# DAT515 Project Report

Developing a Cloud-Based Dog Shelter Website Using Docker and Kubernetes

Katerina Kolarikova, Sofia Michailidu
University of Stavanger, Norway
k.kolarikova@stud.uis.no,s.michailidu@stud.uis.no

## ABSTRACT

The goal of this application was to create a dog shelter website using Docker and Kubernetes. This web page was successfully developed, allowing users to list and add new dogs for adoption while running as containerized application with robust cloud solutions. The primary focus of this project was to explore the potential for enhancing the application through the use of these technologies and to provide an overview of how employing these solutions leads to improved performance and enhanced features of the application.

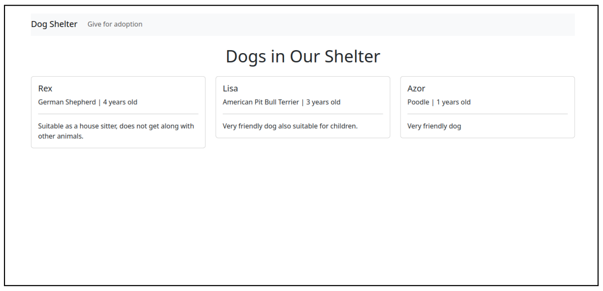
## KEYWORDS

Docker, Kubernetes, Cloud, Containerized Application, Flask, React, MySQL, Web Application

## 1 INTRODUCTION

Nowadays, applications are essential parts of our everyday lives, with thousands of different applications used daily on smartphones and computers. As the number of users continues to grow, it's increasingly important to consider the architecture of these applications to ensure they are robust, efficient, cost-effective, stable, and capable of handling large number of users. Additionally, they must provide security measures to protect user data. One possible approach to addressing these challenges is through containerized applications and cloud solutions.

The goal of this project is to explore containerized and cloud solutions and focus on integrating two main platforms, Docker[1, 8, 9] and Kubernetes[2, 6], into a web application. It also provides an overview of how these technologies can address the previously mentioned challenges. The resulting web application serves as an adoption page for a dog shelter, allowing users to view a list of dogs available for adoption and add new dogs. An example of the user interface can be seen in Figures 1 and 2.



Dogs in Our Shelter

Rex
German Shepherd | 4 years old
Suitable as a house sitter, does not get along with other animals.

Lisa
American Pit Bull Terrier | 3 years old
Very friendly dog also suitable for children.

Azor
Poodle | 1 years old
Very friendly dog

**Figure 1: List of Available Dogs**

*Cloud Computing Technologies (DAT515), UiS,*
2024.

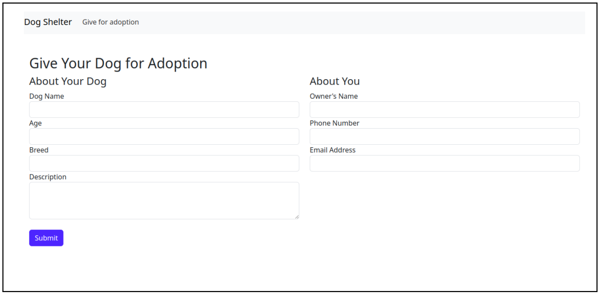


**Figure 2: Form example**

The following chapters provide a description of the process of development, focusing on the application's design, implementation and details of containerization and cloud deployment. Additionally, a basic introduction to the main technologies used in this project is included. The code created for this project is available at https://github.com/dat515-2024/KaSo.

## 2 DESIGN

This chapter provides a high-level system overview, emphasizing core technologies: Flask[3, 7], React[4] and MySQL[5] in cooperation with Docker and Kubernetes. Flask and React are selected due to their popularity in modern web development environments; in particular, Flask's minimalistic framework allows us to concentrate on cloud technology integration. Docker and Kubernetes provide containerization, component isolation, enhanced security and scalability, optimizing system performance and resilience.

To provide deeper understanding of how the developed application operates, comprehensive documentation is available. This documentation offers guidance on the application itself and includes description of each part of the code. This documentation is part of GitHub repository delivered together with this report.

### 2.1 System Architecture Overview

The application employs a three-tier architecture with the following components:

(1) **React Frontend:** Stateless frontend, built with HTML and React and served by NGINX[1], presents features for dog adoption, including showcasing available dogs and providing a form for submitting adoption requests. It relies on the Flask backend to fetch data on demand, ensuring no client data is stored within the browser.

