KUBERNETES CONFIGURATION FILES CREATED WITH A PYTHON QUART WEB INTERFACE FOR REAL LIFE SCENARIOS OF KUBERNETES DEPLOYMENTS

Ioan-Alexandru DUMITRE[1,2], Szabolcs-Gavril RUGINA[2], Laszlo-Csongor LENGYEL[2], Andrei-Bogdan RUS[1], Virgil DOBROTA[1]
[1]*Communications Department, Technical University of Cluj-Napoca, Romania*
[2]*Frequentis Cluj-Napoca, Romania*
Dumitre.Io.Ioan@student.utcluj.ro, Szabolcs.RUGINA@frequentis.com, laszlo-csongor.lengyel@frequentis.com
Bogdan.Rus@com.utcluj.ro, Virgil.Dobrota@com.utcluj.ro
Corresponding author: Virgil Dobrota (e-mail: Virgil.Dobrota@com.utcluj.ro)

Abstract: This paper presents the development of a web interface using Python Quart to simplify the creation and management of Kubernetes manifest files. The interface allows users to create, update, delete, and manage Kubernetes resources through an intuitive platform. Developed using Test-Driven Development (TDD) principles, the application minimizes errors and improves reliability by ensuring that each component is rigorously tested before deployment. By leveraging Quart's asynchronous capabilities, it handles multiple requests efficiently, enhancing scalability and performance. Performance tests with Locust and real-time monitoring with Prometheus show the system's ability to manage multiple users and highlight areas for improvement in request latency.

Keywords: *Asynchronous Programming, Docker, Kubernetes, Python Quart, Test-Driven Development, WSL.*

I. INTRODUCTION

In recent years, container orchestration has become a fundamental aspect of managing software deployment across cloud-native environments. Kubernetes (k8s) has established itself as the de facto standard for orchestrating containerized applications, enabling seamless deployment, scaling, and management of workloads across distributed infrastructures. Originally developed by Google and now managed by the Cloud Native Computing Foundation (CNCF), Kubernetes is widely regarded as the "operating system for the cloud" due to its ability to manage applications consistently across various environments, including on-premises data centers, public clouds, and hybrid deployments [1][2].

To address the demands of modern web applications, the Python Quart framework is utilized for building asynchronous web applications. Quart, which extends the popular Flask framework, leverages the Asynchronous Server Gateway Interface (ASGI) to support concurrency, allowing the simultaneous handling of multiple tasks. This characteristic makes Quart particularly well-suited for real-time, high-concurrency applications that rely on WebSockets and HTTP/2, offering enhanced performance compared to traditional synchronous frameworks like Flask [3]. The adoption of Quart in Kubernetes management solutions enables developers to create responsive, scalable web interfaces that improve user interaction and system performance.

Several Kubernetes management tools have been developed to simplify the often-complex task of deploying and managing services within Kubernetes clusters. These tools provide capabilities such as multi-cluster management, real-time monitoring, and intuitive interfaces that abstract away the intricacies of Kubernetes configuration files [1].

This paper introduces a Kubernetes management web interface developed using Python Quart, aimed at providing a user-friendly, scalable, and efficient solution for managing Kubernetes resources such as Deployments, Services, PersistentVolumes, PersistentVolumeClaims, and ConfigMaps. The application leverages Quart's asynchronous capabilities to enhance the responsiveness of the interface, enabling users to interact with Kubernetes resources in real time. By employing Test-Driven Development (TDD) practices, the system ensures a high degree of reliability, minimizing errors in configuration and deployment processes [8].

Performance testing with Locust was conducted to assess the scalability of the web interface under increasing user loads. The application demonstrated an ability to handle multiple concurrent users without a significant decrease in performance, maintaining proportional request-handling capacity even as the number of requests increased. However, it was observed that response times increased slightly as user loads grew, indicating potential areas for optimization in handling high traffic.

