as it only detects crashes and does not take into account the state of the application itself. For example, a container can be running but the application inside can be found in a deadlock, this means that the application is running, but unable to make progress due to some issue.

Due to this problem, Kubernetes provides a way of defining health checks for applications running in the cluster that can be used to monitor the health of an application and take action if the application is found to be in an unhealthy state. For this, Kubernetes offers three kinds of health checks, where each probe serves a distinct purpose and can be configured according to the application's use case. The three types of Kubernetes health checks are:

- **Liveness Probe:** This probe is used to detect when an application is not responsive and needs to be restarted. If the liveness probe fails, Kubernetes will restart the container to restore the application to a healthy state.
- **Readiness Probe:** This probe is used to ensure that a service is ready to receive traffic. If the readiness probe fails, the pod will be removed from the service load balancers, and traffic will not be routed to it until the probe passes.
- **Startup Probe:** This probe is used to identify and delay the startup of an application until it is prepared to handle requests. While the startup probe fails, liveness and readiness probes do not start, which is usefull for slow starting applications.

In our deployment, we implemented liveness probes to ensure that the applications were responsive and could recover from failures. If, for some reason, the application becomes unresponsive, Kubernetes will automatically restart the container to restore the application to a healthy state. The other types of probes were not used since, overall, our applications start very fast.

Depending on the component, each liveness probe had to be differently configured to ensure that the application was correctly monitored. For example, the liveness probe for the Django

application, shown in Figure 3.1, was configured to check if the application was responsive by making an HTTP GET request to the root endpoint, while the liveness probe for the PostgreSQL database, shown in Figure 3.2, was configured to check if the database was responsive by running a bash command with a SQL query to verify if we would get a response.

```
livenessProbe:
  httpGet:
    path: /
    port: 8000
  initialDelaySeconds: 60
  periodSeconds: 10
  timeoutSeconds: 5
  # successThreshold: 1
  failureThreshold: 1
```

Figure 3.1: Django liveness probe

```
livenessProbe:
  exec:
    command: ["psql", "-w", "-U", "
        postgres", "-d", "postgres",
        "-c", "SELECT 1"]
  initialDelaySeconds: 120
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 2
```

Figure 3.2: PostgreSQL liveness probe

One problem with this type of health check is that it runs the commands directly inside the container, which can lead to false positives. For example, if we want to access a database, it might be accessible from the container, but not from the outside. To solve this problem, we can use an external health check pod that runs a container that tries to access the application to ensure the connection. For this, the pod had access to the cluster Kubernetes API, run a command to check if each of the applications was accessible from the outside and if not, restarted the pod. For this, we used `kubectl get pods` with a custom *jsonpath* to get each of the pods' names, status, IPs and timestamp, and then tried to access the application using that IP using different methods for each component. If the application was not accessible, the pod was restarted.

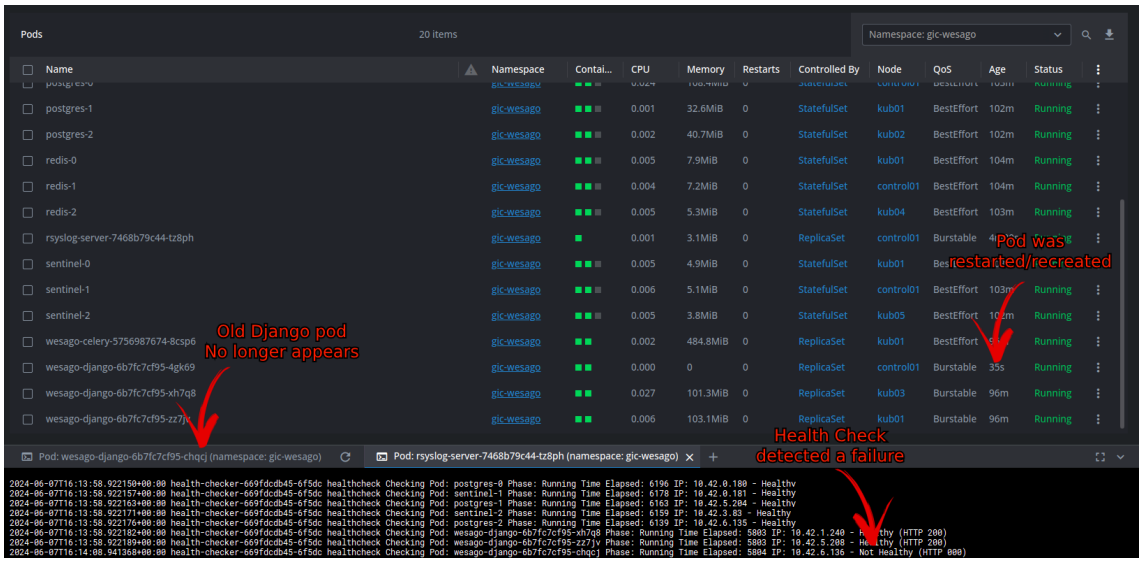

Figure 3.3: Django Auto-Healing due to health check

To test the Auto-Healing feature with the health checks, we simulated a failure in the Django

application by killing the process running inside the container. As we can see in Figure 3.3, after the process was killed, the health check failed, and the container was restarted by Kubernetes. In order to ensure that both approaches were working, we repeated the test using only one of the solutions at a time and in both cases, the container was restarted.