[1]https://nginx.org/en/

(2) **Flask Backend (REST API):** Flask backend functions as a REST API, facilitating data flow between the MySQL database and the frontend. The backend functions as a stateful web server that utilizes database storage to maintain data.
(3) **MySQL Database:** MySQL serves as the persistent data store for the application, with data durability ensured by using a persistent volume and volume claim, providing secure and reliable long-term storage.

Figure 3 illustrates component interactions within the system architecture, highlighting the roles of Docker and Kubernetes. Each component uses a dedicated Docker container and is deployed via Kubernetes, allowing for modular management and interaction. The diagram also provides insight into the interactions between components, illustrating how data flows across the frontend, backend, and database layers. Additionally, we can see persistent volume and persistent volume claim which ensure reliable data storage across pods.

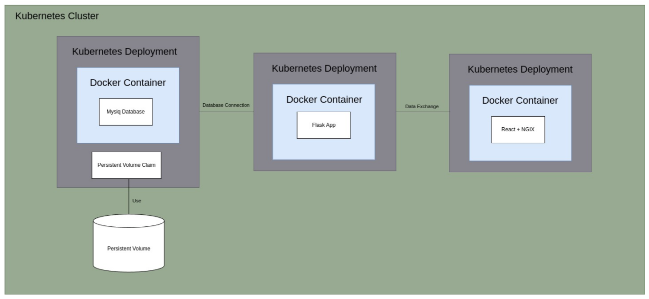


**Figure 3: System Architecture Overview**

## 2.2 Architecture advantages

This architecture offers numerous benefits, especially because of multicontainer architecture. Leveraging technologies such as Docker and Kubernetes significantly contributes to achieving higher levels of isolation by allowing the separation of environmental variables and dependencies for each component. Furthermore, it enhances modularity by enabling independent deployment of each part of the system. Additionally, Docker enhances portability, by eliminating issues often described with the phrase, “it works on my machine”.

This isolation also enhances fault tolerance within the system. Especially because of Kubernetes’ features like automatic pod restarts and many others, which help maintain system resilience.

Another advantage of this architecture is its increased scalability. Each component of the system has distinct scalability requirements; for instance, the frontend must accommodate varying user loads, while the backend needs resources to fulfil requests. By employing technologies like Kubernetes, we can independently scale each layer, depending on problem we would like to address, without impacting the others.

Furthermore, this architecture enables efficient resource management, which is especially advantageous in real-time environments where deployments are cloud-based, and costs might depend on resource utilization.

Lastly, security is significantly improved due to the ability to implement different security policies for each isolated component.

# 3 IMPLEMENTATION

This chapter discusses the implementation of the individual parts of the application, aiming to create a lightweight solution suitable for the use case of dog adoption and fostering.

## 3.1 Implementation of the React Frontend

The React framework provides the construction of websites using **React components**, which serve as individual building blocks of the application. This component-based design offers several advantages, including enhanced reusability and easier maintenance. Key components used in this project are:

- **App Component**: Primary application component, responsible for managing routing and rendering the navigation bar, which includes links to the homepage and the adoption form.
- **Homepage Component**: Component that fetches a list of dogs currently available for adoption from the backend API and displays their details using the DogCard component.
- **DogCard Component**: A reusable component that presents each dog’s name, breed, age, and description within a Bootstrap card layout.
- **Form Component**: A component that functions as a submission form for dog adoption offers, collecting information to be submitted to the backend API.

The application utilizes **React Router** to manage navigation between pages, enabling dynamic routing for user-friendly interactions.

### 3.1.1 *Communication with backend API.*

The application uses an asynchronous HTTP request via the Fetch API to retrieve data from a RESTful endpoint. Integrated error handling manages potential data retrieval issues and enhances user experience by providing feedback.

### 3.1.2 *React Unit Tests.*

The React application includes unit tests developed using the Jest[2] testing framework. Tests are organized by component, covering aspects such as navigation in the App component, details display in the DogCard component, form submissions and validations in the Form component, and data fetching and error handling in the Homepage component. This structured testing approach contributes to higher code quality and reduces the likelihood of regressions when changes are made.