Real-time monitoring with Prometheus provided additional insights into the performance of the system, with metrics such as request latency and request counts being tracked to identify bottlenecks. As more users accessed the application, request latency increased, highlighting potential weak points that could be addressed to improve

Manuscript received October 18, 2024; revised November 29, 2024

performance under heavy loads. Overall, the results of the tests demonstrated the robustness and scalability of the system, confirming its ability to handle a substantial number of users and maintain operational stability over time.

This paper presents the design, development, and evaluation of a Python Quart-based web interface for managing Kubernetes resources. The interface simplifies Kubernetes management by offering an intuitive platform for creating, managing, and deploying resources within a Kubernetes cluster, while extensive performance testing and monitoring demonstrate the system's ability to scale efficiently and handle concurrent requests under various conditions.

The rest of the paper is organized as follows: Section II presents the related work; Section III describes the implementation solution, followed by the experimental results. The last section presents the conclusions and future work.

II. RELATED WORK

With the rise of containerization and cloud-native technologies, Kubernetes has become the de facto standard for managing containerized applications. Numerous tools have been developed to simplify and enhance the Kubernetes management experience, addressing various needs such as lifecycle management, deployment automation, and user-friendly interfaces.

One of the most widely adopted platforms for Kubernetes cluster management is Rancher. Rancher, as presented in [2], is an open-source platform designed to streamline Kubernetes cluster management. It stands out due to its ability to manage multiple clusters under diverse environments, offering an intuitive user interface and robust capabilities for monitoring, authentication, and alerting. This comprehensive feature set makes Rancher appealing to enterprises seeking a unified Kubernetes management solution. However, despite its versatility, Rancher's additional abstraction layers can potentially impact system performance. For example, in [2], the extra overhead introduced by Rancher's management layer was shown to cause performance slowdowns, especially in scenarios where high efficiency and low latency are critical. The trade-off between usability and performance is a recurring theme, as evidenced by [2][4], which highlights the potential complications arising from Rancher's multi-layered architecture. In this paper, the proposed application addresses the challenge of performance overhead introduced by Rancher's additional abstraction layers. Unlike Rancher, which adds a management layer to simplify multi-cluster operations, this application directly interacts with Kubernetes through `kubectl` commands. This ensures a more lightweight and efficient interaction model, especially for single-cluster deployments where minimal resource consumption and low latency are critical. By avoiding Rancher's multi-layered architecture, the proposed system delivers predictable performance even under increasing loads without the added complexity or resource overhead of managing multiple clusters.

Another important tool in Kubernetes management is the Kubernetes Dashboard, a web-based user interface designed for the deployment and management of applications within Kubernetes clusters. As outlined in [5], the dashboard offers a graphical interface that simplifies the deployment of applications and the monitoring of resources. One of its key features is the ability to directly deploy workloads through the UI, a capability that is particularly advantageous for DevOps and SRE teams. Users can create Kubernetes manifests either manually in the browser or by uploading files from external sources such as Git repositories. This level of integration with external systems allows teams to streamline workflows. However, it is worth noting that the Kubernetes Dashboard must be deployed in the same cluster as the kubectl command-line tool, which can complicate deployments in multi-cluster environments, as noted in [5].

Unlike the Kubernetes Dashboard, which must be deployed within the same cluster as the `kubectl` command-line tool, the proposed application is designed to operate independently of the Kubernetes cluster. This decoupling allows for greater deployment flexibility, enabling the application to be hosted in lightweight environments or external servers without imposing constraints on the Kubernetes cluster itself. This independence simplifies deployment workflows, especially in scenarios involving hybrid or multi-cloud infrastructures which involve multi-clusters environments, where tightly coupled deployments like Kubernetes Dashboard can complicate configurations.

