

Bachelor Thesis

Design and implementation of an automated monitoring solution and corrective actions in Openshift for heterogeneous system.



Author: Dave Joe

Matriculation Number: 4019060

Degree: Bachelor of Engineering, Mechatronics

Faculty: Electrical Engineering

Supervisor: Mr. Andre Haeitmann Dutra

First examiner: Prof. Dr. Rainer Hirn

Second examiner: Prof. Dr. Gunther Bohn

Submission Date: 30.01.2024

Contents

List of Figures	III
Abbreviations	IV
1 Introduction	1
1.1 Overview	1
1.2 Aim and Scope	1
1.3 Significance	1
1.4 Objective	1
1.5 Contributions	2
1.6 Outline	2
2 API Development	3
2.1 Design Considerations	3
2.2 Gathering metrics	3
2.3 What are APIs	4
2.3.1 What is an API Endpoint?	5
2.3.2 Why are API Endpoints Important?	5
2.4 Central API	6
2.5 Webpage API	6
2.6 Summary	8
3 Integration with OpenShift	9
3.1 Red Hat OpenShift	9
3.2 Containers: A Modern Approach to Application Deployment	10
3.2.1 Evolution of Deployment Strategies	10
3.3 Kubernetes Overview	11
3.3.1 How Kubernetes Works	12
3.4 The Relationship Between OpenShift and Kubernetes	13
3.5 Docker Overview	13
3.6 How to Build a Container	15
3.6.1 Dockerfile Creation	15
3.6.2 Building the Docker Image	16
3.6.3 Running the Container	17
3.6.4 Container Operations	17
3.7 Reasons to Use Red Hat OpenShift	17
3.8 Implementation Steps	18
3.8.1 Method 1	18
3.8.2 Method 2	18
3.9 Summary	19

4	Real-Time Monitoring	20
4.1	Introduction	20
4.2	Architecture for Real-Time Monitoring	20
4.2.1	In-Memory Data Storage	20
4.2.2	Background Scheduler	20
4.2.3	Metric Collection Routes	21
4.2.4	Data Analysis and Anomaly Detection	21
4.3	Detecting Outages	21
4.3.1	The function	21
4.3.2	Running the function at a set interval	23
4.4	Detecting Anomalies	23
4.4.1	The function	24
4.5	Conclusion	26
5	Corrective actions	27
5.1	Introduction to Corrective Actions	27
5.2	Design Considerations for Corrective Actions	27
5.3	Criteria for Sending Notifications	27
5.4	Implementing Notifications in MS Teams	28
5.4.1	Making the Webhook	28
5.4.2	Using the Webhook	28
5.5	Case Studies in Corrective Actions	30
5.6	Conclusion	31
6	Results	32
6.1	Results	32
6.1.1	Performance Metrics	32
6.1.2	Adaptability of the system	32
6.1.3	Effectiveness of Monitoring and Corrective Actions	32
6.1.4	Conclusion	33
7	Conclusion	34
A	Setting up Incoming Webhooks	35
A.0.1	Generating the Webhook	35

List of Figures

2.1	API Functioning[16]	4
2.2	Endpoint example[3]	5
2.3	Results for Linux	8
2.4	Results for Windows	8
3.1	Redhat Openshift [12]	9
3.2	Evolution of containers[7]	10
3.3	Containers in docker[4]	15
3.4	How to build a container[6]	15
3.5	Dockerfile to layers[5]	16
3.6	Dockerfile	16
3.7	docker build	17
3.8	docker run	17
4.1	Function that checks for outages	22
4.2	Code to run the function at set intervals	23
4.3	Code to check the metrics as soon as they are received.	24
4.4	Code to Check for Anomalies	25
5.1	Function to send a message on MS teams	29
6.1	Enter Caption	33
A.1	Options in Channel	35
A.2	Available connectors	36
A.3	Webhook creation	37

Abbreviations

API:	APPLICATION PROGRAMMING INTERFACE
CPU:	CENTRAL PROCESSING UNIT
RAM:	RANDOM-ACCESS MEMORY
OS:	OPERATING SYSTEM
URL:	UNIFORM RESOURCE LOCATOR
HTML:	HYPERTEXT MARKUP LANGUAGE
JSON:	JAVAScript OBJECT NOTATION
PAAS:	PLATFORM AS A SERVICE
CLI:	COMMAND-LINE INTERFACE
VMs:	VIRTUAL MACHINES
K8s:	KUBERNETES
CNCF:	CLOUD NATIVE COMPUTING FOUNDATION
CI/CD:	CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY
SSE:	SERVER-SENT EVENTS
YAML:	YAML AIN'T MARKUP LANGUAGE

Chapter 1

Introduction

1.1 Overview

This thesis documents the development of an automated monitoring solution in OpenShift, tailored specially towards heterogeneous systems, with a focus on crucial metrics like CPU, RAM, and disk utilization. As a bachelor student working in Software DevOps, I explored the challenges of monitoring diverse technologies and the role of OpenShift in continuous monitoring and reliability. The core of this project revolves around creating robust APIs for data analysis and transmission, and then implementing real-time monitoring with automated corrective actions, primarily alerting through Microsoft Teams.

1.2 Aim and Scope

The aim of the thesis is to enhance system efficiency and stability by developing a real-time monitoring solution that is capable of promptly identifying and reporting anomalies, facilitating swift corrective measures by the relevant team. The scope includes API development, efficient data transmission, integrating the APIs into the Openshift platform and integration with communication tools such as MS teams for immediate alerts.

1.3 Significance

This work is significant in providing a practical hands-on approach to automated monitoring, addressing the complexities in managing heterogeneous systems. Heterogeneous systems implies that the systems we will be monitoring vary in OS and different networks this adds a level of complexity to the project.

1.4 Objective

The primary objective of this thesis is the design and implementation of a comprehensive automated monitoring solution in OpenShift for heterogeneous systems. The focus is on developing a monitoring framework that is capable of efficiently collecting and analyzing data from diverse system components. Furthermore, the thesis aims to establish mechanisms for real-time anomaly detection and automated corrective actions, thereby increasing the system's uptime and efficiency.

1.5 Contributions

The contributions of this project are:

- **Development of Targeted Monitoring Metrics:** Implementing a monitoring system that can specifically track CPU, RAM, and disk utilization, providing the solution with essential data for system health assessment.
- **Real-time Alert Mechanism:** Integrating an alert system that can communicate anomalies and outages directly to a Microsoft Teams channel, ensuring prompt awareness and response from relevant team members.
- **Optimized Resource Utilization:** By focusing on crucial metrics, the system provides actionable insights for better resource management, potentially leading to enhanced system performance and stability.