## 3.2 Implementation of the Flask Backend

The Flask backend offers an efficient REST API to support data access and adoption submissions. Its primary functions include managing and providing dog adoption data and ensuring smooth communication with a MySQL database. The backend is designed to offer lightweight functionality that complements the frontend’s operations and facilitates robust data handling.

To achieve this, the backend employs several key routes, each associated with specific tasks:

- The `/api/dogs` route handles the retrieval of dog adoption records, fetching data from the MySQL database and returning it in a structured format.

[2] https://jestjs.io

- The **`/api/adoption`** route processes new adoption offer submissions from users. This route captures the data and performs the necessary database operations to insert a new record.
- The **`/health`** route is a dedicated endpoint for application health and life checks. More detailed description about this functionality is available in section 4.6.1.

*3.2.1 Connection to the database.* The connection between the Flask backend and the MySQL database is facilitated through the use of the `mysql.connector`[3] package, which allows the Flask application to perform database operations seamlessly. Upon initialization, the Flask application establishes a connection to the MySQL database using connection parameters defined in the default configuration. These parameters are sourced from environment variables for security and flexibility.

*3.2.2 Flask Unit Tests.* The Flask backend includes unit tests, developed using the `unittest`[4] package, to validate its main application endpoints. These tests cover key areas: the health check; data retrieval, ensuring accurate data fetching and formatting while handling database connection issues; and data submission, testing successful processing and insertion of the data, and appropriate response to any submission errors.

### 3.3 Implementation of the Database

The application employs a MySQL database, to systematically store and manage data pertaining to dogs available for adoption. The database schema comprises a single table, which efficiently organizes critical information regarding individual dogs, including their name, breed, age, and description. Individual records have ID assigned.

## 4 DEPLOYMENT

Deployment is a crucial phase in the software lifecycle, as it ensures that applications are reliably and efficiently delivered to the intended environment. This chapter discusses the deployment methods selected for this project, examining the strategies employed to achieve scalable, stable, and seamless application delivery within the target infrastructure. Each approach is described to emphasize how it contributes to maintaining the application's availability, security, and adaptability in response to changing demands.

It is also important to note that the entire solution was deployed on an OpenStack cloud platform. This deployment environment facilitated scalability, resource management, and enhanced the accessibility of the application for testing and further development stages.

### 4.1 Docker

Docker enables the packaging and execution of applications within a loosely isolated environment known as a container. A **container** is a sandboxed process that operates on a host machine, providing isolation from other processes running on the same host.

Container represents a runnable instance of an **image**, which serves as a template for creating these containers. Images are immutable. This allows the same image to be deployed multiple times while ensuring consistency. A **Dockerfile** is a text document that contains a series of instructions for constructing a Docker image.

*4.1.1 Image Layering and Layer Caching.* One of the fundamental concepts in Docker is image layering[5]. Each filesystem changing command in a Dockerfile creates a new layer in the image, where each layer represents a specific set of these changes.

Furthermore, Docker employs layer caching[6] as a performance optimization mechanism during the image build process. When an image is constructed, Docker caches each layer generated. If the same image is rebuilt and no changes have occurred in the layers, Docker can use the cached layers instead of recreating them. This method decreases the time and resources required to build images, as only the layers that have been modified need to be processed. Since each command specified within a Dockerfile creates a new layer in the resulting image, it is necessary to consider proper command sequence to enhance the efficiency of the image-building process.

### 4.2 Kubernetes

Kubernetes is a technology for managing containerized applications. It facilitates scalability, automatic load balancing across components, and configuration management for each deployment, all while ensuring stability. In case of a component failure, Kubernetes assists in resolving the issue through automated actions, such as restarting the affected pod, as well as by providing logs with essential information about the incident. This approach helps to minimize the overall impact on the application and enables a prompt return to full operation.

A Kubernetes cluster employs several essential components required for its operation. Control plane is responsible for managing the cluster, and the worker machines, which execute the applications. The worker machine is referred to as **node**.

In Kubernetes, a **pod** represents the smallest deployable unit. It encapsulates one or more containers that share the same network namespace and storage, allowing for efficient communication and resource sharing among the containers within the pod.