In contrast, Cyclops, as described in [6], focuses on easing the challenges associated with configuring and managing distributed systems on Kubernetes. Cyclops uses an intuitive web-based interface to transform complex Kubernetes YAML configurations into simplified web forms. These forms guide users through the process of deploying applications by pre-filling common data fields, which minimizes the chance of errors during deployment. The structured approach of Cyclops reduces the steep learning curve typically associated with Kubernetes management, as users do not need extensive knowledge of Kubernetes configuration syntax. Furthermore, Cyclops tracks changes and updates made to the configuration templates, facilitating collaboration among teams, especially in scenarios requiring continuous deployment and frequent rollbacks. Its integration with Helm further enhances its utility by allowing users to seamlessly manage Helm charts, which package Kubernetes applications into reusable, configurable units. This integration makes Cyclops a powerful tool for enterprises looking to manage Kubernetes applications with greater efficiency and fewer errors [8].

Another tool that has gained popularity for its ability to manage Kubernetes environments is Kustomize. Unlike Cyclops or Helm, Kustomize takes a different approach by focusing on customization without the need for templates. As outlined in [10], it enables users to modify Kubernetes objects such as Deployments, Services, and ConfigMaps without altering the original YAML files. This is accomplished using a "base and overlay" model, where base files contain the common configuration applicable across different environments, while overlays define the environment-specific changes. This method allows teams to maintain a single source of truth for configuration files, improving consistency and reducing errors. Kustomize's ability to modify Kubernetes objects declaratively makes it a valuable tool for managing configurations across diverse environments. Its features, such as the `secretGenerator` and `configMapGenerator`, further automate the generation of sensitive data and configuration settings, streamlining the deployment process while enhancing security [10] [11].

Kustomize also differs from other tools like Helm in that

it doesn't package applications but rather focuses on patching existing Kubernetes resources. This approach makes it particularly useful in cases where users want to apply modifications to an existing deployment without creating a whole new package. For instance, common labels can be added to all resources, and specific environment variables can be adjusted based on the environment where the application is deployed. This flexibility allows users to adjust deployment configurations quickly and efficiently while maintaining control over the base configurations. Additionally, Kustomize's patching mechanism ensures that changes can be made dynamically, without the need for constant modification of the core YAML files [11].

The challenges that organizations face in deploying and managing Kubernetes applications on a scale have spurred the development of these various tools, each addressing unique needs. While Rancher excels at offering a holistic solution for managing multiple clusters across various infrastructures, its additional layers of abstraction can potentially cause performance degradation. On the other hand, Kubernetes Dashboard simplifies the deployment process through its web-based interface, though it requires careful handling in multi-cluster setups due to its tight coupling with the `kubectl` command-line tool. Cyclops focuses on simplifying the complexities of Kubernetes configuration management by providing a form-based approach, reducing deployment errors and accelerating onboarding for new users. Lastly, Kustomize provides a flexible and declarative solution for managing configurations across multiple environments without modifying the core YAML files [2], [5], [6], [10], [11].

In summary, the tools discussed—Rancher, Kubernetes Dashboard, Cyclops, and Kustomize—represent key innovations in the Kubernetes ecosystem. Each platform addresses different aspects of Kubernetes management, from lifecycle management and monitoring to deployment and customization. While these tools offer varying approaches to solving the complexities of Kubernetes, they all share the common goal of making Kubernetes' environments easier to manage, more efficient, and more reliable for developers and operations teams alike.

III. IMPLEMENTATION

This design implements a web-based interface using Python's Quart framework to manage Kubernetes clusters. The system allows users to create, manage, and delete Kubernetes resources such as deployments, services, ConfigMaps, and persistent volumes (PVs/PVCs), all through an intuitive and user-friendly web interface. The application was developed using Test-Driven Development (TDD) to ensure reliability and efficiency while leveraging asynchronous capabilities of Quart and Kubernetes' orchestration power. The proposed system architecture [12] can be seen in Figure 1.

The key technologies and platforms used in this project were the following: Windows Subsystem for Linux (WSL) with Ubuntu 22.04 for the development environment, Microk8s as a lightweight Kubernetes distribution, Docker as the container platform for managing resources, Prometheus for real-time performance monitoring, Locust for load testing and scalability assessment.