1.6 Outline

The thesis is structured to guide the reader through the various stages of the project.

- In Chapter 2, the design and development of the automated monitoring solution are detailed, emphasizing the technical aspects and the rationale behind choosing specific metrics.
- Chapter 3 focuses on the integration process of the monitoring solution within the OpenShift environment. This chapter will discuss the technologies used, the technical steps involved in deploying the solution to OpenShift, the challenges encountered during integration, and how these challenges were overcome.
- In Chapter 4, the implementation of the real-time monitoring system is discussed, focusing on checking if a system is online and then detecting anomalies based off of metrics received.
- In Chapter 5, the implementation of corrective actions is discussed, focusing on the integration with Microsoft Teams and the mechanisms in place to ensure that the people responsible are promptly notified and the information is made available in an efficient manner without negatively impacting the channel.
- Chapter 6 presents the evaluation of the system's performance, discussing the results and the impact of the solution.
- Finally, Chapter 7 provides a summary of the project, reflecting on the achievements and potential areas for future development.

Chapter 2

API Development

2.1 Design Considerations

The main requirement of this project is to monitor different systems and detect whether there are any anomalies. To achieve this objective, it is first necessary to define the specific nature of anomalies within the context of this thesis. Anomalies are anything that can interfere with the normal operation of the system and inhibit code execution on that particular system. In pursuit of this goal, we have chosen to monitor metrics such as CPU, RAM, and disk space utilization. The rationale behind the selection of these particular metrics will now be elaborated.

- **CPU:** High CPU utilization during code execution can lead to slower performance as the processor becomes overburdened with tasks. In our case, these systems are shared between different people, and there exists a scenario where high CPU usage can negatively impact the developers.
- **RAM:** Adequate RAM is essential for code execution, as it provides the space for storing and quickly accessing the data and instructions that are currently in use. Insufficient RAM leads to the system using disk space, which is significantly slower and can create a bottleneck.
- **Disk Space Utilization:** While disk space does not directly impact code execution like RAM and CPU, it affects the overall performance of the system. In shared systems, high disk usage can lead to cluttering and accumulation of junk files, affecting system efficiency.

2.2 Gathering metrics

This section goes over how the metrics gathered from the systems that are being used. The process involves the use of a Python script, `data.py`, designed to collect and send system metrics from the system to a specified API endpoint.

The script utilizes the `psutil` library to access system metrics. It continuously monitors and collects data on CPU utilization, RAM usage, and disk occupancy in terms of percentages. `Psutil`[8] is a Python library for system and process monitoring, providing functionalities like those of UNIX tools (`ps`, `top`, `netstat`) across multiple platforms including Linux, Windows, and macOS.

Once these metrics are collected, they are structured into a JSON format, comprising of `cpu_percent`, `ram_percent`, and `disk_occupancy`. This data is then sent to an API end-

point, defined by the variable `api_url`. This is the endpoint that is responsible for collecting all these metrics.

The script is designed to run in a loop, sending updated metrics at regular intervals (every 5 seconds as per the current setting). It includes error handling to manage any issues that might arise during data collection or transmission, ensuring reliable and consistent monitoring.

2.3 What are APIs

An Application Programming Interface (API)[1] is a set of protocols for building and integrating application software. It facilitates interactions between different software programs, allowing them to communicate and collaborate effectively. APIs are designed to expose specific functionalities and data, keeping the internal workings of a system hidden, thus simplifying software development.

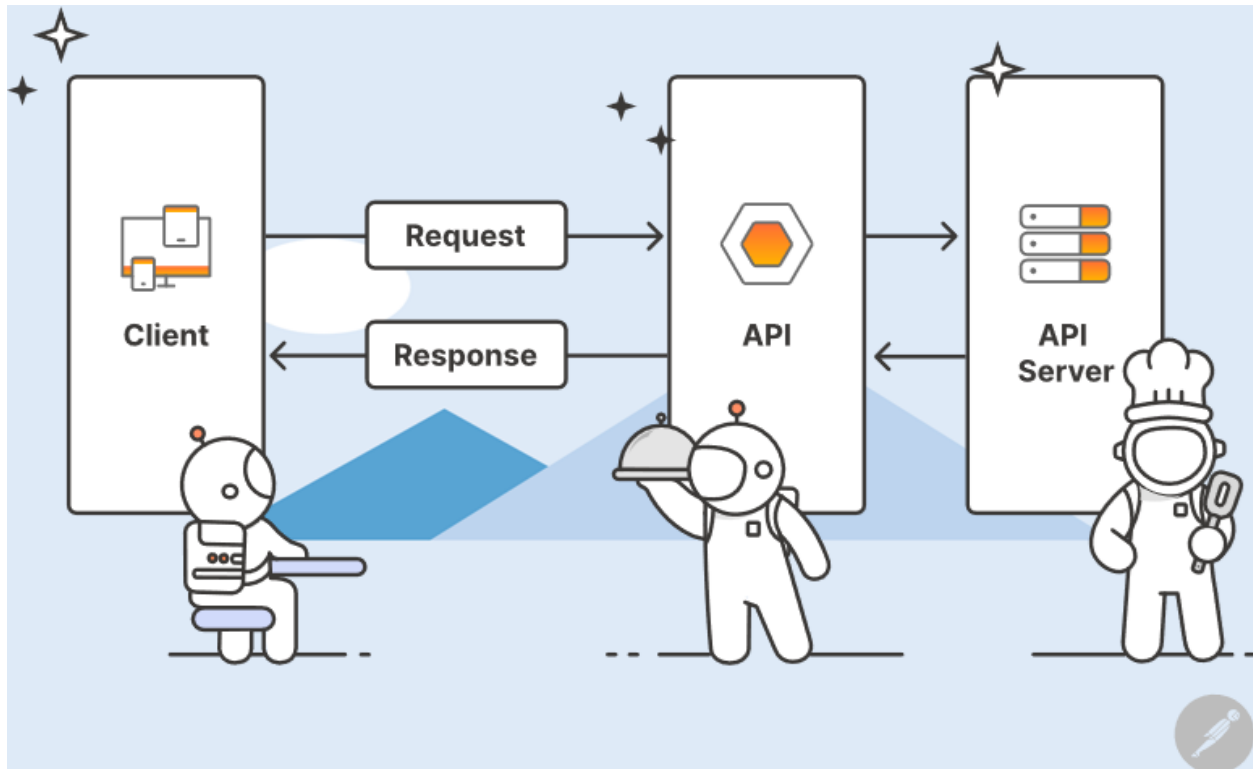


Figure 2.1: API Functioning[16]

2.3.1 What is an API Endpoint?

An API endpoint[15] is fundamentally one end of a communication channel. When an API interacts with another system, the points of interaction are known as endpoints. These endpoints often take the form of URLs pointing to a server or service, each representing a specific location from which APIs can access the resources they need to perform their functions.

