DETI - 2023/2024

Initial Deployment

Report 1

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

Gestão de Infraestruturas de Computação

Professor: João Paulo Barraca

April 27, 2024

Contents

Chapter 1

Introduction

1.1 Scope

In the context of "Gestão de Infraestruturas de Computação" course, as part of the Master's degree in Computer Science and Engineering at the University of Aveiro, the students were challenged to deploy an open-source software and a static website in a cluster environment. The main goal is to choose a third-party software composed of multiple components, each one with its own requirements and dependencies, and deploy it in a cluster environment using Kubernetes. In order to accomplish this task, first we must understand the application at a deep level, and then design a deployment strategy that takes into account the requirements and limitations of the application and the cluster environment.

1.2 The chosen software

The open-source software to be chosen for this project had to meet the following requirements:

- Provides a service
- Operates as a redundant cluster
- Must have at least 4 components
- Must be web-based
- Must have a database
- Must have a cache
- Must run on Linux

The software that we elected for this project that meets all the requirements is Misago[1]. Misago is a web application that provides a general-purpose forum service. It implements several forum-related features such as public and private threads, posts, pools, following system, and

[1] https://github.com/rafalp/Misago

many others. To accomplish this, Misago is composed of 5 different components, each one with a specific responsibility. The components and their respective roles are:

- **Django**: The framework used to build the software. It is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.
- **PostgreSQL**: The database used to store the data. It is a powerful, open-source object-relational database system.
- **Celery**: Is used to handle asynchronous tasks. It is a distributed task queue that is used to handle a vast number of messages or tasks. It can be scaled horizontally to handle a large number of messages.
- **Redis**: Used as cache and message broker for Celery. It is an open-source, in-memory data structure store.
- **Nginx**: Is used as a reverse proxy to handle incoming HTTP requests and acts as a load balancer to distribute incoming traffic across multiple Django application instances.

Figure 1.1 shows the architecture of Misago and how the components interact with each other.

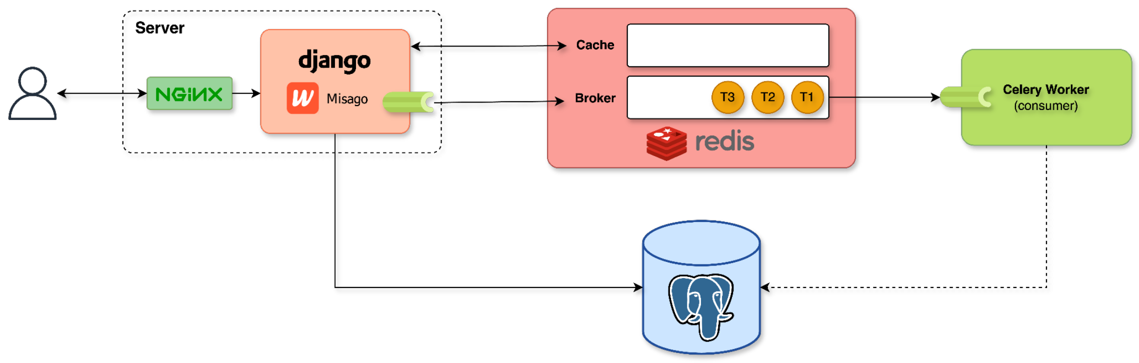


Figure 1.1: Misago Architecture

1.3 Our product

For this project, we named our product Wesago. Wesago is an adaptation of Misago software and is targeted to the University of Aveiro community. Wesago provides a forum service for the students, teachers, and staff of the University of Aveiro, where they can share knowledge, ask questions, and discuss topics related to the institution.

In terms of components, Wesago is composed of the same components as Misago, with the same roles. The only difference is that Wesago is customized to fit the University of Aveiro community. In addition to the application itself, we also have a static website that is meant to be the company/product front page. The website is a static page built with Docusaurus, which is a modern static website generator that uses the React framework as its core, making it easy to build a website with several features. The main components are:

- **React**: The framework used to build the website. It is a JavaScript library for building user interfaces.
- **Nginx**: Is used as a web server to serve the static files.

In the next sections, we are going to explain in detail the deployment strategy for Wesago and the static website. We are going to present and analyze the followed strategies and configurations, justify the choices made, and identify the limitations and bottlenecks of the deployment. Finally, we are going to evaluate the cluster operation and present the conclusions and future work.

Chapter 2

Deployment strategy

Deploying an already existing application, developed by a third-party developer, using Kubernetes is a challenging task and involves careful planning and consideration of various factors in order to ensure that the application runs efficiently and without any issues. The most important factors to ensure the successful deployment are the understanding of the components and their functions, the comprehension of the interactions and dependencies between components and the understanding of the correct Kubernetes resources to use for each component.

As shown in the first presentation, and as it can be seen in the introduction 1, the Wesago application is composed of several components, being them: a PostgreSQL database for storing the data of the application, a Redis database for caching and message brokering, a Django application as the main application server, a composition of Celery workers for running tasks and a Nginx server for reverse proxying the requests to the application. Each component has its own specific function and interacts with the other components in order to provide the full functionality of the application. To increase the complexity of the deployment, joining the Wesago application, we also have to take into consideration the deployment of a static website, this time developed by us, that is composed of a React application used to generate the website pages and a Nginx server used to serve the static files of the website.

These components interact with each other in order to provide the full functionality of the application. These interactions can lead to the existence of dependencies between components, meaning that one component cannot function without the other. In our case, we found two groups of component dependencies: the first group is composed of the Django application that cannot execute properly without the PostgreSQL database and the Redis database, and the second group is composed of the Celery workers that cannot run well without the PostgreSQL database and the Redis message broker. This means that in our deployment strategy, we need to take special care of these dependencies and ensure that the components are deployed in the correct order to minimize the risk of failure.