Lastly, we also implemented a custom health check that tried to access the ingresses of the application through the internet to ensure that the application was accessible from outside of the Kubernetes cluster network. This was done very similarly to the previous configuration, but this time by running a script that tried to access the application through the internet using the ingress IP. If an error occurred, the pod would create an alert by writing a log for analysis, since this would need to be taken care of manually, since we could not ensure that the error was coming from the application itself and not from the network.

Due to the different probability of finding a relevant error in each of the health check methods, each of them has a different time interval for running the health checks. The first method is the one that runs most frequently, every 10 seconds, the second runs every 30 seconds, and the last one runs every 60 seconds. This way, we tried to not overload the cluster with unnecessary checks since, if the first method was failing for a pod, the other two would also fail.

Chapter 4

Redundancy

As mentioned in chapter 1, redundancy is an important factor in improving the reliability of the application and it has 2 main aspects: running multiple instances of the same component and deploying pods across different nodes in the cluster. In this chapter, we will discuss the work done to achieve these goals.

4.1 Instance Replication

Running multiple instances of the same component is a simple way to improve redundancy, if one instance fails, the other instances can continue to handle the requests, ensuring that the application remains available and/or that the data is not lost. In Kubernetes, there are already some resources that support instance replication by simply adding the *replicas* field to the resource configuration. In our case, we used at least 3 *desired replicas* for each component to ensure that the application remains available even if one instance fails.

Even though this is a simple way to achieve redundancy and should work out of the box for many components, particularly for stateless components, it is not enough for stateful applications, such as databases, since, for example, the data is not replicated across the instances by default, which could lead to data loss in case of a failure and/or wrong data consistency.

In our deployment, we found 2 components that required special attention regarding instance replication, since just changing the number of replicas was not enough to ensure redundancy and data consistency:

- **PostgreSQL:** Responsible for storing all the data related to the application and, the eventual failure of this component can lead to the loss of all the data needed to properly execute the application which might not be possible to recover from. If replicated instances are not properly configured, data consistency can be compromised since each instance can have different data.

- **Redis:** Functions as a cache and as a message broker. If it was just for the cache functionality, the usage of a Kubernetes deployment would be enough since even if the cache was

lost, we would only notice a decrease in performance and not a complete failure of the application and this could be easily recovered. However, since Redis is also used for message brokering that will eventually lead to writes in the PostgreSQL database, we also need to ensure that the Redis component is highly available and the messaging data is not lost in case of failure.

To ensure redundancy and data consistency for these components, we used the following strategies:

PostgreSQL

For the PostgreSQL database component, we thought of two possible approaches for the replication of the component:

- **Using a fixed master-slave configuration:** This is a scalable solution in which we would have a master database that would be responsible for writing and reading all the data and multiple slave databases that would be responsible for having that data replicated and for reading the data. This approach would improve the performance of the application since the read operations could be distributed across multiple databases, not overloading the master database.
- **Using a dynamic master-slave configuration:** This approach would be more complex to implement but would provide an extra robustness solution in case of a failure. In this case, we would have a master database that would be responsible for writing and reading the data and multiple slave databases that would be responsible only for reading the data, just like in the previous configuration. However, in case of a failure of the master database, one of the slave databases would be promoted to master and the application would continue to work without any issues.

After some discussion and analysis of the possible approaches, we decided to go with the first method of using a statefulset with multiple replicas and a fixed master-slave configuration. This approach provides better scalability and performance to the database, than not using replication at all, and is simpler to implement than the dynamic master-slave configuration. The problem with the dynamic master-slave configuration is that it would require the usage of operators to automate the failover process since PostgreSQL does not provide this functionality out of the box, and the no allowed usage of operators was one of the limitations of the project.

The final replication strategy of the PostgreSQL database can be seen in Figure 4.1.

To test the replication of the PostgreSQL database, as seen in Figure 4.2, we simulated the deployment composed of a master (left terminal) and a slave (right terminal). As it can be seen in the image, initially, both the master and the slave are synchronized having no data in each of them. By inserting data only in the master, we can see from the image that the created table was replicated to the slave.