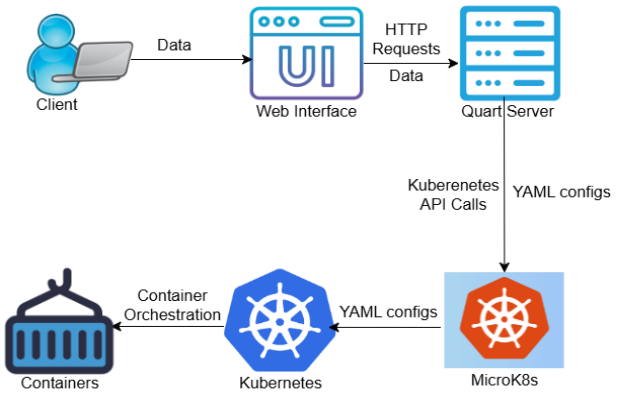

Figure 1. System architecture

A. Environment Setup

The setup process began by configuring the development environment on a Windows machine using WSL. This was followed by the installation of essential tools like Microk8s for Kubernetes, Docker for container management, and various Python libraries such as Quart for building the web interface and pytest for unit testing, Prometheus for monitoring, Locust for load testing. These tools facilitated seamless Kubernetes cluster management through commands prefixed with microk8s. The environment setup included the key steps as can be seen in Figure 2.

```
alex@AlexDumitree:~$ sudo apt upgrade & apt update
alex@AlexDumitree:~$ pip install quart pytest pytest-asyncio pyyaml
alex@AlexDumitree:~$ sudo apt install docker.io
alex@AlexDumitree:~$ sudo snap install microk8s –classic
alex@AlexDumitree:~$ sudo usermod -a -G microk8s $USER
```

Figure 2. Environment setup

The application is designed using layered architecture, each layer responsible for specific aspects of the system. This modular structure ensures maintainability and scalability of the application.

a) *Routing Layer:* Defines the endpoints and routes for user interactions.
b) *Services Layer:* It implements the core logic for managing Kubernetes resources.
c) *Kubernetes Utilities Layer:* This layer facilitates the deployment and management of Kubernetes resources via YAML manifest files.
d) *Configuration Management Layer:* This converts user input into properly formatted Kubernetes YAML configuration files. Testing Layer: Provides test cases to validate each feature of the application, ensuring that it works as intended.

B. Layers

In the Routing Layer, various routes are defined to handle user actions like creating deployments or services, injecting data, or deleting Kubernetes resources. For example, the route responsible for creating a Kubernetes deployment can be seen in Figure 3. It illustrates how HTTP requests (in this case, a POST request to /create-deployment) are handled. The Routing Layer collects form data, processes it by calling the service layer function `create_deployment()`, and renders the result on the `result.html` page.

```
@app.route('/create-deployment', methods=['POST'])
async def create_deployment_route():
    try:
         data = await request.form
    except Exception:
         return await render_template('result.html',
         message="Invalid form data")
    message, success = create_deployment(data)
    return await render_template('result.html',
    message=message)
```

Figure 3. Routing option

The cores logic for Kubernetes resource management resides in the Services Layer. Each function handles a specific resource type, such as deployments, services, or persistent volumes (PV/PVC). Figure 4 shows the function of creating a Kubernetes service. This retrieved data from the form, validated it, generated a Kubernetes service configuration file (YAML format), and called the Kubernetes utility function `deploy_to_microk8s()` to deploy it.

```
submitted_configs = {}
def create_service(data):
    service_name = data.get('service_name')
    app_label = data.get('app_label')
    port = data.get('port')
    target_port = data.get('target_port')
    protocol = data.get('protocol')
    service_type = data.get('type')
```

Figure 4. create_service function

The Kubernetes Utilities Layer handles the interaction between the web application and the Kubernetes cluster. It contains utility functions responsible for deploying resources, injecting data into existing resources, and handling errors such as duplicate configurations. As seen in Figure 5 the `deploy_to_microk8s ()` function ensured that Kubernetes resources were deployed properly by interacting with the `kubectl` command:

```
def deploy_to_microk8s(deployment_config,
 service_config):
with open('/tmp/kubernetes_deployment.yaml', 'w') as f:
        f.write(deployment_config)
with open('/tmp/kubernetes_service.yaml', 'w') as f:
        f.write(service_config)
subprocess.run(['microk8s', 'kubectl', 'apply', '-f',
'/tmp/kubernetes_deployment.yaml'], check=True)
subprocess.run(['microk8s', 'kubectl', 'apply', '-f',
'/tmp/kubernetes_service.yaml'], check=True)
```