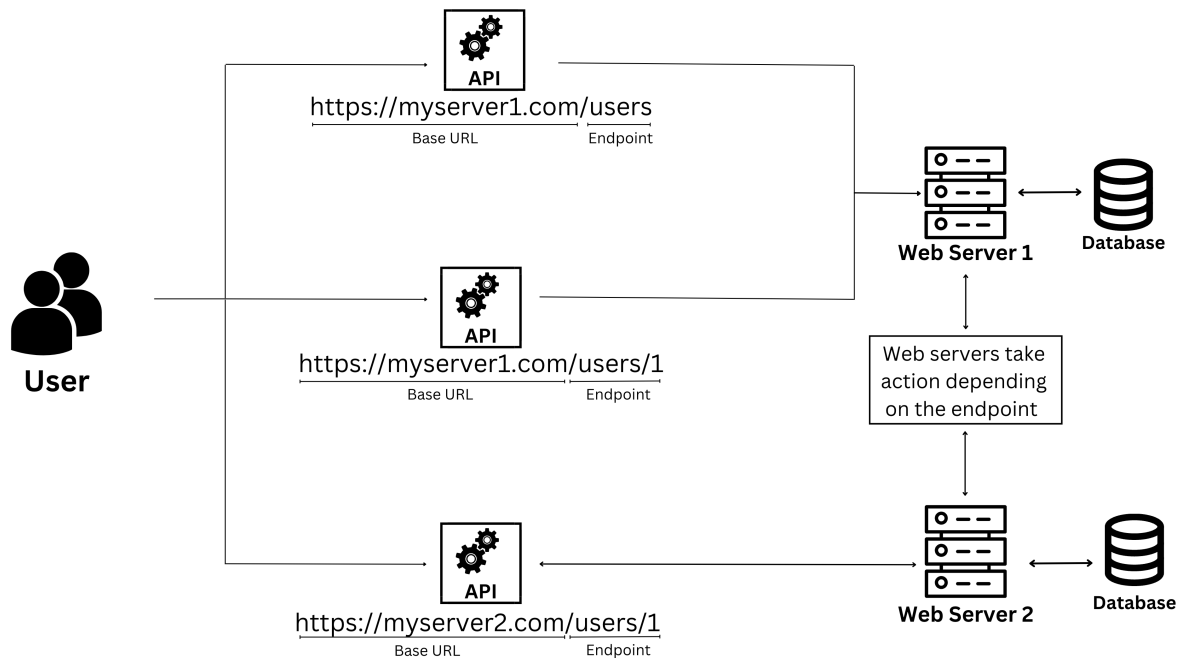


Figure 2.2: Endpoint example[3]

The operation of APIs is centered around the concepts of **requests** and **responses**. An API sends a request to a web application or server and waits for a response. The location where the API sends these requests and where the requested resource resides is what is referred to as an endpoint.

2.3.2 Why are API Endpoints Important?

Across the globe, companies utilize APIs to exchange essential information, carry out processes, and conduct transactions. The importance of APIs is only set to increase as technology progresses. Ensuring the integrity and functionality of each touch-point in API communication is crucial to the success of each API. Endpoints determine where resources can be accessed by APIs and are instrumental in ensuring the correct functioning of the software that interacts with them. In essence, the performance and effectiveness of an API hinge on its ability to communicate seamlessly with these API Endpoints.

2.4 Central API

In the following section, the thesis delves into the specifics of the Application Programming Interface (API) tasked with gathering metrics from a range of systems and subsequently making them available. Henceforth, this API will be designated as the **Central API** for ease of reference and clarity. The `api.py` script in our project serves as an API that handles the collection and storage of system metrics from Linux and Windows systems. After the collection of these metrics these metrics are then organised and sent out to another API. Below is a breakdown of its key components:

- **Flask Setup:** The script uses Flask[13], a micro web framework in Python, to set up the API. Flask provides the necessary tools to create routes and handle HTTP requests.
- **In-Memory Storage:** Two dictionaries, `linux_metrics` and `windows_metrics`, are used for in-memory storage of metrics received from Linux and Windows systems, respectively.
- **Metric Reception Endpoints:**
 - The `/linux-metrics` endpoint is set up to receive and store metrics from Linux systems. It processes POST requests, extracts the received data, and stores it in the `linux_metrics` dictionary.
 - Similarly, the `/windows-metrics` endpoint handles metrics from Windows systems, storing the data in `windows_metrics`.
- **Metric Retrieval Endpoints:** The `/get-metrics` endpoint allows retrieval of all stored metrics, returning them in a JSON format. This facilitates easy access and visualization of the metrics. This is how our other API gets its information.
- **Corrective actions:** The central API is also responsible for detecting anomalies or outages and then taking the required action to resolve the issue. This aspect of the API will be discussed in greater detail in 5.

This API forms the backbone of our monitoring solution, efficiently managing the reception, storage, and retrieval of system metrics, thereby enabling real-time monitoring and analysis of system performance.

2.5 Webpage API

The `stats.py` script, coupled with the HTML code, forms a Web API that displays real-time system metrics for Windows and Linux servers. This API will be referred to as the **Webpage API** in the future. The Python script uses Flask to create an API that serves system metrics data as JSON, which is then consumed by an HTML page that visualizes this data using charts.

The Flask application defines routes for accessing the homepage, as well as specific endpoints for fetching metrics from Windows and Linux systems. It uses the `requests` library to fetch metrics from the endpoint `/get-metrics` which was created in the Central API, parses the JSON response, and extracts CPU usage, RAM usage, and disk occupancy data. This data is then sent to the HTML page.

The HTML page includes links to view metrics for each system and uses Chart.js to plot the data on canvas elements. The JavaScript code establishes a server-sent events (SSE) connection to the `/plot` endpoint, which streams the metrics data continuously. The charts are dynamically updated with this data, showing the percentage of CPU, RAM, and disk usage over time.

This integration of Flask and Chart.js allows for the creation of a live dashboard, which can be critical for system administrators for monitoring and decision-making purposes. The `stats.py` script and HTML interface work together to create an interactive experience where the system's health can be monitored in real-time through a web browser.

2.6 Summary

In conclusion, this chapter synthesizes the pivotal developments within our monitoring solution, underscoring the importance of the selected metrics and methodologies that contribute to the creation of a robust and efficient system. The process is initiated by extracting pertinent data from the designated systems. This data is subsequently relayed to a Central API, which acts as a nexus for data aggregation. The Webpage API is tasked with retrieving this data from the Central API, which it then forwards to the HTML page. Here, the data is elegantly presented in the form of charts, offering an intuitive visualization of the system's metrics for effective monitoring and analysis.