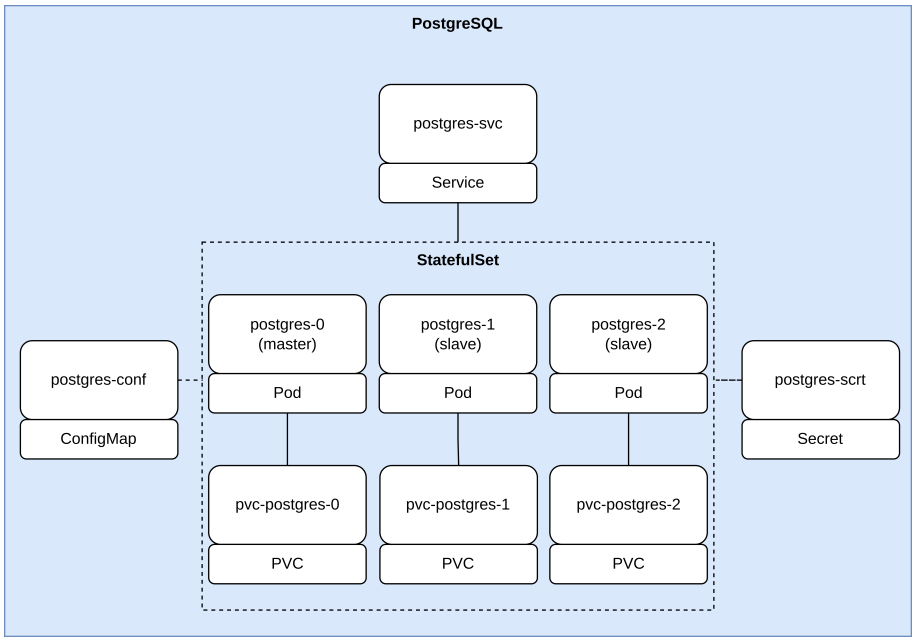

Figure 4.1: PostgreSQL Architecture

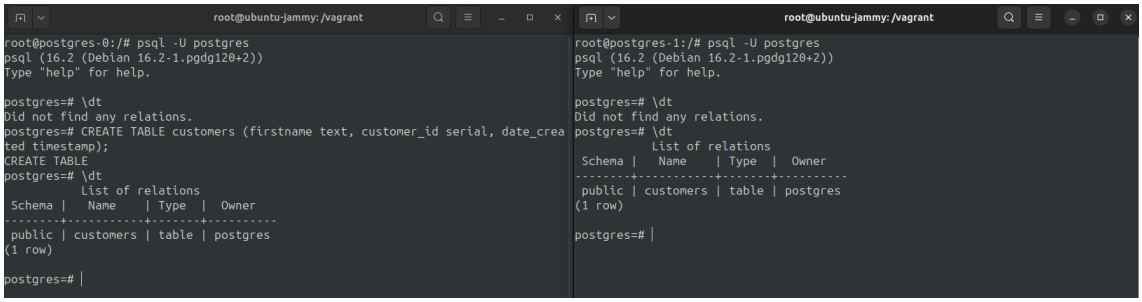

Figure 4.2: PostgreSQL Replication

The biggest problem with this approach is that since we are not updating the master-slave configuration of the PostgresSQL deployment dynamically, in case of a master failure, there is a considerable downtime while the master pod is being rebuilt, automatically by the Kubernetes control-loop mechanism or manually. This limitation might cause some difficulties to the application as it will not be able to add new data to the database. On the other hand, if the data is only being read, the application should keep working normally and data and its consistency will be preserved.

Redis

For the Redis component, we had a similar problem to the PostgreSQL database, having the same two possible approaches for the replication of the component:

- **Using a fixed master-slave configuration:** Scalable solution where data writes can only be done in the master replicas and data reads can be done in any replica. In case of a master

failure, the system would not be fully available.

- **Using a dynamic master-slave configuration:** More complex approach, with the same benefits as the previous configuration, but in case of a master failure, one of the slave replicas would be promoted to master and the system would continue to work without any issues.

After some discussion and analysis of the possible approaches and the application functionalities, this time we decided to go with the second method. This approach joins both high scalability and high availability of the Redis component and it is possible to be implemented without using operators since Redis has a built-in mechanism to update the master-slave configuration using Redis Sentinels.

The final replication strategy of the Redis component can be seen in Figure 4.3.