Kubernetes provides a wide range of resources that can be used to deploy applications, each with its own specific function and use case. For this deployment, we thought of using the following resources: Deployments, StatefulSets, Pods, Services, ConfigMaps, Secrets, PersistentVolume, and Ingresses. Each resource has its specific function and is self-explained by its name, but it is important to distinguish between why and when we used Deployments or StatefulSets. In Kubernetes, deployments are usually used for stateless applications meaning that the application does not store any data and can be easily replaced by another instance of the application, while statefulSets, on the other hand, are mostly used for stateful applications, meaning that the application needs to store data and needs to be replaced by the same instance of the application, without losing it's state. In our case, we found that Deployments would be the solution for the Django application, the Celery workers and the Nginx proxy, and StatefulSets would be used for databases such as the PostgreSQL database and the Redis database.

Figure 2.1, shows the our initial deployment strategy that we tried to follow while deploying all of the components of the application.

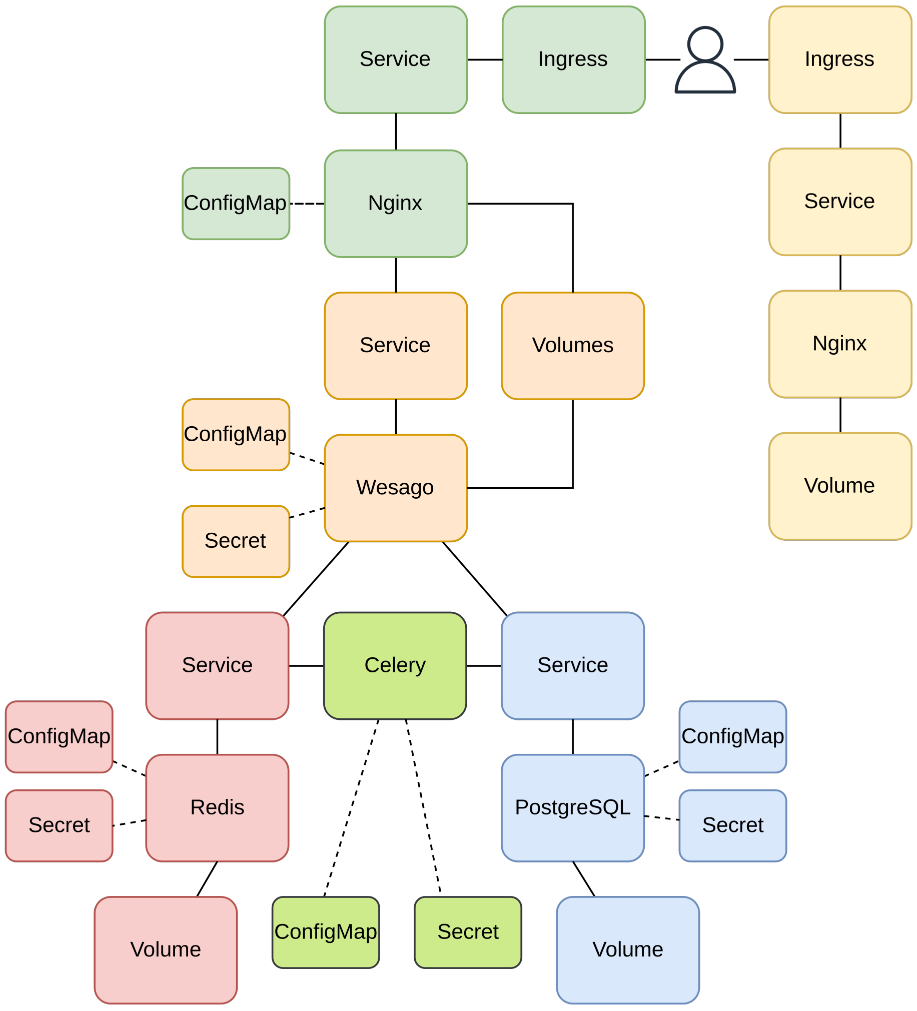


Figure 2.1: Initial Deployment Strategy (each colour refers to a different component).

In the next sections, we will describe a more in-depth view of the strategies used to deploy each component of the application and the static website.

2.1 Wesago

2.1.1 PostgreSQL

The PostgreSQL database is the main data storage component of the Wesago application. It is 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. The usage of a Kubernetes statefulset is almost mandatory in this case since, as previously seen, it provides stable and unique identifiers for the pods and allows persistent storage that might be useful for failure recovery.

With this in mind, we had to make some key decisions and have special care regarding the deployment of the PostgreSQL database. We thought of three possible approaches for the PostgreSQL deployment:

- **Using a statefulset with a single replica:** This approach would be the simplest one to implement and in case of a failure of the database we could recover the data using the persistent volume if it was not lost.
- **Using a statefulset with multiple replicas and a fixed master-slave configuration:** This approach requires a bit more configuration in the implementation but provides a more scalable solution since while data writing can only be done in the master replica, data reading can also be done in the slave databases.
- **Using a statefulset with multiple replicas and 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. 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 second method of using a statefulset with multiple replicas and a fixed master-slave configuration. This approach provides better scalability of the database and is the only possible replication approach that does not require the usage of operators, which was one of the limitations of the project.

For this deployment we used the following resources: a StatefulSet with 3 replicas, a headless service to allow the communication to specific pods, a PersistentVolumeClaim for each replica, a ConfigMap to store the configuration files and a Secret to keep the credentials of the database.

The final deployment strategy of the PostgreSQL database can be seen in figure 2.2.