Figure 5. deploy_to_microk8s function

Finally, the Configuration Management Layer is responsible for configuration management, converting the data from the form into YAML format. This has two main components: (1) `config_generator.py` to copy the predefined dictionary templates; and (2) `deployment_templates.py` to populate them with data from the user (see Figure 6). Each field of the template was matched with the corresponding data from the user input that was passed as arguments to the functions.

```
service_config['metadata']['name'] = service_name
service_config['spec']['type'] = service_type
service_config['spec']['selector']['app'] = app_label
service_config['spec']['ports'][0]['protocol'] =
protocol
service_config['spec']['ports'][0]['port'] = port
service_config['spec']['ports'][0]['targetPort']      =
target_port
```

Figure 6. config_generator.py file

Moving on to the Testing Layer to ensure that each functionality of the application was working properly various test cases were developed in `test_routes.py`. These tests simulated different HTTP requests with valid, invalid, and missing data sent to the Quart application testing how the application responds. To set up an instance of the application for testing purposes, the QuartClient module has been used. The code in Figure 7 was essential for testing the application.

```
@pytest.fixture
@pytest.mark.asyncio
async def client():
    async with QuartClient(app) as client:
        yield client
```

Figure 7. test_routes.py file

The client function built the environment and verified that everything was set up for testing. The decorator `pytest.fixture` transformed this function into a fixture. This offered a stable basis for tests to operate smoothly and repeatably, by providing the state or context for the tests. The other decorator `pytest.mark.asyncio` was used to indicate that this fixture was asynchronous. `QuartClient(app)` provided an instance of a client interacting with the application and as client assigned this instance to the client variable. The statement `yield client` gave access to the client to all the tests that used this fixture. Several test cases were tested in `test_routes.py`: (1) creation of a deployment with valid and invalid data respectively duplicate data; (2) creation of a service with valid and invalid data respectively duplicate data; (3) creation of a configmap, injecting a configmap, retrieving of deployments, services, and pods.

C. Performance Monitoring and Load Testing

The system integrated Prometheus for performance monitoring. It scraped metrics from the web application and visualized them on a dashboard. The configuration in Figure 8 added the web app as a target for Prometheus monitoring.

```
scrape_configs:
- job_name: 'prometheus'
static_configs:
- targets: ['localhost:9090']
- job_name: 'your_app'
static_configs:
    -targets: [`127.0.0.1:5000’] ['127.0.0.1:5000']
```

Figure 8. prometheus.yml file

The script `locustfile.py` was used for load testing, simulating high-traffic scenarios by creating and deleting resources in a rapid sequence. According to Figure 9, it generated random names for deploying, to create, and then to delete them to test system performance.

```
class UserBehavior(HttpUser):
    wait_time = between(1, 5)
    def generate_unique_name(self, base):
         unique_suffix =
         ''.join(random.choices(string.ascii_lowercase +
string.digits, k=6))
    return f"{base}-{int(time.time())}-{unique_suffix}"
```

Figure 9. locustfile.py file

D. Front-end Interface

The front-end of the application was developed using HTML and CSS, with JavaScript providing interactivity. The forms on the main page were designed to be

dynamically shown or hidden based on the user's selection. This was managed through the `toggleForm()` JavaScript function that can be seen in Figure 10.

```
<head>
    <title>Kubernetes Management</title>
    <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
     <h1>Kubernetes Management</h1>
     <p>This project provides a web interface for
managing Kubernetes resources. You can create and
manage deployments, services, ConfigMaps, and volumes
easily through this interface.</p>
    <div class="button-container">
         <button onclick="toggleForm('deployment-
         form')">Create Deployment</button>
         <button onclick="toggleForm('service-
         form')">Create
Service</button>
   </div>
```