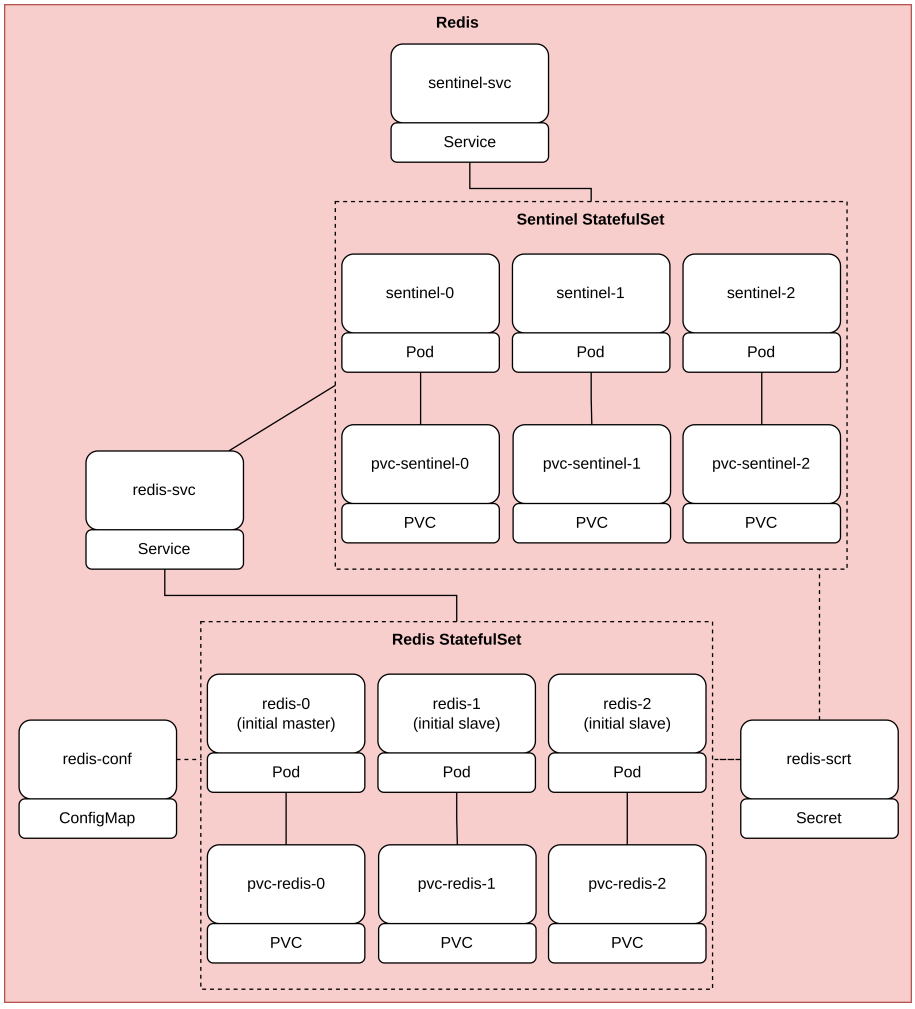

Figure 4.3: Redis Architecture

To test the adopted replication strategy for Redis, after successfully deploying all the resources, we killed the master Redis pod and checked if one of the slave pods was promoted to master. As we can see in Figure 4.4, in the beginning, the master is pod redis-0 (left terminal) and the slaves

are pods redis-1 and redis-2 (middle and right terminals). After killing the master, from Figure 4.5 we can see that the pod redis-2 was promoted to master and the pod redis-0 was promoted to slave.

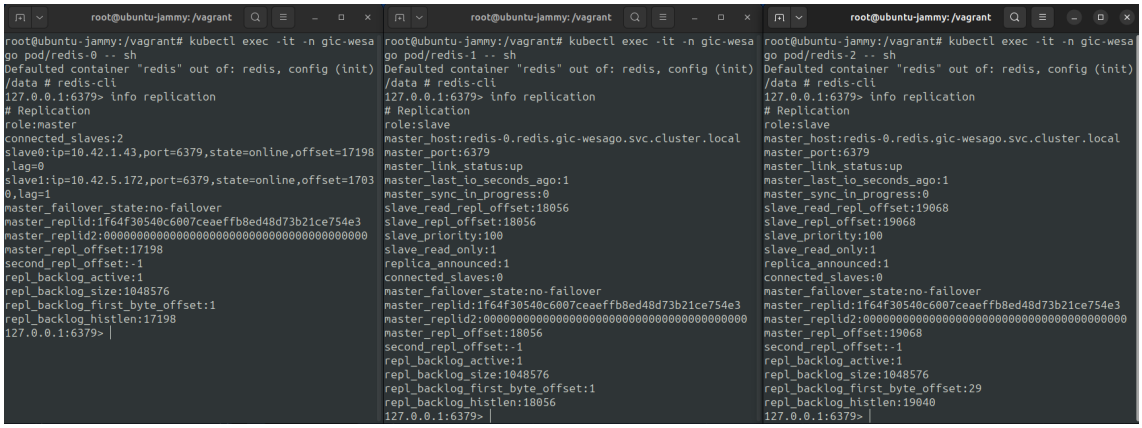

Figure 4.4: Redis pods before kill: left is the master

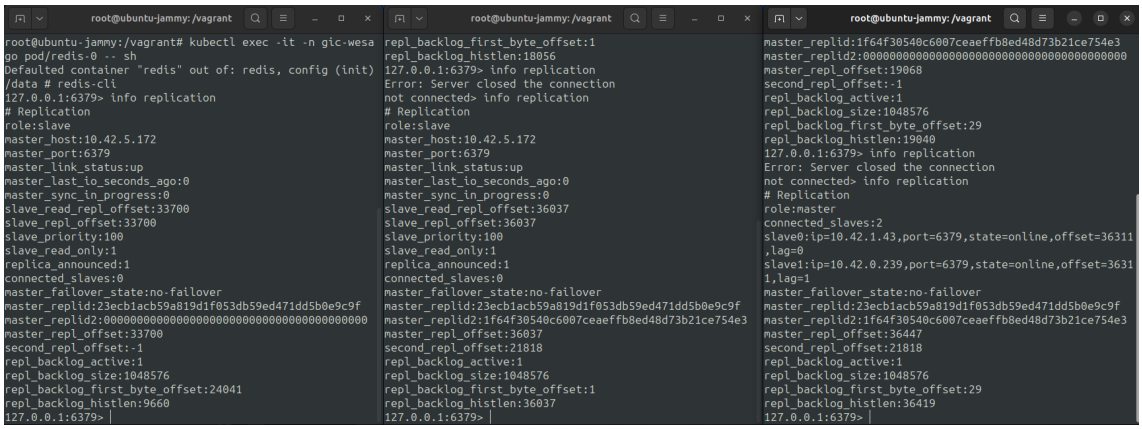

Figure 4.5: Redis pods after kill: right is the new master

Figure Figure 4.6 shows the data is being replicated across the Redis instances.

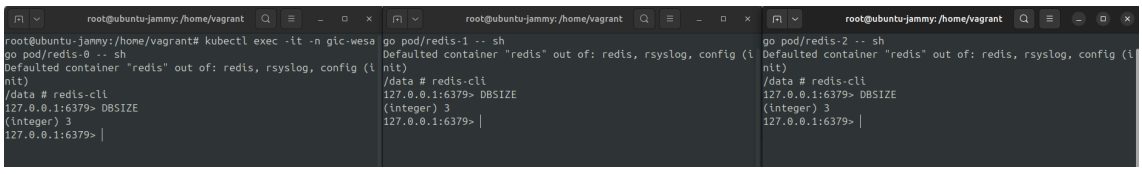

Figure 4.6: Redis Data Replication

4.2 Pod Distribution

Instance replication is not enough to ensure redundancy, as all instances of the same component can be running on the same node, if that node fails, all instances will be lost. To improve redundancy, we need to distribute the pods across different nodes in the cluster. This can be achieved by

using the Affinity feature in Kubernetes, which allows us to specify rules for how pods should be distributed across nodes. In our case, we want to ensure that pods of the same component are not running on the same node. This can be done by adding a PodAntiAffinity rule to the Deployment or StatefulSet configuration.