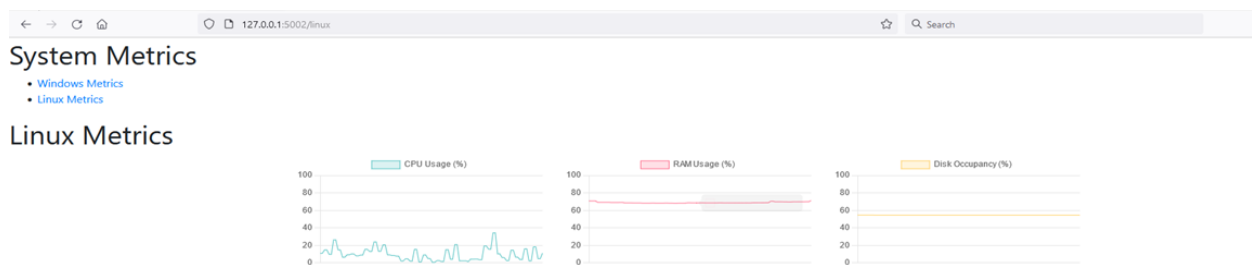


Figure 2.3: Results for Linux

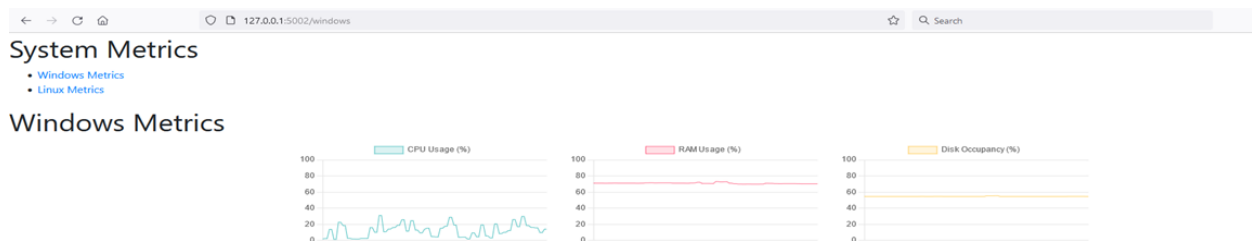


Figure 2.4: Results for Windows

Chapter 3

Integration with OpenShift

3.1 Red Hat OpenShift

Red Hat OpenShift[9] is an enterprise-grade Kubernetes container platform designed to facilitate automated operations across hybrid cloud, multicloud, and edge deployments. The platform aims to enhance developer productivity by offering a supportive environment for innovation and rapid development.

Integral to OpenShift is its inclusion of an enterprise-grade Linux operating system and a comprehensive ecosystem encompassing a container runtime, networking, monitoring, registry, and robust authentication and authorization solutions. It provides automated life-cycle management, resulting in heightened security, customizable operational solutions, manageable cluster operations, and versatile application portability.

As a Platform as a Service (PaaS) [17] offering enabled for cloud deployment, OpenShift transitions traditional application infrastructures from physical and virtual mediums to the cloud. The platform's open-source nature supports a multitude of applications, streamlining their development and deployment on the OpenShift cloud platform. Platform as a service (PaaS) is a complete development and deployment environment in the cloud, with resources that enable the consumer to deliver everything from simple cloud-based apps to sophisticated, cloud-enabled enterprise applications. An organization can purchase the resources it needs from a cloud service provider on a pay-as-you-go basis and access them over a secure Internet connection.

OpenShift caters to a broad spectrum of applications across various programming languages such as Ruby, Node.js, Java, Perl, and Python. One of OpenShift's defining characteristics is its extensibility, which allows for the integration of applications written in additional languages not natively supported, further emphasizing its flexibility and adaptability to diverse development needs.

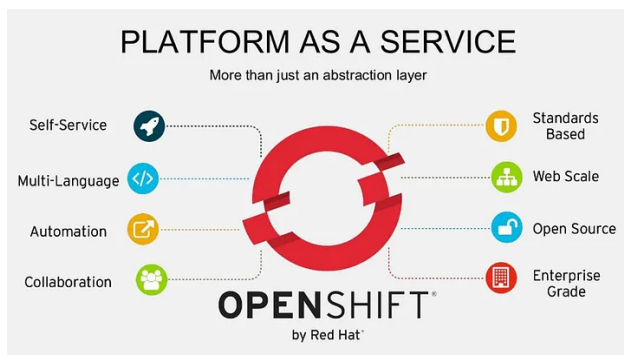


Figure 3.1: Redhat Openshift [12]

3.2 Containers: A Modern Approach to Application Deployment

Containers^[14] have emerged as a pivotal technology in modern application deployment, offering a lightweight and efficient method to run applications. They represent a significant evolution from traditional and virtualized deployment methodologies, characterized by enhanced portability and resource efficiency.

3.2.1 Evolution of Deployment Strategies

Understanding the emergence of container technology requires a look at the historical progression of deployment strategies.

Traditional Deployment Era

In the traditional deployment model, applications were directly run on physical servers without the capability to define resource boundaries. This often led to resource allocation conflicts, where one application could dominate resources, detrimentally impacting others. The alternative, running each application on a separate physical server, resulted in resource underutilization and increased costs.

Virtualized Deployment Era

Virtualization addressed some of these challenges by allowing multiple Virtual Machines (VMs) to run on a single physical server. VMs provided isolation and security by encapsulating an application and its environment, including a full operating system. This approach improved resource utilization and scalability, but it also introduced overhead due to the need to run multiple operating systems.