A **Service** in Kubernetes defines a policy for exposing the pods behind a single endpoint. It facilitates communication between various components of the application and provides stable endpoints for clients, effectively load balancing traffic to the pods. The primary types of services include:

- **ClusterIP**: Exposes the service on a cluster-internal IP address.
- **NodePort**: Exposes the service on a static port on each node's IP address.
- **LoadBalancer**: Exposes the service externally using a provided load balancer.

*4.2.1 Importing Images into Container Runtimes.* In addition to managing application deployment and scaling, it is crucial to ensure that container images are accessible within the Kubernetes environment. To enable the proper functioning of applications, it is

[3]https://pypi.org/project/mysql-connector-python/
[4]https://docs.python.org/3/library/unittest.html
[5]https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/
[6]https://docs.docker.com/get-started/docker-concepts/building-images/using-the-build-cache/

necessary to import the required images directly into the container runtime.

## 4.3 MySQL Database

This chapter discusses the deployment of MySQL database. To integrate MySQL with Docker and Kubernetes, several implementation steps are required including the creation of multiple Kubernetes configuration files. Key challenges include establishing persistent storage and data security tasks. Additionally, ensuring data security is critical: sensitive database connection information needs to be securely stored, with access restricted to internal only. Reliable data availability and stable database connectivity are also essential. To ensure this, the database is requested from the Flask backend using the DNS name. The following section details the solutions implemented to address these challenges.

*4.3.1 Docker Implementation.* A Dockerfile is implemented to create the database container image, containing sections highly specific to database setup. First, it specifies the location for copying files which are executed automatically on startup to create the database and tables. Second, it defines essential environment variables, including root and database user credentials, which are necessary for secure database connectivity.

*4.3.2 Kubernetes Implementation.* As part of the Kubernetes implementation, four key files were created: Deployment, Service, Persistent Volume, and Secrets. The Persistent Volume and Secrets files were particularly important in addressing specific project challenges. The Persistent Volume provides long-term data storage within the cluster and is used with a Persistent Volume Claim, which is a request for storage made by user. Secrets securely store sensitive information. A detailed breakdown of these files follows:

- **MySQL Deployment** This file defines the Kubernetes deployment configuration, specifying key elements required to mount and use the database. For example, specifying the variable `MYSQL_ROOT_PASSWORD` is important and the value is stored securely in the Kubernetes Secrets and imported from this file. Additionally, the path where to mount volume is defined as well as which Persistent Volume Claim to use for storage requests.
- **Persistent volume** This file defines the Persistent Volume and Persistent Volume Claim. Persistent Volume specifies essential information such as volume location, storage size and access mode. Here, the `ReadWriteOnce` mode specifies, that the the volume is *mounted* with read write access by a single node [7]. In the second part of this file, the Persistent Volume Claim is defined. It can *request* physical storage with `ReadWriteOnce` access, ensuring only one node can access the database at a time. Additionally, resource requests define the required storage capacity. Setting these policies, especially access modes are important for preventing database conflicts.
- **MySQL service** This file defines essential information, such as the port and type of Service. Importantly, it is type ClusterIP, making it inaccessible from outside the cluster, thereby enhancing database security by preventing unauthorized access.
- **Kubernetes Secrets** The Secrets file in Kubernetes stores sensitive information, such as database usernames and passwords. By using Secrets, sensitive data are kept separate from application configuration, reducing the risk of exposure[8].

[7]https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/

## 4.4 Flask Backend Deployment

The deployment of the Flask backend for this application is designed to ensure availability, security, and scalability. Key deployment features include Docker containerization, horizontal scaling, and secure environment variable management.

*4.4.1 Docker Containerization.* In the containerization of the backend, emphasis was placed on the effective implementation of layering and layer caching concepts. During the deployment process, the installation of packages represents a step that can benefit from caching. Therefore, instead of copying all the files and then installing the packages, the Dockerfile is structured to separate the tasks into distinct stages. Initially, the `requirements.txt` file is copied into the container, followed by the installation of the necessary packages. After these steps the remaining application files, such as the code, are copied into the container. This method effectively minimizes unnecessary reinstalls of packages when changes are made to the application code, resulting in reduced overall build times.