In Figure 4.7 we can see an example where we have 3 pods of the same component running on 3 different nodes. If one of the nodes fails, the other pods will still be running, ensuring that the application remains available.

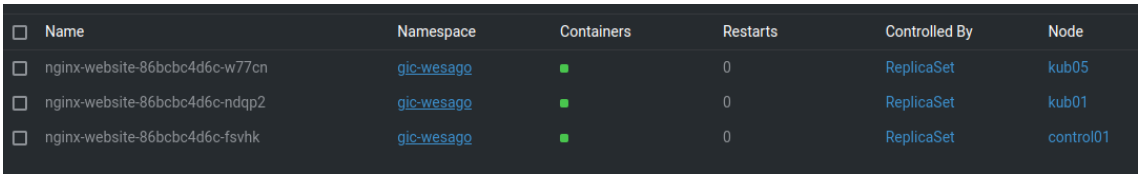

Figure 4.7: Pod Distribution with Affinity

Chapter 5

Autoscaling

Autoscaling is a critical feature for managing resources efficiently in a Kubernetes cluster. It enables the system to automatically adjust the number of running instances of an application based on the current load. This not only helps in maintaining optimal performance during high demand but also reduces costs by scaling down resources during periods of low usage.

Kubernetes supports 2 types of autoscaling, each addressing different aspects of resource management:

- **Horizontal Pod Autoscaling (HPA):** This type adjusts the number of pod replicas of a deployment or statefulset based on observed CPU/memory utilization or other selected metrics.
- **Vertical Pod Autoscaling (VPA):** This adjusts the resource requests and limits of containers in the pods based on historical usage data.

For our project, we used Horizontal Pod Autoscaling to scale the number of pod replicas based on CPU utilization. Specifically, the minimum number of replicas was set to 2 (for availability reasons), the maximum number of replicas was set to 5, and the target CPU usage was set to 70%. This is a simple starting point and should be adjusted based on the application's performance, considering each component separately.

To show the autoscaling in action, let's consider the scenario for the static website with target CPU usage set to 10%. Initially, the website is deployed with 3 replicas, which are reduced to 2 replicas due to low traffic after some time. To simulate high traffic, we used a load testing tool named *Plow*[1] to generate a high number of requests to the website.

In Figure 5.1 we can see requests generated by Plow, which caused the CPU utilization to increase. As a result, the number of replicas was scaled up to 4 to handle the increased load as shown in Figure 5.2.

[1] https://github.com/six-ddc/plow

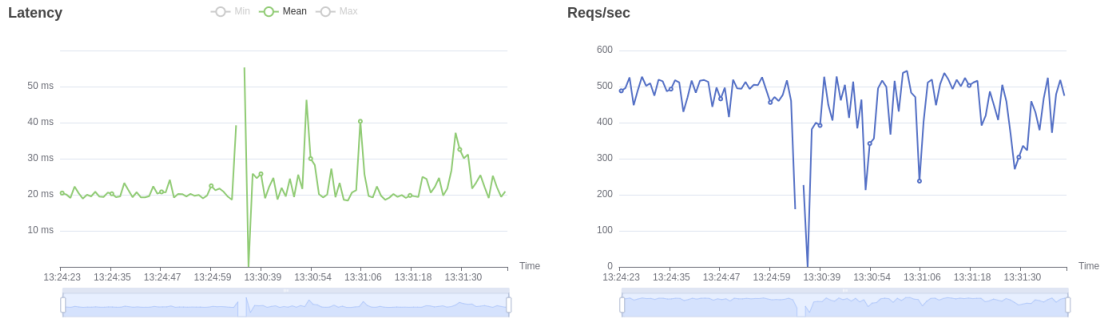

Figure 5.1: Plow load testing tool

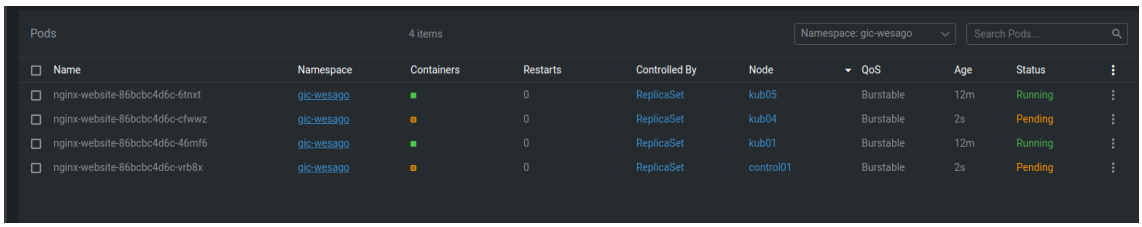

Figure 5.2: Autoscaling in action

If the load continues to increase, the number of replicas will be scaled up to the maximum number of replicas (5 in this case) to handle the increased load. Once the load decreases, the number of replicas will be scaled down to the minimum number of replicas (2 in this case) to reduce costs and resource usage.