Figure 10. index.html file

Each form corresponded to a different Kubernetes resource, such as deployments or services. When a user filled out a form and submitted it, the `submitForm()` function (see Figure 11), handled the asynchronous request, by forwarding data to the backend without reloading the page.

```
async function submitForm(event, messageId) {
event.preventDefault();
const form = event.target;
const formData = new FormData(form);
const response = await fetch(form.action, {
method: form.method,
body: formData
});
const message = await response.text();
document.getElementById(messageId).innerHTML = message;
document.getElementById(messageId).style.display =
'block';}
```

Figure 11. submitForm() function

The template in `resources.html` (Figure 12) was used to dynamically display lists of Kubernetes resources along with their IP and options to delete the resource. To display resources dynamically based on the resource type, placeholders from `jinja2` were needed. This template passed a list of resources names from `index.html` and a `jinja2` for loop iterates over this list rendering each resource in the list. After rendering the resources, the IP and an option to delete them were added in the form of a button.

```
{% for resource, detail in resources %}
    <li>
    {{ resource }} - {{ detail }}
    <form action="/delete-resource" method="post"
    style="display:inline;"
onsubmit="deleteResource(event, '{{ resource }}', '{{
resource_type }}')">
    <input type="hidden" name="resource_type" value="{{
resource_type }}">
    <input type="hidden" name="resource_name" value="{{
resource }}">
    <button type="submit">Delete</button>
    </form>
    </li>
```

Figure 12. Dynamically populated results

IV. EXPERIMENTAL RESULTS

The application started locally on loopback address 127.0.0.1 and listened on port 5000. When a user accessed the home page via HTTP GET, the application responded with the appropriate template.

A. Deployment Creation

The interface enabled resource creation through forms, beginning with Kubernetes Deployments. For instance, when a user completed a Deployment Form (name: test, label: myapp, image: nginx), the deployment was processed by Kubernetes, as shown in Figure 13.

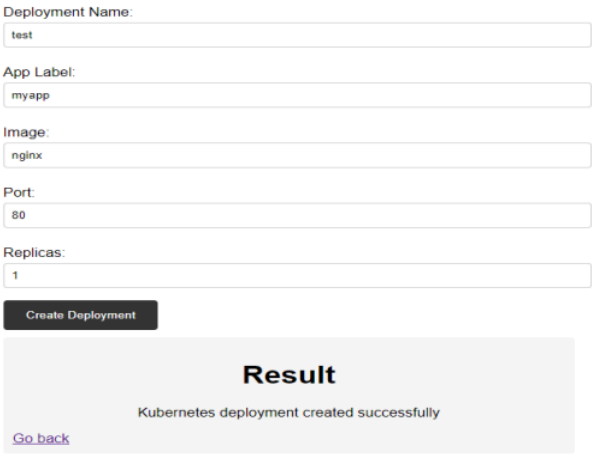

Figure 13. Deployment Form

This setup allowed the user to create and to validate deployments in the cluster through both the web interface and command-line tools. If invalid input was provided, the system responded with appropriate error messages, such as when an out-of-range port number has been entered.

B. Service and ConfigMaps Creation

Similarly, the Service form collected details like the service name, label, port, and service type, validating the input before sending it to Kubernetes. Once deployed, the service could be accessed and verified by checking its IP address. Figure 14 illustrates the output of a nginx service after deployment. The next stage involved creating and injecting ConfigMaps into pods. ConfigMaps allowed the application's configurations to be updated without rebuilding the container.