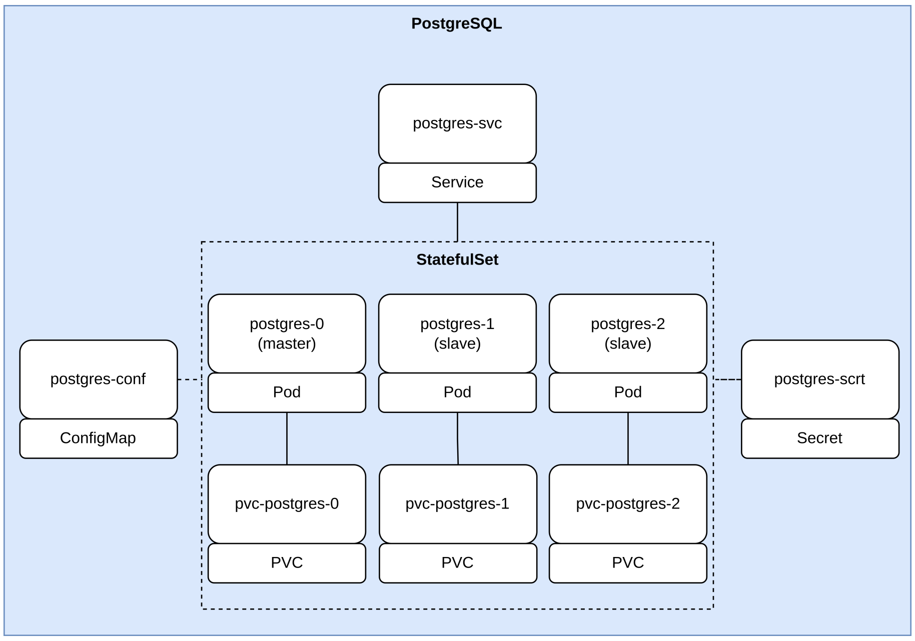


Figure 2.2: PostgreSQL Architecture

2.1.2 Redis

The Redis component has two main functionalities in the Wesago application: the cache function and the message broker function. If it was just for the cache, 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 a failure.

With this in mind, we had to, once again, make some decisions and have special care regarding the deployment of Redis. We thought of three possible approaches for the deployment:

- **Using a deployment with a single replica:** This approach would be the simplest one to implement and in case of a failure of the Redis component we could recover the data using the persistent volume if it was not lost.
- **Using a statefulset with multiple replicas and a fixed master-slave configuration:** This approach requires a bit more configuration in the implementation but provides a more scalable solution since while data writing can only be done in the master replica, data reading can also be done in the slave databases, but in case of a master failure, the system would not be fully available.
- **Using a statefulset with multiple replicas and a dynamic master-slave configuration updated by redis sentinels:** This approach is more complex to implement but would provide some more guarantees in case of failure. In this approach, redis would have a master that

would be allowed to write and read data and multiple slaves who would be responsible for reading the data. 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, this time we decided to use the third approach. This approach joins both high scalability and high availability of the Redis component and, this time is possible to be used since Redis has a built-in mechanism to update the master-slave configuration using Redis Sentinels.

For this deployment we used the following resources: for Redis, a StatefulSet with 3 replicas, a headless service to allow communication to specific pods, a PersistentVolume for each replica, a ConfigMap to store the configuration files and a Secret to keep the credentials; for the Redis sentinels we used a StatefulSet with 3 replicas, a service, a PersistentVolume for each replica for storing configurations and the same Secret used for Redis.

The final deployment strategy of the Redis database can be seen in figure 2.3.

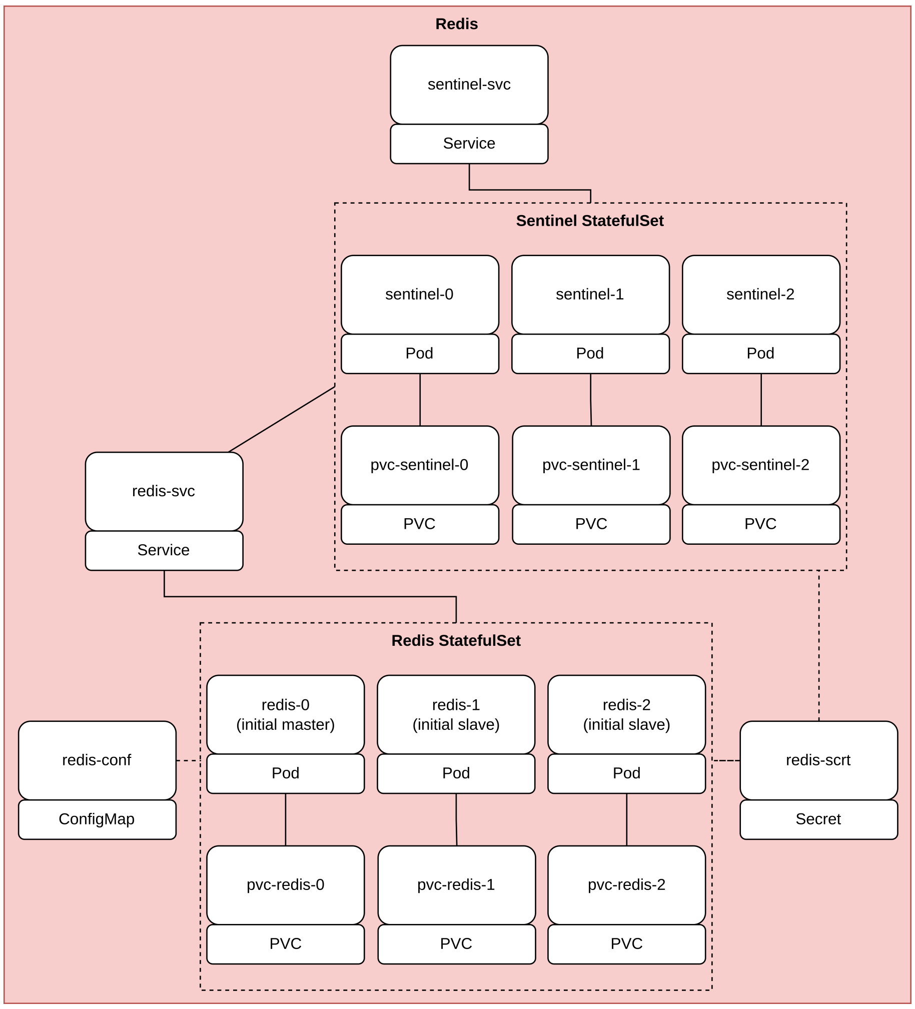


Figure 2.3: Redis Architecture

2.1.3 Django

The core component of Wesago is the Django application. This component implements all the business logic of Wesago and makes it available by exposing a REST API. In addition to implementing the business logic, it also defines the frontend of the application that will be served to the users.