Chapter 6

Monitoring and observability

When it comes to managing a complex system composed of multiple components, monitoring and observability are fundamental practices that must be adopted. Monitoring and observability are two distinct concepts, but they are often used interchangeably. Monitoring is the process of collecting data about the system and its components, while observability is the ability to understand the system based on the data collected.

In this project, in order to understand the internal state of the system and the behavior of each component, we focused our attention on collecting logs and metrics. Based on this collected data, we can identify potential issues, troubleshoot problems, and optimize the system.

6.1 Log collection

Logs include application and system-specific data that details the operations and flow of control within a system. Log entries describe events, such as starting a process, handling an error, or simply completing some part of a workload. Logging complements metrics by providing context for the state of an application when metrics are captured.

In this project, as we have multiple pods, each with its own containers, managing logs from all these containers can be challenging. If we need to debug an issue, we would have to check the logs of each container, which can be time-consuming and inefficient. To overcome this problem, we have centralized the logs in a single server using Rsyslog.

Given the flexibility of Rsyslog, we could adopt several strategies to centralize logs. For this project, we chose to use a sidecar container running a Rsyslog agent in each pod that listen and sends the logs to a central Rsyslog server via UDP or TCP. The main container and the Rsyslog agent share a volume, allowing the main container to write logs and the agent to send them to the server. The server receives the logs and writes them to a file. If there are multiple replicas of the same component, the server aggregates the logs from all replicas into a single file. For example, if we have 4 pods running the Django application, the server will aggregate the logs from all 4 pods into a single file named "django.log".

To better understand the log centralization process, the diagram in Figure 6.1 shows the relationship between the main container, the Rsyslog agent, and the central server, highlighting the communication channels and the data flow.

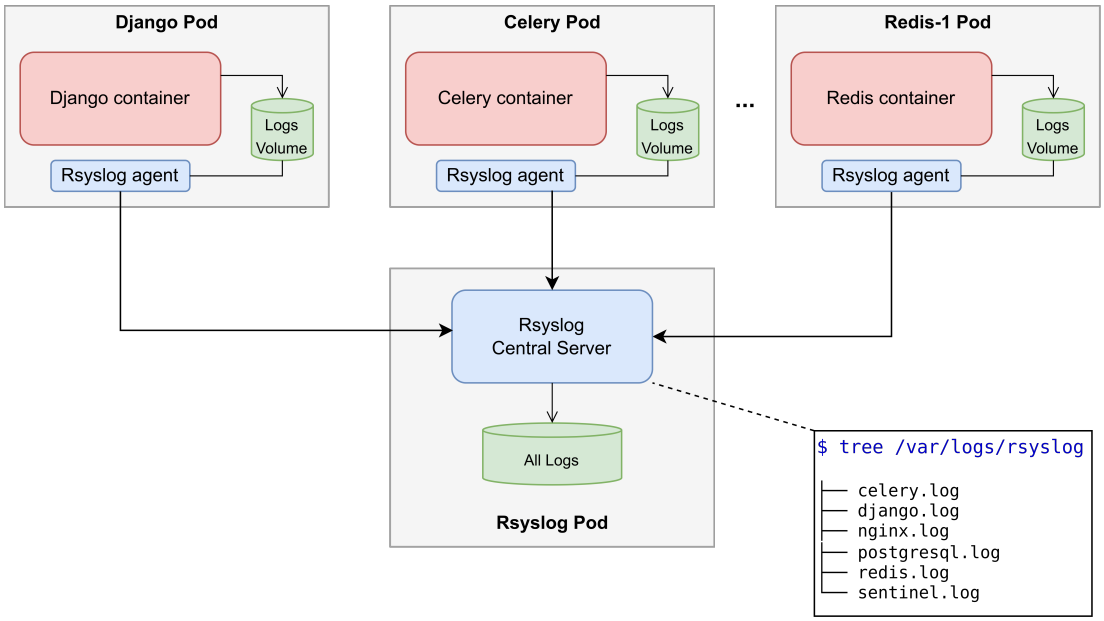

Figure 6.1: Logs centralization strategy

Whenever the Rsyslog central server receives new log entries, it adds a timestamp of when the log was received, the hostname that created the log, the application name, and the log message itself. This format follows the RFC 5424 standard, which is a standard for syslog messages. For exemplification, Figure 6.2 shows a log entry of the PostgreSQL container already stored in the central server.

Figure 6.2: Example of a log entry in the Rsyslog server

Even though Rsyslog provides a way to centralize logs, there are some challenges in standardizing logs. For instance, the logs generated by different applications may have different formats. In our case, the logs generated by Django are different from the logs generated by Redis because Redis does not allow changing the log format. This inconsistency can make it difficult to parse and analyze logs, but the beginning of each log in the server is the same because it is independent of the application that generated it.

6.2 Metric collection and visualization

Metrics are quantitative data that describe the state of a system at a specific point in time. Metrics are essential for monitoring the health and performance of a running system. In this project, we focused on collecting metrics to understand the behavior of the system and its components.

To collect metrics, we initially started by configuring Prometheus from scratch, however, some interesting metrics provided by the kube-state-metrics service were not accessible due to permission restrictions. To overcome this limitation, we decided to use the Prometheus server that was already running in the default namespace of the cluster which already collects a considerable number of metrics, including some specific ones provided by the kube-state-metrics agent.