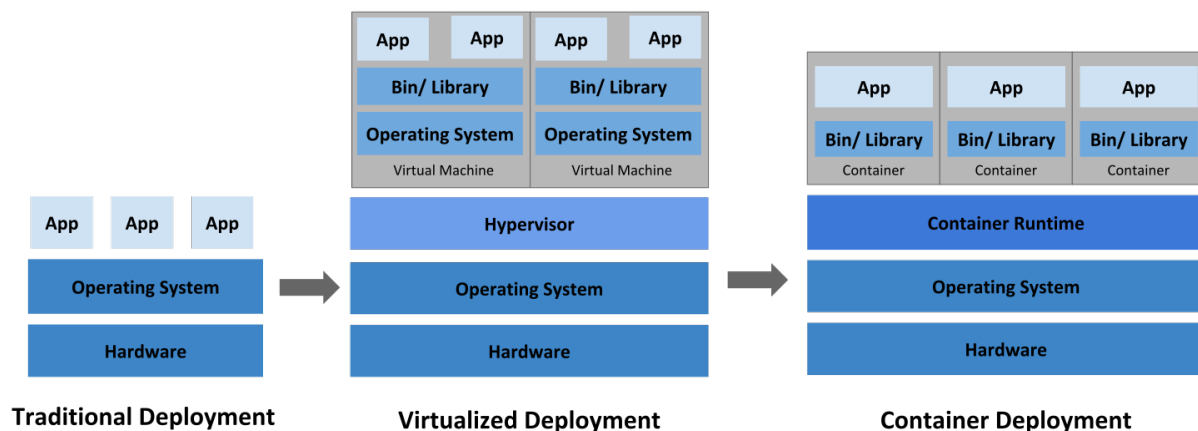


Figure 3.2: Evolution of containers^[7]

Container Deployment Era

Containers emerged as a solution that offers isolation similar to VMs but without the overhead of running multiple operating systems. Sharing the host system's OS, containers are lightweight and require less resources. This approach streamlines the deployment process, making it faster and more efficient, particularly in dynamic, scalable environments like cloud computing.

In summary, the transition from traditional to virtualized, and ultimately to containerized deployment, represents a significant technological shift towards more efficient, scalable, and cost-effective application deployment methodologies.

3.3 Kubernetes Overview

Kubernetes^[7] is an open-source platform designed for managing containerized workloads and services. It excels in automating, scaling, and operating these containerized applications, offering both declarative configuration and extensive automation capabilities. The platform boasts a large and rapidly expanding ecosystem with a wide range of available services, support, and tools.

Originating from the Greek word for 'helmsman' or 'pilot,' Kubernetes, often abbreviated as K8s (reflecting the eight letters between "K" and "s"), was open-sourced by Google in 2014. It synthesizes Google's extensive experience in running production workloads at scale with the community's best practices and innovations.

In the context of containerized applications, Kubernetes addresses the operational challenges in production environments. It ensures efficient container management, handling tasks such as automatic replacement and scaling of containers to ensure continuous operation and minimal downtime.

As a resilient framework for distributed systems, Kubernetes not only facilitates scaling and fail safes of applications but also provides patterns for deployment, such as canary deployments, enhancing the robustness and efficiency of the application delivery process. Canary deployments are a strategy for updating the deployment. This is a strategy where new versions of an application are gradually rolled out to a small subset of users or environments first. This method allows for testing and validation under real conditions before a full deployment, thereby minimizing potential disruptions.

3.3.1 How Kubernetes Works

Kubernetes [11] functions as a container orchestrator, essentially serving as an administrator for operating a fleet of containerized applications. It automates tasks such as restarting containers or allocating resources, simplifying the management of containerized applications.

Kubernetes introduces several concepts and components in its architecture:

Pods: The smallest unit in Kubernetes, a pod, is a group of containers sharing the same resources. Containers in a pod are treated as a single application, sharing the same IP address and resources like memory and storage.

Deployments: These define the scale and replication of pods on Kubernetes nodes, describing the number of replicas and the update strategy for the application.

Services: Services play a critical role in application architecture. They act as a stable front for a set of pods, often providing load balancing and ensuring that the service remains accessible even as individual pods are created, destroyed, or updated. This decouples the front-facing part of an application from the backend pods, enhancing the overall system's reliability and flexibility. Since individual pod lifetimes are not reliable, services provide a stable interface to groups of pods, abstracting away their internal changes and inconsistencies.

Nodes: These are the machines (virtual or physical) that run and manage pods, performing the tasks assigned by Kubernetes.

The Kubernetes Control Plane: This is the central management entity of Kubernetes, where operations are issued via HTTP calls or command-line scripts.

Cluster: A cluster represents the entire set of Kubernetes components functioning together.

Control Plane Components:

- **API Server:** Exposes a REST interface to the cluster.
- **Scheduler:** Assigns work to nodes, managing resources effectively.
- **Controller Manager:** Ensures the cluster's shared state operates as expected.

Worker Node Components:

- **Kubelet:** Manages the state of pods, providing regular updates to the control plane.
- **Kube Proxy:** Routes incoming traffic to the appropriate containers.
- **etcd:** A distributed key-value store used by Kubernetes for cluster state management.

These components collectively ensure that Kubernetes orchestrates containerized applications efficiently, maintaining the overall health and performance of the system.

3.4 The Relationship Between OpenShift and Kubernetes

Red Hat OpenShift and Kubernetes^[10] play integral roles in the realm of cloud-native applications, each serving distinct but complementary functions.

Red Hat OpenShift is an enterprise-grade open source application platform designed to accelerate the development and delivery of cloud-native applications. It ensures consistency across hybrid and multi-cloud environments, extending all the way to the edge. Powering OpenShift is Kubernetes, the container orchestration engine, complemented by a range of features from the Cloud Native Computing Foundation (CNCF) open source ecosystem. Red Hat packages and supports these features, offering a comprehensive and integrated application platform. OpenShift can be consumed as a public cloud service from major cloud providers like AWS, Microsoft Azure, Google, and IBM, or self-managed on various infrastructures, spanning data centers, public clouds, and edge environments.

Kubernetes, in contrast, is an open source software focusing on automating, deploying, managing, and scaling containers. While Kubernetes provides a solid foundation for container orchestration, it requires additional capabilities such as automation, monitoring, log analytics, and developer tools to be fully enterprise-ready. Integrating these capabilities is a manual process for organizations using Kubernetes alone.

The primary difference between **OpenShift and Kubernetes** lies in these added components. Kubernetes orchestrates containers but requires integration with other elements like networking, storage, and CI/CD pipelines for comprehensive application development and delivery. OpenShift, on the other hand, offers these components out-of-the-box with Kubernetes at its core, recognizing that Kubernetes alone is not sufficient for the accelerated development and scalable delivery of containerized applications.

In summary, while Kubernetes provides the essential orchestration capabilities, OpenShift extends this foundation by integrating additional tools and services, making it a more holistic platform for enterprise application development.

Containers, the fundamental building blocks in these ecosystems, are lightweight, standalone, executable packages that include everything needed to run an application: code, runtime, system tools, and libraries. This technology ensures consistency across various computing environments, facilitating the deployment and scaling of applications.

3.5 Docker Overview

Docker^[4] stands as an open platform that revolutionizes the development, shipment, and operation of applications. It distinctly separates applications from the underlying infrastructure, facilitating rapid software delivery. Docker's approach to infrastructure management mimics application management, promoting efficiency and agility.