To successfully deploy the Django application, we first created a Docker image with the source code and all the necessary dependencies. The Docker image is based on the official Python image and installs all the necessary dependencies using the pip package manager. As we are deploying the application with a "Production" mindset, instead of running the Django app with the development server, we are using the Gunicorn WSGI server to serve the application.

In terms of configuration, the Django app depends on several configuration data that can be divided into two groups: sensitive data and non-sensitive data. The sensitive data includes the Django secret key, the database host, username, password, and name, and the Redis host and port. On the other hand, non-sensitive configuration data includes language settings, timezone settings, and static and media file settings. To provide these configurations to the Django application, we used a Kubernetes ConfigMap for non-sensitive data and a Kubernetes Secret for sensitive data.

To store all the static content that will be served, and all the media files uploaded by the users, we used 2 PersistentVolumeClaims, one for the static files and one for the media files. We have a third PersistentVolumeClaim exclusively dedicated to a gallery of avatars that already comes with the original Misago software.

Since the Django application is stateless, we use a Kubernetes Deployment resource that uses the previously created Docker image, the ConfigMap and Secret. It also mounts the PersistentVolumeClaims to the correct directories inside the container. The Deployment resource defines the number of replicas that will be running at the same time, in our case, we defined 3 replicas to ensure availability. To expose the Django application, we used a Kubernetes Service resource that will be used by the Nginx proxy.

Finally, we also created 3 Kubernetes Jobs that are used to run the initial setup of the Django application.

- ***migrations-job.yaml***: runs the python manage.py migrate command to create the database schema.
- ***collectstatic-job.yaml***: runs the python manage.py collectstatic command to collect all the static files in the static volume.
- ***create-superuser-job.yaml***: runs the python manage.py createsuperuser command to create the superuser that will be used to access the admin page.

2.1.4 Nginx

To avoid exposing the Django application directly to the internet, we used an Nginx server as a reverse proxy. The Nginx server is responsible for receiving the incoming HTTP requests and forwarding them to the Django application using its Kubernetes Service name. The Nginx server also serves the static and media files that are stored in the persistent volumes, meaning that both the Django application and the Nginx server need to share the same persistent volumes.

Similar to the Django application, the Nginx server is also deployed using a Kubernetes Deployment resource as it is stateless. To make it available to the internet, we first created a Service resource that exposes the Nginx server on port 80, and then an Ingress resource that forwards the incoming requests to that service. The Ingress resource also defines the hostname "wesago.k3s" that will be used to access the application.

2.1.5 Celery

To improve the overall performance of the application, the system uses Celery to handle asynchronous tasks, including notifications, database deletions, and other tasks that are not critical to the user experience and do not need to be executed immediately. As Celery is a distributed task queue it can be scaled horizontally to handle a large number of messages.

In terms of deployment, the Misago authors facilitated the deployment of Celery workers by incorporating the worker's tasks into the Django application source code. This way we could reuse the same Docker image that we used for the Django application to deploy the Celery workers. The only difference is that we need to run the Celery worker command instead of the Gunicorn server command. Similar to the Django application, for the deployment we used a Deployment resource, a ConfigMap, and a Secret. As the Celery workers only need to access the PostgreSQL database and the Redis message broker, we do not need to mount any persistent volumes, run Jobs or create Services.

2.2 Product website

In Figure 2.4 we can see the deployment strategy for the static website.

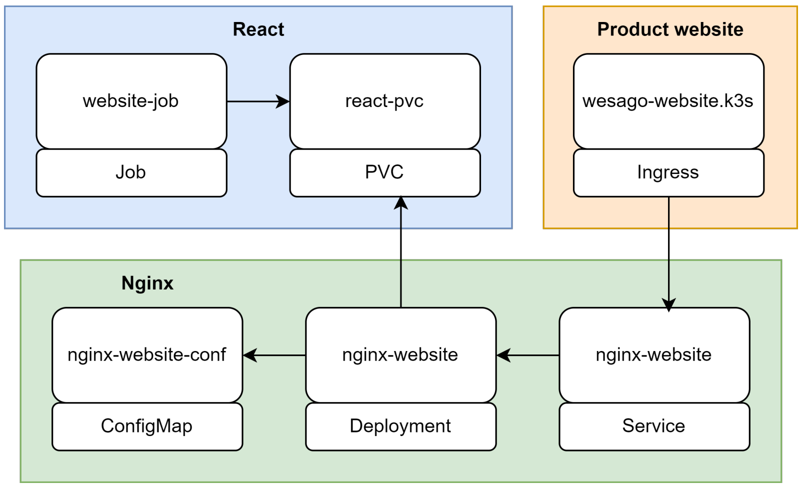


Figure 2.4: Website Deployment Strategy

2.2.1 React

As mentioned previously, the product page is a static website, so it does not need a container or a service to run. Only needs a volume to store the files. To achieve this, we start by creating a PersistentVolumeClaim where the files will be stored. Then, we create a docker image that contains the source code of the website, install the dependencies and build the website. Finally, we create a Job with the docker image and with the volume mounted to it, which is responsible for copying the files to the volume.

2.2.2 Nginx

Nginx is used as a web server to serve static files. We create a Deployment with the Nginx image and mount the volume created previously to it, this deployment has 3 replicas to ensure high availability. To change configurations easily, we create a ConfigMap with the Nginx configuration file and mount it to the Nginx deployment. Then, we create a Service to expose the Nginx deployment to an Ingress. An alternative to this approach would be to put the configuration file and the static files in the docker image for the Nginx deployment. However, this approach would make it harder to change configurations and update the website.