*4.4.2 Scaling Configuration and Horizontal Pod Autoscaler (HPA).* The backend deployment is configured with an initial replica count of 2 to ensure a baseline level of availability. For scalability, a Kubernetes Horizontal Pod Autoscaler (HPA)[9] is implemented, dynamically adjusting the number of replicas based on CPU utilization.

Horizontal scaling is suitable for stateless APIs as it facilitates the distribution of requests across multiple instances. It helps to prevent the overloading of individual replicas and enhances the application's resilience to handle peak loads effectively. In the project's configuration, if CPU utilization consistently exceeds 50% of the maximum CPU set for the container, the HPA can scale the application up to a maximum of 5 replicas.

The HPA operates within a continuous loop, wherein it systematically monitors resource usage via the metrics server. Based on the collected data, it calculates the required number of replicas using the formula:

$$\text{desiredReplicas} = \lceil \text{currentReplicas} \times \left( \frac{\text{currentMetricValue}}{\text{desiredMetricValue}} \right) \rceil$$

The HPA then determines whether to scale the application up or down to meet the calculated capacity and updates the number of replicas accordingly.

The requested resources are set to 100 mCPU and limit resources to 200 mCPU. Figure 4 illustrates the relationship between resource usage and the number of replicas.

[8]https://kubernetes.io/docs/concepts/configuration/secret/
[9]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

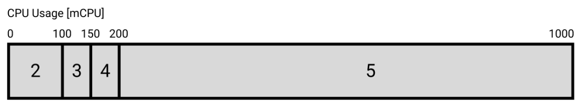


**Figure 4: Dynamic Scaling: Number of Replicas in Relation to CPU Usage**

To test the HPA's scaling behaviour, load-testing was performed using Apache HTTP server benchmarking tool (ab) with 5,000 total requests and a concurrency level of 100. This simulation triggered an increase in replica count, demonstrating the HPA's ability to dynamically scale the application in response to demand.

*4.4.3 Environment Variable Management.* Sensitive configuration details, such as database credentials, are securely managed through environment variable injection[10] from Kubernetes Secrets. This setup enables secure access to MySQL connection parameters, with each variable injected into the application using `valueFrom-.secretKeyRef` to reference the corresponding secret key.

*4.4.4 Service Accessibility and Load Balancing.* The Flask backend must be accessible from outside the Kubernetes cluster, as the React frontend initiates API requests directly from the user's browser. To ensure this external accessibility while accommodating the dynamic scaling capabilities of the HPA, a Kubernetes Service of type LoadBalancer has been implemented.

## 4.5 React Frontend Deployment

The deployment of the frontend uses Docker containerization through a multi-stage Dockerfile for the creation of static files as well as the initialization of an NGINX server for serving these files. Once the container is built, it is deployed using Kubernetes exposed to external traffic, serving as the interface for client interactions.

*4.5.1 Multi-stage Dockerfile.* The Dockerfile for the React frontend utilizes a multi-stage build[11] process to optimize the final image size and improve efficiency. This approach involves multiple `FROM` instructions within the same Dockerfile, each defining a separate stage. By separating the process into multiple distinct stages, smaller Docker images can be created, as only the necessary artifacts are included in the final stage. This is allowed by using the `COPY -from=<stage>` instruction that selects the files to be transferred from one stage to the other, keeping the final image lightweight.

The stages can be referenced by their order, specifically by an integer starting from 0, or be named for better readability and maintainability.

The first stage of the Dockerfile for the React frontend part is dedicated to building the React application, where Node.js is used to install dependencies and then compile the application to transform the source code into a version suitable for production use. The second stage uses NGINX to serve the static files generated from the build. It is configured using a dedicated configuration file that sets various parameters. By copying only the build artifacts from the first stage, unnecessary build tools and intermediate files are being excluded from further work.

*4.5.2 Frontend Service.* For the React frontend, a Kubernetes Service of type NodePort is deployed. This service type is chosen because the frontend needs to be accessible directly from client browsers outside the cluster network, as the NodePort service exposes a specific port for external access.

## 4.6 Security Improvements

This chapter describes the various security enhancements implemented in this project, detailed in the following subsections.

*4.6.1 Health Checks and Life Checks.* A key security enhancement in this solution involves liveness and readiness probes. The liveness probe provides Kubernetes information on when to restart a container, for example in case of deadlocks, and enhances application availability in the case of errors. The readiness probe determines when a container is ready to receive traffic [12]. In this implementation, both probes are employed for the Flask and React applications, with each probe configured as an endpoint that Kubernetes attempts to access.