At the core of Docker is its capability to package and execute an application within a lightly isolated environment known as a container. These containers operate concurrently on a host without dependency on the host's system installations, offering both isolation and security. Lightweight in nature, containers encapsulate everything necessary to run an

application, ensuring consistency across different working environments.

Docker simplifies the development, distribution, and execution of applications through container technology. A Dockerfile, containing Linux commands, creates a Docker image, which serves as a template for containers. These images are efficiently layered and shared across containers.

The Docker Engine, an integral part of Docker's architecture, is driven by the Docker daemon, which manages and runs Docker containers. The Engine is responsible for the entire lifecycle of containers, from building and starting to stopping and removing them. It facilitates communication between various Docker components and ensures that containers are isolated and secure.

Additionally, Docker images, the blueprints for containers, are stored and retrieved from the Docker Registry. This Registry can be private, for internal use within an organization, or public, like Docker Hub or Google Container Registry, allowing for broader access and sharing. The flexibility of using either private or public registries aids in version control and secure distribution of images. This architecture not only streamlines application deployment but also simplifies the process of updating and distributing applications across diverse environments, enhancing both efficiency and scalability. Docker's utility extends to the entire lifecycle management of containers, encompassing:

- Development of applications and their components within containerized environments.
- Utilization of containers as units for distribution and testing, ensuring uniformity across various stages of development.
- Deployment of applications into production environments, whether they be on local data centers or cloud-based infrastructures, as containers or orchestrated services.

Docker's significant role in the development process can be seen in its support for fast, consistent delivery of applications. By providing standardized environments for local development, Docker enhances continuous integration and continuous delivery (CI/CD) workflows. An example scenario would be:

- Developers write and share code using Docker containers, promoting collaborative and consistent development.
- Applications are pushed to testing environments, where both automated and manual tests are conducted.
- Any discovered bugs are fixed in the development environment and redeployed for validation.
- Once testing concludes, deploying the updates to customers is streamlined, involving merely updating the production environment with the new container image.

The platform's portability enables Docker containers to run across various environments, from local laptops to cloud services. This flexibility is key for responsive deployment and scaling of workloads. Additionally, Docker's efficiency allows for increased workload management on existing hardware, offering a cost-effective solution for both dense environments and smaller-scale deployments.

Figure 3.3: Containers in docker[4]

3.6 How to Build a Container

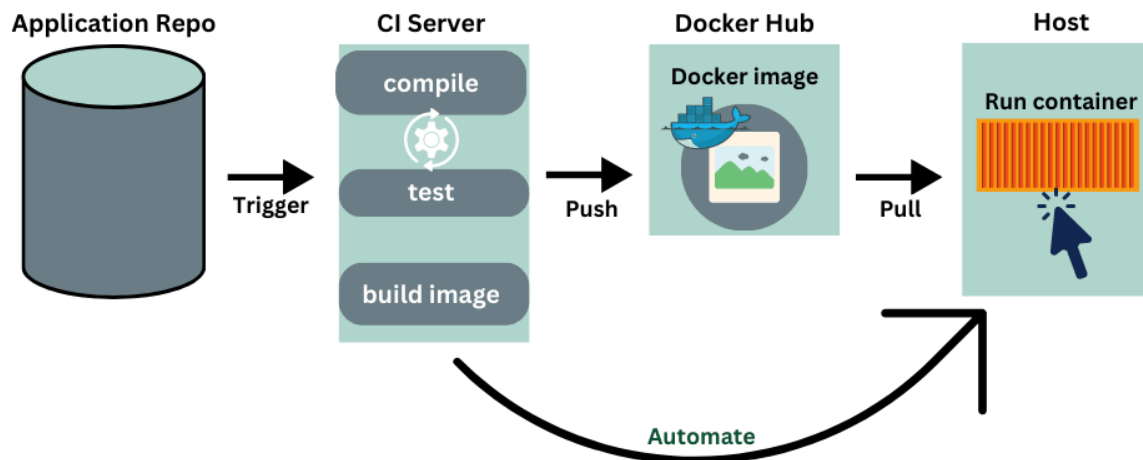


Figure 3.4: How to build a container[6]

3.6.1 Dockerfile Creation

The process begins with the creation of a Dockerfile, which is essentially a script that defines the steps for assembling a Docker image. A Dockerfile will specify the base image, set the working directory, install any necessary dependencies and establish the runtime environment for the application. The order of Dockerfile instructions matters. A Docker build consists of a series of ordered build instructions. Each instruction in a Dockerfile roughly translates to an image layer [5]. The following diagram 3.5 illustrates how a Dockerfile translates into a stack of layers in a container image.

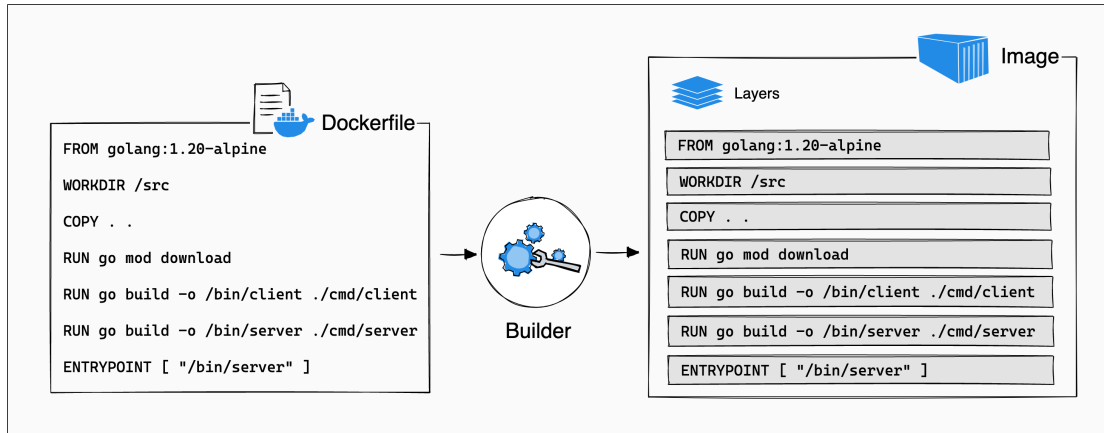


Figure 3.5: Dockerfile to layers[5]

```
FROM python:3.9
# Using the base image of python
EXPOSE 8081/tcp
# Exposing a port to communicate with our API
ENV FLASK_PORT=8081
# Sets an environment variable
WORKDIR /app
# Sets the working directory
COPY . .
# Copies all the required files to the containers working directory
RUN pip install -r requirements.txt
# Runs a command to install all requirments using pip
CMD ["python", "./stats2.py"]
# Run the code to start the API
```