2.2.3 Website Ingress

An Ingress is used to expose the website to the internet, with the domain wesago-website.k3s[1]. We use Traefik as the Ingress controller. The Nginx service is used as the backend of the Ingress.

[1] `http://wesago-website.k3s`

Chapter 3

Limitations and Bottlenecks

Throughout the analysis and development of the deployment, we encountered certain constraints that we could not address due to factors beyond our control, or because the proposed solution was not entirely fault-tolerant. The identified constraints and bottlenecks include:

- **No volume replication:** Due to the limited storage of the cluster, persistent volumes only have one replica, meaning that in case of a failure of the node where the volume was stored or the volume itself, the data would be lost. This limitation was out of our control since it was a limitation of the given cluster itself.

- **No usage of operators:** Another limitation was the non-usage of operators to manage the databases. Using operators could improve the solution for some challenges related to database replication, however it was suggested to not use operators for this project.

- **No certainty of reading the most updated data:** Since we use slaves for reading data, there is no certainty that we are reading the most updated data as slaves might still be syncing with the master. This is a complex challenge that needs to be further explored.

- **Downtime in case of Redis master failure:** In case of a Redis master failure, there is still a downtime while a new master is elected. This limitation might cause some difficulties in the application execution as the first write after the master failure might take a bit longer to be executed.

- **Downtime in case of PostgreSQL master failure:** 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. This limitation might cause some difficulties in 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.

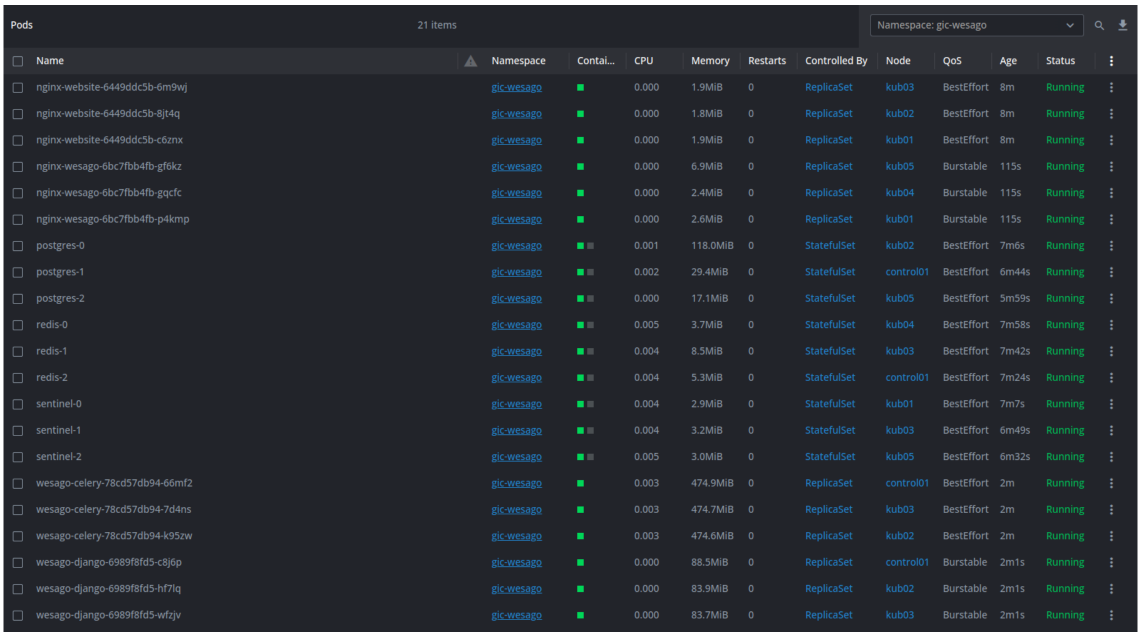
- **Already existing deployments for some components:** We found that there are already many libraries that implement some important points that we had to implement manually, for example helm charts for a faster configuration of the deployment. Considering the point of view, this might be a limitation since these libraries are much more robust and are heavily tested for production environments, having better availability and fault tolerance strategies. However, this limitation might be overshadowed by the fact that in security terms, we should not trust third-party libraries for production deployments unless very carefully analyzed and tested. Another reason to not use this type of libraries is the context of the project, where we were asked to develop from the ground the deployment.

In the second report, we expect to address some of these limitations and bottlenecks, providing solutions to mitigate them using new strategies that we will study later on.

Chapter 4

Cluster Evalutation

After creating all the Kubernetes resources mentioned before, all the components of the Wesago application and the static website were successfully deployed in the Kubernetes cluster.