The usage of these checks might be seen as part of the pod description after running:

```
kubectl describe pod <pod_name>
```

This command will produce the following lines as described in Terminal Output 1.

**Listing 1: Liveness and Readiness in Pod Description**

```
Liveness:    http-get http://:2222/health delay=8s
timeout=1s period=10s #success=1 #failure=3

Readiness:   http-get http://:2222/health delay=5s
timeout=1s period=10s #success=1 #failure=3
```

Upon examining the requests sent to our Flask backend, the following Terminal Output 2 that logs a request to the health endpoint was observed, which further verifies the successful implementation of the check:

**Listing 2: Health Check Request**

```
10.244.0.1 - - [04/Nov/2024 10:01:13] "GET /health
HTTP/1.1" 200 -
```

*4.6.2 Secrets.* The Secrets file is utilized to store sensitive information, keeping it separate from configuration files to minimize the risk of leaks. In this project, it was specifically used to store database credentials, as detailed in Chapter 4.3.2.

The successful usage of this strategy is also obvious from the terminal result of the command

```
kubectl describe pod pod_name
```

In the provided Terminal Output 3, we can then find the environment variables being successfully set from the secrets:

[10]https://kubernetes.io/docs/tasks/inject-data-application/define-environment-variable-container/
[11]https://docs.docker.com/build/building/multi-stage/
[12]https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

**Listing 3: Enviromental Variables**

```
MYSQL_HOST:
    mysql.default.svc.cluster.local
MYSQL_PORT:
    3306
MYSQL_DATABASE:
    animals
MYSQL_ROOT_USERNAME:
    <set to the key 'root_username'
    in secret 'flaskapi-secrets'>
    Optional: false
MYSQL_ROOT_PASSWORD:
    <set to the key 'root_password'
    in secret 'flaskapi-secrets'>
    Optional: false
MYSQL_USER_USERNAME:
    <set to the key 'user_username'
    in secret 'flaskapi-secrets'>
    Optional: false
MYSQL_USER_PASSWORD:
    <set to the key 'user_password'
    in secret 'flaskapi-secrets'>
    Optional: false
```

#### 4.6.3 *Root ReadOnly File System.*

For the Flask backend, the root file system was set as read-only to prevent modifications and reduce the potential for attacks, as attackers cannot alter these files. This setting is appropriate for the Flask backend, which functions solely as an API and does not require modifications to these files for its own run.

#### 4.6.4 *Resource restriction.*

Kubernetes allows resource restrictions on containers, specified through requests and limits. Setting resource restrictions is crucial to prevent resource shortages on a node during pod usage spikes, avoid out-of-memory errors, and help mitigate the impact of Denial of Service(DOS) attacks. The request defines the minimum resources allocated to a pod, guiding Kubernetes to which node is suitable for this pod [13].

Limits caps resource usage, preventing the container from exceeding the resources. Typical constraints include memory and CPU. Generally, CPU is measured in Kubernetes CPU units, and memory in bytes. Both of these restrictions are defined for the React frontend and Flask backend, as shown in Table 1 for CPU usage and in Table 2 for memory.

**Table 1: Resource Restrictions for CPU**

| Component | Request | Limit |
|---|---|---|
| Flask Backend | 100m | 200m |
| React Frontend | 100m | 200m |

[13]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

**Table 2: Resource Restrictions for Memory**

| Component | Request | Limit |
|---|---|---|
| Flask Backend | 64Mi | 128Mi |
| React Frontend | 64Mi | 128Mi |

To check the resource allocation and usage of a specific node, the following command was be executed:

```
kubectl describe node <node_name>
```

The output contains following information:

- **Namespace:** default
- **Name:** react-frontend-deployment-9ffccd6cf-r9jtj
- **CPU Requests:** 100m (2%)
- **CPU Limits:** 200m (5%)
- **Memory Requests:** 64Mi (0%)
- **Memory Limits:** 128Mi (1%)
- **Age:** 34m

This information highlights how resources are allocated, ensuring that the application operates within specified parameters.