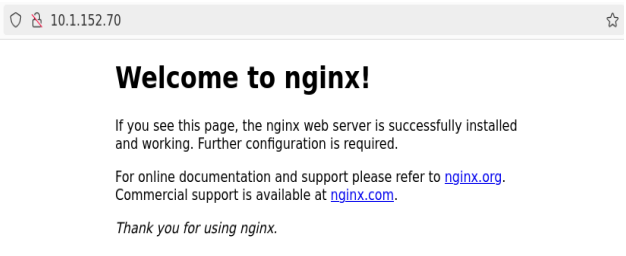

Figure 14. Deployed service

C. Persistent Storage with PVs and PVCs

To manage persistent data, the application supported the creation of PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs). PVs ensured that data persisted beyond the lifecycle of individual pods. For example, a PV was created with a storage capacity of 2Gi, as shown in Figure 15, and it was then claimed by a PVC, which allowed multiple pods to access shared storage. This storage flexibility is essential for maintaining persistent

application states, even as pods are restarted or rescheduled.

Figure 15. PersistentVolume Form

To enhance user experience, the application included buttons for visualizing each Kubernetes resource, such as deployments, services, ConfigMaps, PVs, and PVCs. As in Figure 16, these could also be deleted directly from the interface, improving resource management and transparency for the user.

Figure 16. Interface resource retrieval options

D. Performance Testing: Locust and Prometheus

The performance of the system was evaluated using Locust for load testing and Prometheus for monitoring. Locust tests simulated increasing user loads, ranging from 1 to 20 concurrent users, tracking metrics like response time and requests per second (RPS). Figure 17 summarizes the results, showing that the application efficiently handled traffic, maintaining acceptable response times and zero failed requests up to 20 users.

Run Number	Number of concurrent users	Ramp up rate	Average RPS	Average failures/s	Average response time (ms)
no. 1	1	1	0.6	0	285
no. 2	2	2	1.1	0	364
no. 3	5	5	2.3	0	732.2
no. 3	10	10	2.6	0	2323.9
no. 4	20	20	2.7	0	6733.63

Figure 17. Performance testing results

The graphs representing these results can be seen in Figure 18. Prometheus was used to monitor request latency and the total number of requests. As shown in Figure 19, latency increased predictably as traffic scaled up, with notable increments during higher user loads, such as run 3, where 10 concurrent users resulted in a latency of approximately 507 ms. Despite this, the system demonstrated robustness and scalability, remaining responsive under heavy traffic.

The proposed management interface offers basic troubleshooting feedback directly through the user interface.

Figure 18. Graphs for performance testing

Users are informed about common deployment issues, such as invalid Docker images, duplicate resources, or syntax errors in configuration files, ensuring that typical errors can be corrected quickly. However, the system does not currently provide in-depth troubleshooting tools, such as detailed logs or advanced diagnostics, which are planned for future development.

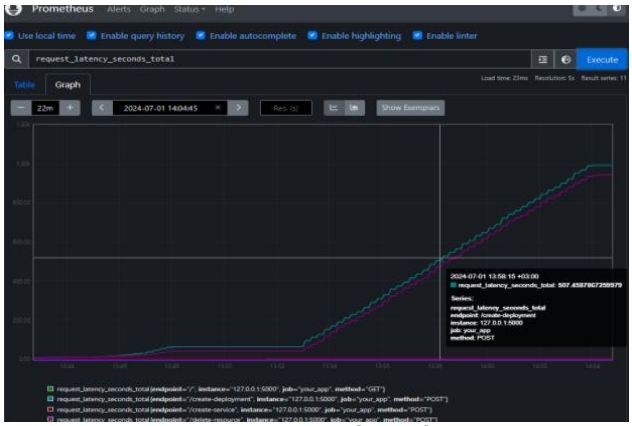

Figure 19. Prometheus charts

V. CONCLUSIONS AND FUTURE WORK

This paper presented a way to develop a Kubernetes management web interface using Python Quart, with thorough testing and monitoring to ensure functionality, performance, and reliability.

This application distinguishes itself from its modern asynchronous architecture powered by Python Quart. Unlike frameworks that rely on the Web Server Gateway Interface (WSGI), Python Quart is built on the Asynchronous Server Gateway Interface (ASGI). ASGI enables asynchronous request handling, allowing the application to manage multiple simultaneous HTTP requests, WebSocket connections, and background tasks with minimal latency. In terms of performance the application stands out from similar technologies, as using Python Quart is particularly advantageous, especially in high-concurrency scenarios, as it prevents blocking operations that are common in WSGI-based frameworks.