Figure 3.6: Dockerfile

Docker also stores each layer in its cache so when you run a build, the builder attempts to reuse layers from earlier builds. If a layer of an image is unchanged, then the builder picks it up from the build cache. If a layer has changed since the last build, that layer, and all layers that follow, must be rebuilt. Example: The Dockerfile used in the project: Code snippet: 3.6

3.6.2 Building the Docker Image

The Dockerfile is then used to construct a Docker image by executing the Docker build command. This command requires the name and location of the image to be provided. The resulting Docker image functions as a blueprint for the container, encapsulating the application's code, its runtime environment, and all dependencies. Code snippet: 3.7

```
docker build -t my-python-app
```

Figure 3.7: docker build

3.6.3 Running the Container

Following the creation of the Docker image, the container is initiated using the `docker run` command. This operation launches the container predicated on the image specified and commences the execution of the application within a self-contained environment. Code snippet: 3.8

```
docker run my-python-app
```

Figure 3.8: docker run

3.6.4 Container Operations

Docker includes a suite of commands for container management. For instance, `docker ps` is used to list active containers, `docker stop` to terminate a running container, and `docker rm` to delete a container that is no longer in use.

3.7 Reasons to Use Red Hat OpenShift

Red Hat OpenShift was chosen as our application hosting platform for the following key reasons:

- **Reliability:** The project necessitates uninterrupted monitoring of systems to promptly detect and address anomalies. Red Hat OpenShift’s architecture ensures high availability with built-in fail-safes that maintain continuous operation. This level of reliability is crucial to ensure the API’s persistent uptime and the integrity of monitoring processes.
- **Scalability:** As the scope of the monitoring expands, scalability becomes essential. The environment provided by Red Hat OpenShift facilitates the efficient scaling of our containerized applications. The platform’s support for containers means additional systems can be accommodated to monitor with minimal code adjustments, leveraging the inherent resource efficiency and scalability of container technology.

These features of Red Hat OpenShift align with our project’s core requirements, providing a stable and adaptable foundation for our system monitoring solutions.

3.8 Implementation Steps

This section details the procedure for deploying containers on Red Hat OpenShift and will detail 2 ways to deploy a pod to run the container. Both methods are similar but in method one Redhat Openshift uses an image of the container to deploy the pod. Method 2 uses only a Dockerfile as reference and the container image is built by Openshift itself.

3.8.1 Method 1

1. With the APIs' code finalized, a Dockerfile is crafted for each API.
2. In the Dockerfile's directory, execute the "docker build" command to create the corresponding docker image for each API.
3. Following the creation of the docker images, the Red Hat OpenShift CLI(command-line interface) is leveraged to upload these images to OpenShift's image registry.
4. After uploading the images to OpenShift's repository, the container can be deployed using the OpenShift web user interface.
5. It is essential to note that during deployment, we must specify the exposed ports for each API to ensure they are accessible from other systems.

3.8.2 Method 2

1. Similar to Method 1, start by creating a Dockerfile for each of the APIs once the code is complete.
2. Establish two separate repositories on GitHub, one for each API's container.
3. With the code hosted, the containers can be deployed directly through the OpenShift web user interface using the Dockerfile.
4. Unlike Method 1, where custom ports can be specified, this method defaults to port 8080 and port 8081. We need to adjust this setting in the YAML file before creating a pod.
5. After correctly configuring the port, containers can be spun up and the specified ports become accessible from other systems.

These methodologies outline the essential steps for container deployment on Red Hat OpenShift, offering flexibility in the management and scaling of our API services.

3.9 Summary

This chapter has provided an in-depth exploration of Red Hat OpenShift and its pivotal role in contemporary application deployment. We have discussed the evolutionary shift from traditional and virtualized deployments to containerized solutions, emphasizing the critical advantages of containers in terms of portability and resource efficiency. Docker's contribution to this landscape as an enabling technology for containers was highlighted, with its capabilities for simplifying application delivery and management. Kubernetes was introduced as an orchestrator for these containers, providing the automation and scalability required for modern distributed systems. The discussion proceeded with an examination of the concrete steps to build and deploy containers using Red Hat OpenShift, detailing two methods that leverage Docker and OpenShift's own capabilities. We concluded with a rationale for choosing Red Hat OpenShift, focusing on its reliability and scalability, which are indispensable for the consistent monitoring and performance required by our project.

Chapter 4

Real-Time Monitoring

4.1 Introduction

With the increasing complexity of digital infrastructures, real-time monitoring has become crucial for maintaining system health and operational efficiency. The overarching goal of our monitoring process is twofold: to ensure all systems are functioning optimally and to minimize downtime across the network, thereby swiftly restoring any compromised system to its optimal state.

This chapter will explore the design considerations and challenges inherent in establishing effective real-time monitoring. We address critical aspects such as detecting system outages, analyzing performance metrics for anomalies.

The subsequent sections will detail the architecture of our monitoring system, the specific metrics we track, and the mechanisms in place to alert us to potential issues. By laying out the structure and flow of our real-time monitoring solution, this chapter aims to provide insights into building resilient and responsive monitoring systems capable of adapting to an ever-evolving digital landscape.

4.2 Architecture for Real-Time Monitoring

The architecture of our real-time monitoring system is founded on several key components, each essential for effective and continuous monitoring. The system is constructed atop the solution used for monitoring these systems, more specifically, it is integrated into the Central API. This monitoring solution depends on the data gathered through the API, which it subsequently analyzes.

4.2.1 In-Memory Data Storage

Metrics data for both Linux and Windows systems are stored in-memory using Python dictionaries. This approach facilitates quick access and manipulation of the monitoring data.

4.2.2 Background Scheduler

APScheduler^[2], a Python library, is employed for scheduling tasks. This component is responsible for periodically checking system metrics, crucial for real-time monitoring.

4.2.3 Metric Collection Routes

The Flask application defines specific routes for collecting metrics data. These routes are essential for receiving real-time data from different systems and processing them accordingly.

4.2.4 Data Analysis and Anomaly Detection

The system includes a module for analyzing incoming metrics and detecting anomalies, primarily based on threshold values, to identify potential issues in real-time. Overall, this architecture forms the backbone of our real-time monitoring system, enabling effective and continuous tracking of system health and performance.

4.3 Detecting Outages

This section outlines our approach for detecting outages or interruptions in communication. To achieve this, we employ a function that checks whether the Central API is continuously receiving metrics from the relevant systems.