The system's response to resource limits differs between backend and frontend. For the React frontend, exceeding memory triggers a default Kubernetes pod restart or stops the process which asked for the memory allocation, while CPU limits result in waiting for available resources. In the case of the Flask backend, exceeding limits prompts scaling, as outlined in Section 4.4.2.

#### 4.6.5 *Environment Variable Injection.*

Finally, an important security enhancement involves the use of environment variable injection. This approach allows to keep sensitive information outside the application code and reduce the risk of exposure. In this project, environment variable injection was employed to securely store database connection details within the Flask backend, as discussed in Chapter 4.4.3.

## 4.7 Testing and Results

Testing is a crucial phase in the development process, essential for validating the application's functionality. To ensure correct operation, we employed both manual and automated testing strategies. Automated tests were implemented to verify the functionality of the application's backend (Flask) and frontend (React), as detailed in Chapters 3.1.2 and 3.2.2. Manual testing involved simulating user interactions on the webpage, as well as reviewing various logs and system descriptions to verify correct behaviour. Examples illustrating these testing methods are provided throughout the report in each relevant chapter.

Furthermore, the development process followed an iterative approach, allowing for incremental additions of new features or enhancements after completing (or nearly completing) current stages. Given the collaborative nature of the project, regular communication and feature documentation were essential. This approach ensured that team members maintained consistency across the application's components and prevented unintentional disruptions to each other's work.

## 5 CONCLUSION

In this project, we implemented a containerized, cloud-based web application designed to support the operations of a dog shelter. The application provides a platform for listing available dogs for adoption and adding new ones, offering a simple and user-friendly interface.

The development followed a three-tier architecture, with the backend implemented in Flask, the frontend in React served by NGINX, and MySQL as the database. Each component was containerized using Docker to ensure modularity, and Kubernetes was employed to orchestrate these containers efficiently. To enhance the user interface, additional technologies such as Bootstrap and HTML were utilized.

The primary focus was on leveraging Docker and Kubernetes to explore their advantages in application development, including improved scalability, efficiency, and security. Key steps in the implementation included setting up Persistent Volumes and Persistent Volume Claims for durable data storage, securing sensitive database credentials with Kubernetes secrets, and configuring resource limits to prevent resource exhaustion. Additionally, horizontal pod autoscaling was applied to enhance reliability. Further features implemented included backend load balancing, DNS for database access, health and liveness checks, environment variable injection for database connection management and a read-only root file system for security.

Comprehensive documentation detailing the application architecture and functionality has been included in the associated GitHub repository. All components and features of the application were rigorously tested, both manually and through automated unit tests.

In future work, we propose implementing a network policy to enhance security by regulating traffic and communication between pods. This policy would specify rules defining connection permissions, such as allowed ports or authorized pods, ensuring that only the Flask API can access the MySQL database while denying access to other traffic. This approach aims to reduce the risk of unauthorized access. To support this, the cluster must utilize a network plugin that enforces NetworkPolicy, such as Calico. Currently, the access restriction issue is partially addressed by configuring the service type as ClusterIP, which permits only internal access.

In conclusion, the web application was created successfully and through its implementation, we were able to explore the advantages of containerized and cloud-based solutions. Additionally, the project effectively addressed the issues outlined in the introduction, achieving enhanced scalability, efficiency, and security.

## REFERENCES

[1] 2024. https://www.docker.com/ Accessed: 2024-11-04.
[2] 2024. https://kubernetes.io/ Accessed: 2024-11-04, Language: English.
[3] 2024. https://flask.palletsprojects.com/en/stable/ Accessed: 2024-11-04, Language: English.
[4] 2024. https://react.dev/ Accessed: 2024-11-04, Language: English.
[5] 2024. https://www.mysql.com/ Accessed: 2024-11-04, Language: English.
[6] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running.* " O'Reilly Media, Inc.".
[7] Miguel Grinberg. 2018. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.".
[8] Karl Matthias and Sean P Kane. 2015. *Docker: Up & Running: Shipping Reliable Containers in Production.* " O'Reilly Media, Inc.".
[9] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

*This report benefited from the use of OpenAI's ChatGPT*[14] *for formal editing and rephrasing. ChatGPT was utilized to enhance the clarity and professionalism of the text, ensuring that language and style met formal academic standards.*

[14] chat.openai.com