Pods 21 items Namespace: gic-wesago

Name	Namespace	Contai...	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
nginx-website-6449ddc5b-6m9wj	gic-wesago	🟩	0.000	1.9MiB	0	ReplicaSet	kub03	BestEffort	8m	Running
nginx-website-6449ddc5b-8jt4q	gic-wesago	🟩	0.000	1.8MiB	0	ReplicaSet	kub02	BestEffort	8m	Running
nginx-website-6449ddc5b-c6znx	gic-wesago	🟩	0.000	1.9MiB	0	ReplicaSet	kub01	BestEffort	8m	Running
nginx-wesago-6bc7fbb4fb-gf6kz	gic-wesago	🟩	0.000	6.9MiB	0	ReplicaSet	kub05	Burstable	115s	Running
nginx-wesago-6bc7fbb4fb-gqcfc	gic-wesago	🟩	0.000	2.4MiB	0	ReplicaSet	kub04	Burstable	115s	Running
nginx-wesago-6bc7fbb4fb-p4kmp	gic-wesago	🟩	0.000	2.6MiB	0	ReplicaSet	kub01	Burstable	115s	Running
postgres-0	gic-wesago	🟩⬜	0.001	118.0MiB	0	StatefulSet	kub02	BestEffort	7m6s	Running
postgres-1	gic-wesago	🟩⬜	0.002	29.4MiB	0	StatefulSet	control01	BestEffort	6m44s	Running
postgres-2	gic-wesago	🟩⬜	0.000	17.1MiB	0	StatefulSet	kub05	BestEffort	5m59s	Running
redis-0	gic-wesago	🟩⬜	0.005	3.7MiB	0	StatefulSet	kub04	BestEffort	7m58s	Running
redis-1	gic-wesago	🟩⬜	0.004	8.5MiB	0	StatefulSet	kub03	BestEffort	7m42s	Running
redis-2	gic-wesago	🟩⬜	0.004	5.3MiB	0	StatefulSet	control01	BestEffort	7m24s	Running
sentinel-0	gic-wesago	🟩⬜	0.004	2.9MiB	0	StatefulSet	kub01	BestEffort	7m7s	Running
sentinel-1	gic-wesago	🟩⬜	0.004	3.2MiB	0	StatefulSet	kub03	BestEffort	6m49s	Running
sentinel-2	gic-wesago	🟩⬜	0.005	3.0MiB	0	StatefulSet	kub05	BestEffort	6m32s	Running
wesago-celery-78cd57db94-66mf2	gic-wesago	🟩	0.003	474.9MiB	0	ReplicaSet	control01	BestEffort	2m	Running
wesago-celery-78cd57db94-7d4ns	gic-wesago	🟩	0.003	474.7MiB	0	ReplicaSet	kub03	BestEffort	2m	Running
wesago-celery-78cd57db94-k95zw	gic-wesago	🟩	0.003	474.6MiB	0	ReplicaSet	kub02	BestEffort	2m	Running
wesago-django-6989f8fd5-c8j6p	gic-wesago	🟩	0.000	88.5MiB	0	ReplicaSet	control01	Burstable	2m1s	Running
wesago-django-6989f8fd5-hf7lq	gic-wesago	🟩	0.000	83.9MiB	0	ReplicaSet	kub02	Burstable	2m1s	Running
wesago-django-6989f8fd5-wfzjv	gic-wesago	🟩	0.000	83.7MiB	0	ReplicaSet	kub03	Burstable	2m1s	Running

Figure 4.1: All resources created successfully

When we access the Wesago application using the hostname "wesago.k3s", we are redirected to the main page of the application and we are able to navigate through the different sections. When creating a new thread, post, or pool, the application behaves as expected and the data is stored in the PostgreSQL databases. Whenever we upload images to the application, they are displayed correctly and analysing its URL we can see that they are stored in the media volume and served by the Nginx server.

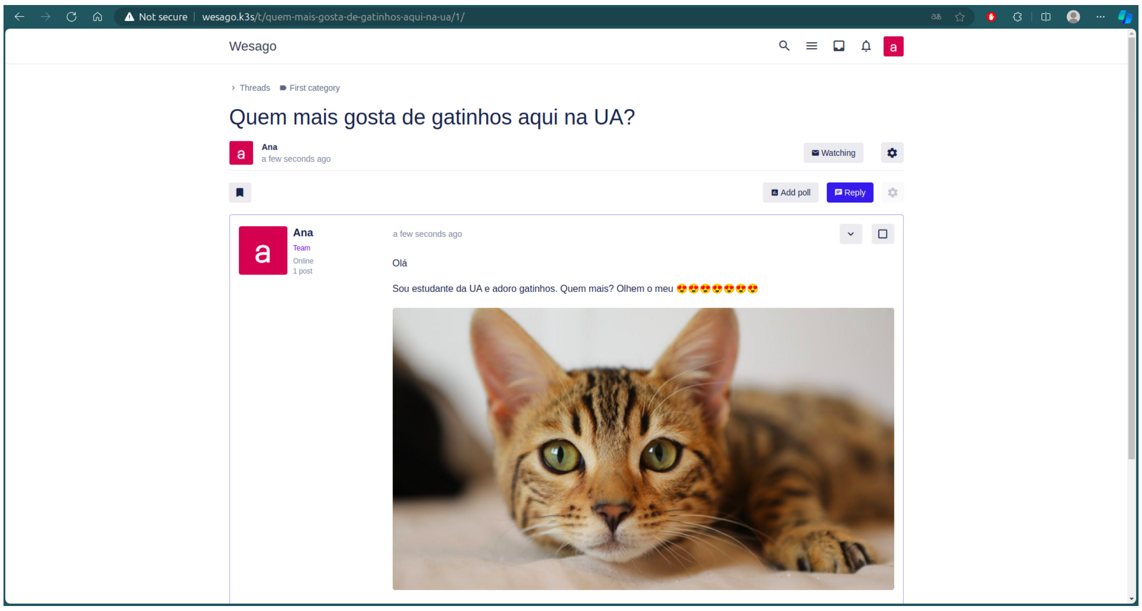

Not secure | wesago.k3s/t/quem-mais-gosta-de-gatinhos-aqui-na-ua/1/
Wesago
Threads
First category
Quem mais gosta de gatinhos aqui na UA?
Ana
a few seconds ago
Watching
Add poll
Reply
Ana
Team
Online
1 post
a few seconds ago
Olá
Sou estudante da UA e adoro gatinhos. Quem mais? Olhem o meu 😍😍😍😍😍😍😍

Figure 4.2: Page when accessing a thread on "wesago.k3s"

Furthermore, when we access the static website using the hostname "wesago-website.k3s", the page loads as expected with all components. We can conclude that the website is being correctly served by the Nginx server.