Even though we have access to a wide range of metrics, from the most general to the most specific, we focused on collecting metrics that best reflect the state of the application. The metrics we used are the following:

- **kube_pod_container_status_running**: used to get the number of running containers of a given type which could indicate the number of pods of that type that are running.
- **kube_pod_container_status_restarts_total**: used to calculate the number of container restarts of a given type which could indicate the number of times a pod of that type has been restarted.
- **container_cpu_usage_seconds_total**: used to calculate the CPU usage over time.
- **container_memory_usage_bytes**: used to calculate the memory usage of a container.
- **container_network_receive_bytes_total**: used to calculate the total number of bytes received by a container.
- **container_network_transmit_bytes_total**: used to calculate the total number of bytes transmitted by a container.
- **container_fs_reads_bytes_total**: used to calculate the total number of bytes read by a container from the filesystem.
- **container_fs_writes_bytes_total**: used to calculate the total number of bytes written by a container to the filesystem.
- **kubelet_volume_stats_available_bytes**: used to calculate the available storage space in a volume.
- **kubelet_volume_stats_capacity_bytes**: used to calculate the total storage capacity of a volume.

Although the mentioned metrics provide good insights into the system's behavior and performance, we applied several operations on them to make them more meaningful. Some of these

operations include calculating the average, sum, and rate of the metrics over time. To get measures for a specific component, we also used filters in the Prometheus queries.

To visualize the metrics in a more appealing way and monitor them over time, we created different dashboards using Grafana. We created a dashboard for each component of the system and an additional dashboard to monitor the volumes. Each dashboard provides a comprehensive view of the system's performance and health, allowing us to quickly identify potential issues and take corrective actions.

For each component, the dashboards present:

- The number of pods running
- The number of pods restarted
- CPU usage by each container over time
- Memory usage by each container over time
- Network traffic received and transmitted by each container over time
- Filesystem reads and writes by each container over time

The volumes dashboard presents the percentage of used storage space for application and database volumes, and logs volumes. This information is crucial to prevent storage space issues and ensure the system's availability.

Figures 6.3 to 6.7 illustrate the developed Grafana dashboards.