Testing the core features of the interface confirmed that

it could effectively create, manage, and retrieve deployments, services, PersistentVolumes, PersistentVolumeClaims, and ConfigMaps within Kubernetes. These tests demonstrated the application's ability to handle both valid and invalid inputs, providing appropriate responses without being disrupted by erroneous configurations or duplicate entries. The utility functions interacting with Kubernetes were validated, ensuring accurate command execution across different configuration states. This confirmed the system's ability to deploy configurations consistently, even when faced with incomplete or incorrect data, contributing to operational stability. Performance testing with Locust revealed how the application handles concurrent user loads, demonstrating scalability as user numbers increased. While response times grew under heavier loads, the system maintained its request-handling capacity proportionally. Prometheus monitoring offered real-time insights into request latency and count, highlighting that increased user traffic led to greater latency. This provided a clear view of the system's behavior under stress and identified potential areas for improvement. Despite the growing latency, the application consistently handled higher traffic levels, proving its capability to manage multiple requests over extended periods.

Unlike other similar technologies such as Rancher, which adds abstraction layers that can increase overhead, the proposed solution directly interacts with Kubernetes, making it ideal for single-cluster deployments. It also addresses the deployment constraints of Kubernetes Dashboard by operating independently of the cluster, allowing for more flexible hosting options also, compared to Cyclops, which simplifies configurations through pre-filled forms, this solution provides a modular architecture that combines configuration management with performance monitoring, offering greater control and scalability.

The future improvements could focus on enhancing scalability and reliability, extending the system's use to larger projects and enterprises.

ACKNOWLEDGMENT

An initial expanded version was presented by I.A. Dumitre as B.Sc. thesis in Telecommunications Technologies and Systems in 19 July 2024.

REFERENCES

[1] M. Levan, "50 Kubernetes Concepts Every DevOps Engineer Should Know: Your Go-To Guide for Making Production-Level Decisions on How to and Why to implement Kubernetes", Packt Publishing, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10163034 .

[2] M. Mattox, "Rancher Deep Dive: Manage enterprise Kubernetes seamlessly with Rancher", Packt Publishing, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/10162992 .

[4] "6 Common Issues and How to Tackle Them with Kubernetes Monitoring Tools", ManageEngine, 2023. [Online]. Available: https://blogs.manageengine.com/application-performance-2/appmanager/2023/02/24/6-issues-kubernetes-monitoring-tools.html .

[5] "What is Kubernetes Dashboard?", Devtron, 2023. [Online]. Available: https://devtron.ai/what-is-kubernetes-dashboard .

[6] S. Rajhi, "Turning Kubernetes into a developer friendly platform with Cyclops", 2023. [Online]. Available: https://medium.com/@seifeddinerajhi/turning-kubernetes-into-a-developer-friendly-platform-with-cyclops-%EF%B8%8F-e5a81128030f .

[7] B.B. Rad, H.J. Bhatti, M. Ahmadi, "An Introduction to Docker and Analysis of its Performance", Int. Journal of Computer Science and Network Security (IJCSNS), 17(3), 228, 2017.

[8] J. Schmitt, "Test-Driven Development (TDD) explained", 2024. [Online]. Available https://circleci.com/blog/test-driven-development-tdd/.

[9] J. Schmitt, "What is Helm?", 2023. [Online]. Available: https://circleci.com/blog/what-is-helm/.

[11] S. Reddy, "Kustomize Kubernetes native configuration management", 2023. [Online]. Available: https://subbaramireddyk.medium.com/kustomizekubernetes-native-configuration-management-f51630d29ac0.

[12] I.A. Dumitre, "Creation of Kubernetes Configuration Files with a Python Quart Web Interface Developed Using TDD for Presenting Real Life Scenarios of Kubernetes Deployments", B.Sc. Thesis, Technical University of Cluj-Napoca, Romania, 19 July 2024.