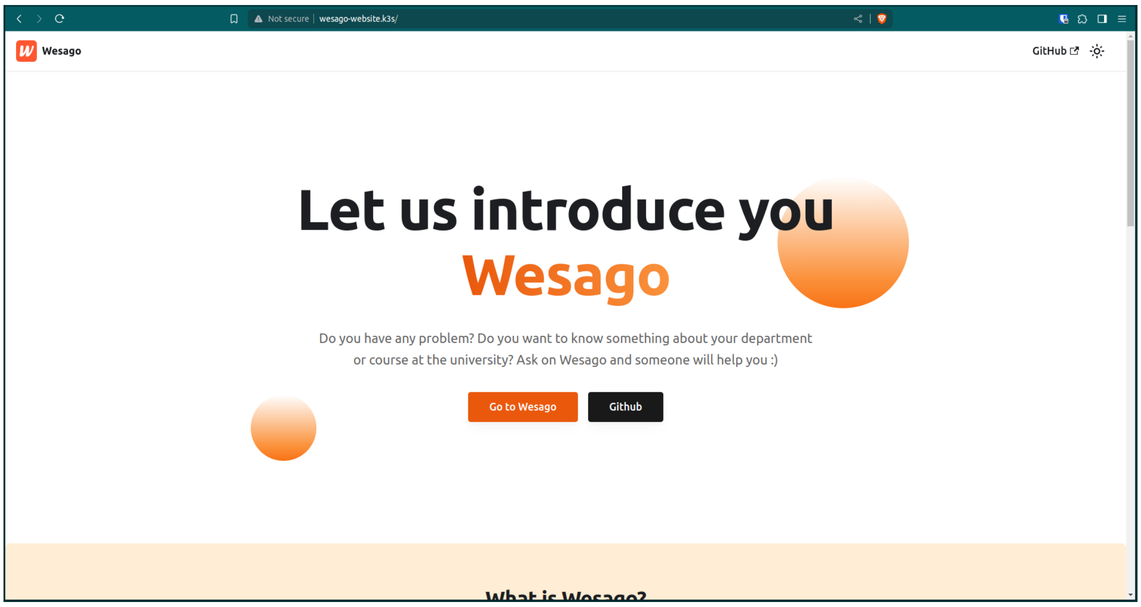
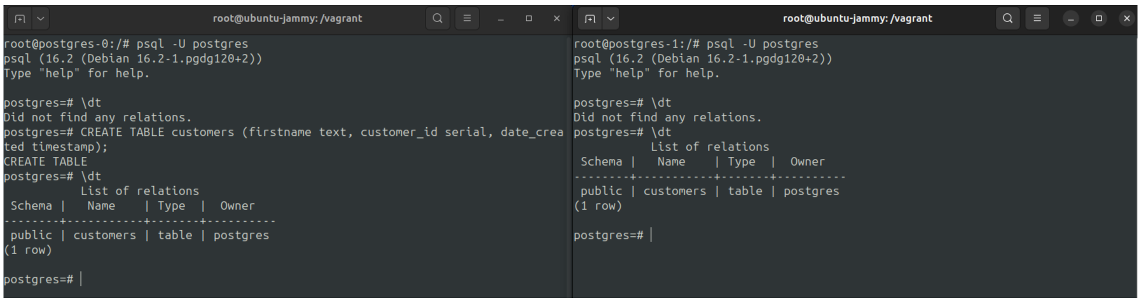

Not secure | wesago-website.k3s/
Wesago
GitHub
Let us introduce you
Wesago
Do you have any problem? Do you want to know something about your department
or course at the university? Ask on Wesago and someone will help you :)
Go to Wesago
Github
What is Wesago?

Figure 4.3: Main page when accessing "wesago-website.k3s"

In terms of database replication, as it is possible to see in Figure 4.4, the PostgreSQL is composed of a master (on the right terminal) and a slave (on the left terminal), and are working as expected. Initially, both the master and the slaves are syncronized 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. By having the same data, the slaves are a great option for allowing reads and improving the performance of the database and the whole system.



```
root@postgres-0:/# psql -U postgres
psql (16.2 (Debian 16.2-1.pgdg120+2))
Type "help" for help.

postgres=# \dt
Did not find any relations.
postgres=# CREATE TABLE customers (firstname text, customer_id serial, date_crea
ted timestamp);
CREATE TABLE
postgres=# \dt
           List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | customers | table | postgres
(1 row)

postgres=# 
```

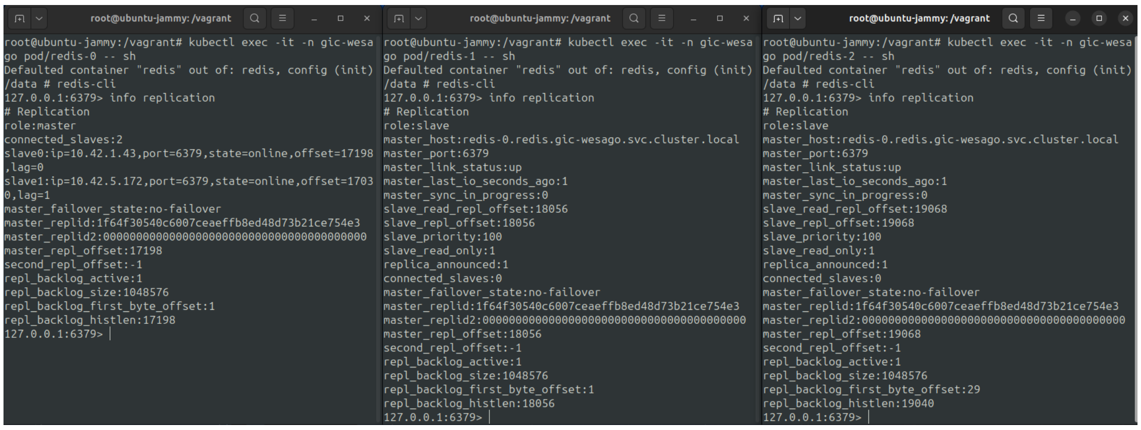
```
root@postgres-1:/# psql -U postgres
psql (16.2 (Debian 16.2-1.pgdg120+2))
Type "help" for help.

postgres=# \dt
Did not find any relations.
postgres=# \dt
           List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | customers | table | postgres
(1 row)

postgres=# 
```

Figure 4.4: PostgreSQL Replication

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 4.5, 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 4.6 we can see that the pod redis-2 was promoted to master and the pod redis-0 was promoted to slave. This means that the Redis replication strategy is working as expected.