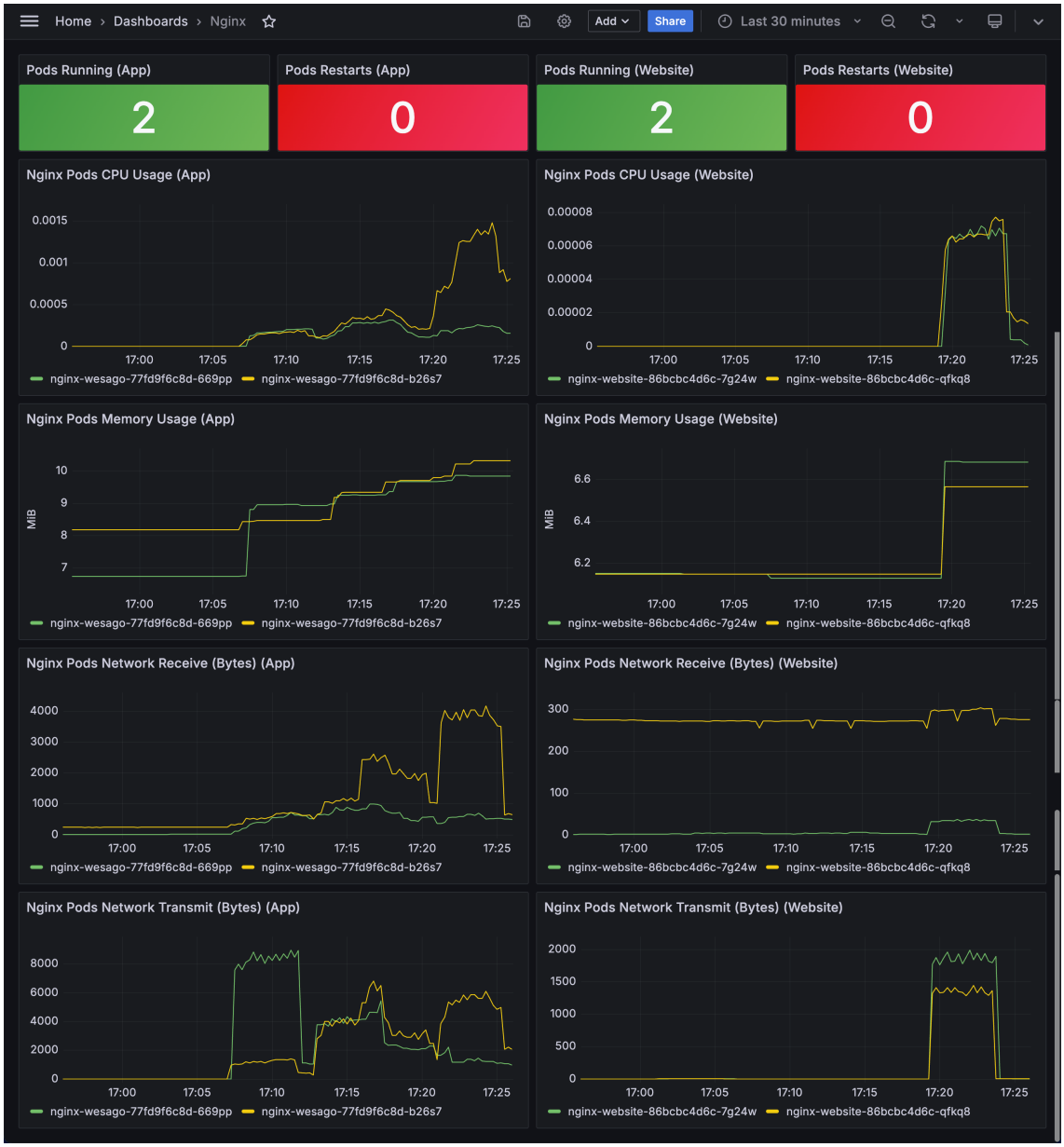

Figure 6.3: Grafana dashboard for Nginx

Figure 6.4: Grafana dashboard for Django

Figure 6.5: Grafana dashboard for Postgres

Figure 6.6: Grafana dashboard for Redis

Figure 6.7: Grafana dashboard for Persistent Volumes

Chapter 7

Disaster Recovery

A robust disaster recovery plan is essential for ensuring the availability and reliability of the application in the event of catastrophic failures. The goal of this plan is to minimize downtime and data loss while enabling quick recovery to normal operations. **This chapter outlines the disaster recovery strategies and procedures for the application, particularly focusing on the use of Longhorn for volume management, since Longhorn is not managed by us, we assume that is configured according to our requirements, including volume replication and backup**.

7.1 Disaster Recovery Strategies

There are 2 main objectives for the disaster recovery plan:

- **Recovery Point Objective (RPO):** The maximum acceptable amount of data loss measured in time. In this case is set to 15 minutes, meaning the system should be able to recover data from up to 15 minutes before the disaster. This time is an initial guess and should be adjusted according to traffic and the amount of data generated.
- **Recovery Time Objective (RTO):** The maximum acceptable amount of time to recover the system after a disaster. In this case is set to 1 hour, meaning the system should be able to recover and be operational within 1 hour after the disaster.

To backup data, we can use Longhorn's built-in backup functionality to take regular snapshots of Persistent Volumes. These snapshots are then stored in a remote backup location (e.g., AWS S3). These snapshots are taken every 15 minutes to ensure that the RPO is met.

The backup's integrity should be verified regularly to ensure that it can be restored in case of a disaster. This can be done by restoring the backup to a separate environment and running tests to ensure that the data is consistent and accurate.

7.2 Disaster Recovery Plan

Before a disaster occurs, it is important to have a well-defined disaster recovery plan in place to ensure that the system can be quickly and efficiently recovered. But first, we need to identify the disaster and assess the impact on the system. This can be done by using monitoring tools to detect anomalies and alert the team in case of a disaster, regular health checks, and audits to identify potential vulnerabilities and risks.

Then the Disaster Recovery Plan can be executed, which consists of the following steps:

- **Infrastructure Recovery:** The first step is to recover the infrastructure, including the Kubernetes cluster and configurations.
- **Deployment Recovery:** The next step is to recover the application deployment, including the necessary Kubernetes resources and configurations, which can be easily done using the automatic deployment process mentioned in chapter 2.
- **Data Recovery:** Then the data can be recovered using the snapshots taken by Longhorn.
- **Testing:** After the recovery is complete, the system should be tested to ensure that it is functioning correctly and that the data is consistent and accurate.

By following this disaster recovery plan, the system should be able to recover quickly and efficiently in the event of a disaster, ensuring minimal downtime and data loss.

After the disaster recovery plan is executed, it is important to analyze the root cause of the disaster, take steps to prevent it from happening again in the future and iterate on the disaster recovery plan to improve its effectiveness.

Then the availability of the system should be measured by

$$Availability = \frac{MTBF}{MTBF + MTTR} \tag{7.1}$$

where MTBF is the Mean Time Between Failures and MTTR is the Mean Time To Recovery. The goal is to maximize the availability of the system by minimizing the MTTR and increasing the MTBF.

Implementing a comprehensive disaster recovery plan is crucial for ensuring the availability and reliability of the application in the event of catastrophic failures. By following the strategies and procedures outlined in this chapter, the system should be able to recover quickly and efficiently, ensuring minimal downtime and data loss.

Chapter 8

Conclusion

In the first part of this project, we developed a deployment strategy for the Wesago application in a Kubernetes cluster. Although the strategy was functional and modular, allowing easy deployment of components, some reliability gaps remained. In this second part, we focused on enhancing reliability by implementing redundancy, health checks, autoscaling, monitoring, and disaster recovery strategies.

Redundancy techniques, such as instance replication and pod distribution, ensured the application remained available during failures. Multiple instances and distributed pods across nodes improved system reliability. Health checks, with liveness probes for each component, ensured the application stayed responsive and recovered from failures.

Autoscaling efficiently managed cluster resources by adjusting the number of running instances based on current load, maintaining optimal performance during high demand. Monitoring involved centralizing logs and visualizing metrics in Grafana dashboards, allowing quick issue identification and a long-term view of system performance.

We also explored disaster recovery strategies to ensure quick recovery from catastrophic failures, defining a plan to meet Recovery Time and Point Objectives, minimizing downtime and data loss.

While the strategies improved system reliability, further enhancements are possible, such as advanced monitoring with alerts and testing the disaster recovery plan.

Overall, the work done in this project was essential to improve the system's reliability and ensure that the application remains available and reliable in the event of failures.