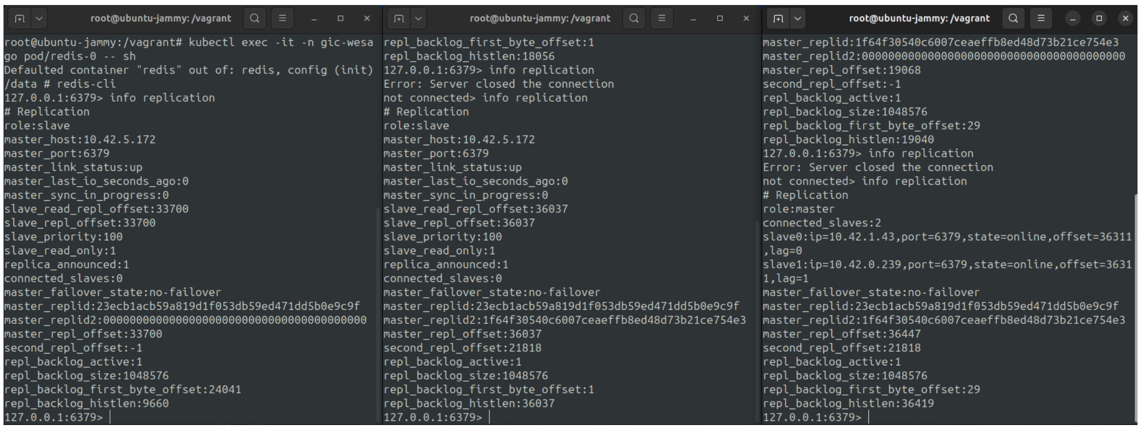

root@ubuntu-jammy: /vagrant
root@ubuntu-jammy: /vagrant
root@ubuntu-jammy: /vagrant

```
root@ubuntu-jammy:/vagrant# kubectl exec -it -n gic-wesa
go pod/redis-0 -- sh
Defaulted container "redis" out of: redis, config (init)
/data # redis-cli
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=10.42.1.43,port=6379,state=online,offset=17198
,lag=0
slave1:ip=10.42.5.172,port=6379,state=online,offset=1703
0,lag=1
master_failover_state:no-failover
master_replid:1f64f30540c6007ceaeffb8ed48d73b21ce754e3
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:17198
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:17198
127.0.0.1:6379> 
```

```
root@ubuntu-jammy:/vagrant# kubectl exec -it -n gic-wesa
go pod/redis-1 -- sh
Defaulted container "redis" out of: redis, config (init)
/data # redis-cli
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:redis-0.redis.gic-wesago.svc.cluster.local
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_read_repl_offset:18056
slave_repl_offset:18056
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:1f64f30540c6007ceaeffb8ed48d73b21ce754e3
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:18056
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:18056
127.0.0.1:6379> 
```

```
root@ubuntu-jammy:/vagrant# kubectl exec -it -n gic-wesa
go pod/redis-2 -- sh
Defaulted container "redis" out of: redis, config (init)
/data # redis-cli
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:redis-0.redis.gic-wesago.svc.cluster.local
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_read_repl_offset:19068
slave_repl_offset:19068
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:1f64f30540c6007ceaeffb8ed48d73b21ce754e3
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:19068
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:29
repl_backlog_histlen:19040
127.0.0.1:6379> 
```

Figure 4.5: Redis pods before kill: left is the master



```
root@ubuntu-jammy:/vagrant# kubectl exec -it -n gic-wesa
go pod/redis-0 -- sh
Defaulted container "redis" out of: redis, config (init)
/data # redis-cli
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:10.42.5.172
master_port:6379
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_read_repl_offset:33700
slave_repl_offset:33700
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:23ecb1acb59a819d1f053db59ed471dd5b0e9c9f
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:33700
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:24041
repl_backlog_histlen:9660
127.0.0.1:6379> 
```

```
repl_backlog_first_byte_offset:1
repl_backlog_histlen:18056
127.0.0.1:6379> info replication
Error: Server closed the connection
not connected> info replication
# Replication
role:slave
master_host:10.42.5.172
master_port:6379
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_read_repl_offset:36037
slave_repl_offset:36037
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:23ecb1acb59a819d1f053db59ed471dd5b0e9c9f
master_replid2:1f64f30540c6007ceaeffb8ed48d73b21ce754e3
master_repl_offset:36037
second_repl_offset:21818
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:36037
127.0.0.1:6379> 
```

```
master_replid:1f64f30540c6007ceaeffb8ed48d73b21ce754e3
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:19068
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:29
repl_backlog_histlen:19040
127.0.0.1:6379> info replication
Error: Server closed the connection
not connected> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=10.42.1.43,port=6379,state=online,offset=36311
,lag=0
slave1:ip=10.42.0.239,port=6379,state=online,offset=3631
1,lag=1
master_failover_state:no-failover
master_replid:23ecb1acb59a819d1f053db59ed471dd5b0e9c9f
master_replid2:1f64f30540c6007ceaeffb8ed48d73b21ce754e3
master_repl_offset:36447
second_repl_offset:21818
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:29
repl_backlog_histlen:36419
127.0.0.1:6379> 
```

Figure 4.6: Redis pods after kill: right is the new master

Chapter 5

Conclusion

In this project, 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 designed a deployment strategy that takes into account the requirements and limitations of the application and the cluster environment. In order to achieve the final goal, several Kubernetes resources were used including Deployments, StatefulSets, Services, ConfigMaps, Secrets, PersistentVolumeClaims, Ingresses, and Jobs. In addition, several replication strategies were also studied and implemented for the PostgreSQL and Redis databases.

Although the deployment was successful, several factors non-related to the system itself can be considered and will be our future work. These factors include:

- **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. This will help to reduce costs and improve performance.
- **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 system's redundancy can be further improved by deploying the pods across different nodes in the cluster with the use of node affinity. This will help to reduce the impact of node failures on 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.
- **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.

By addressing these future work areas, the system can be made more reliable, scalable, and cost-effective.