4.3.1 The function

The `check_all_metrics`(as shown in code snippet: 4.1) function plays a crucial role in this process. It periodically checks the timestamp of the last metric received from both Linux and Windows systems. Here's a brief description of the code's functionality:

- The function first obtains the current time and compares it with the last received metric time for each system.
- Some measures are implemented to ensure that the corrective actions are not issued too often and this will further be discussed in more detail in the next chapter.
- In such cases, We have to issue the corrective actions, which we will discuss in the next chapter.
- This alert mechanism helps in promptly identifying and addressing potential system outages.

```
def check_all_metrics():
    current_time = datetime.datetime.now()

    if linux_metrics and any(linux_metrics):
        last_received_time_str = max(linux_metrics.keys())
        last_received_time = datetime.datetime.strptime(\
            last_received_time_str, "%Y-%m-%d %H:%M:%S")

        time_difference = (current_time - last_received_time).\
            total_seconds()
        print(time_difference)
        if time_difference >= cooldown_period_no_metrics_in_seconds:
            message = "Alert: No metrics received from linux system."
            send_teams_message(message)
            last_no_metrics_alert_timestamps["linux"] = current_time
    else:
        print(linux_metrics)
        message = "Alert: No metrics received from linux system."
        send_teams_message(message)
        last_no_metrics_alert_timestamps["linux"] = current_time

    if windows_metrics and any(windows_metrics):
        last_received_time_str = max(windows_metrics.keys())
        last_received_time = datetime.datetime.strptime(\
            last_received_time_str, "%Y-%m-%d %H:%M:%S")

        time_difference = (current_time - last_received_time).\
            total_seconds()
        print(time_difference)
        if time_difference >= cooldown_period_no_metrics_in_seconds:
            message = "Alert: No metrics received from windows system."
            send_teams_message(message)
            last_no_metrics_alert_timestamps["windows"] = current_time
    else:
        print(windows_metrics)
        message = "Alert: No metrics received from windows system."
        send_teams_message(message)
        last_no_metrics_alert_timestamps["windows"] = current_time
```

Figure 4.1: Function that checks for outages

4.3.2 Running the function at a set interval

To ensure continuous monitoring, the `check_all_metrics` function must operate at regular intervals. This periodic execution is facilitated by the `APScheduler` library, which is used to schedule tasks within a Flask application.

The relevant code segment sets up the `APScheduler`'s `BackgroundScheduler`, which is responsible for running the `check_all_metrics` function at predefined intervals(as shown in code snippet: 4.2). This interval is determined by the variable `metrics_check_interval`, ensuring that the system consistently checks for the receipt of metrics. The scheduler is then started, and the Flask application is run with these scheduled checks in place.

```
if __name__ == '__main__':
    scheduler = BackgroundScheduler()
    scheduler.add_job(check_all_metrics, 'interval',\
seconds=metrics_check_interval)
    scheduler.start()
    port = 8081
    port = int(port)
    app.run(port=port,host='0.0.0.0')
```

Figure 4.2: Code to run the function at set intervals

This approach of using a background scheduler is integral to maintaining real-time awareness of system status. It allows the monitoring system to autonomously detect outages or interruptions in metric flow, thereby enhancing the reliability and responsiveness of the overall monitoring solution.

4.4 Detecting Anomalies

This section focuses on the methods employed to detect anomalies in the metrics we gather. A key strategy is to analyze the metrics immediately upon receipt, providing close to real-time information for prompt response. We primarily identify anomalies by looking for abnormally high average percentages, which then trigger corrective actions. We use averages instead of individual cases to prevent a temporary peak from triggering a corrective action. The code below(as shown in figure: 4.3) plays a crucial role in this process:

- Two Flask routes, `/linux-metrics` and `/windows-metrics`, receive POST requests containing metric data for Linux and Windows systems, respectively.
- Upon receiving the data, the system records a timestamp and stores the metrics.
- It prints the received metrics along with the timestamp for logging purposes.
- The `check_metrics` function is called to analyze the metrics for each system. This function is designed to detect any values that exceed predefined thresholds, indicating

potential anomalies. The function first calculates the average of the last 3 metrics received and then checks if it is above the specified threshold.

- If anomalies are detected, appropriate actions are initiated to address these irregularities.

```
@app.route('/linux-metrics', methods=['POST'])
def receive_linux_metrics():
    data = request.get_json()
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    linux_metrics[timestamp] = data
    print(f"Received Linux Metrics at {timestamp}: {data}")
    check_metrics("Linux",data)
    return jsonify({'status': 'success'})

@app.route('/windows-metrics', methods=['POST'])
def receive_windows_metrics():
    data = request.get_json()
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    windows_metrics[timestamp] = data
    print(f"Received Windows Metrics at {timestamp}: {data}")
    check_metrics("Windows",data)
    return jsonify({'status': 'success'})
```

Figure 4.3: Code to check the metrics as soon as they are received.

To enhance our anomaly detection, we promptly pass the received metrics through the `check_metrics` function at the respective endpoints. This immediate evaluation allows for swift identification of any abnormal metric values.

4.4.1 The function

The `check_metrics` function checks if any of the metrics are experiencing particularly high usage or loads for a set amount of time. Note that temporary spikes in the load can be ignored as these spikes do not impact the performance of the system . The code below(as shown in code snippet: 4.4) plays a crucial role in this process:

- It iterates through each metric received, comparing the metric value against a predefined threshold, which is set at 90% in this case.
- When looking at metrics we look at the average of the last 3 received metrics to make sure that temporary spikes do not trigger any actions.

- Some measures are implemented to ensure that the corrective actions are not issued too often and this will further be discussed in more detail in the next chapter.

```
def check_metrics(system,metrics):
    if len(metrics)>3:
        m1_key=list(metrics.keys())[-1]
        m2_key=list(metrics.keys())[-2]
        m3_key=list(metrics.keys())[-3]
        last_3_metrics = [metrics[m1_key],metrics[m2_key],metrics[m3_key]]
        avg_cpu=0
        avg_ram=0
        avg_dsk=0
        for metric in last_3_metrics:
            avg_cpu = avg_cpu + metric["cpu_percent"]
            avg_dsk = avg_dsk + metric["disk_occupancy"]
            avg_ram = avg_ram + metric["ram_percent"]
        avg_metrics={"cpu_percent":avg_cpu/3,"ram_percent":avg_ram/3,\
"disk_occupancy":avg_dsk/3}
        for metric_name, metric_value in avg_metrics.items():
            threshold=90
            if metric_value > threshold:
                current_time = datetime.datetime.now()
                last_alert_time = last_alert_timestamps[(system,\
metric_name)]

                if (current_time - last_alert_time).total_seconds() >= \
cooldown_period_in_seconds:
                    message = f"Alert: Metric '{metric_name}' ({system}) \
is above {threshold}% - Value: {round(metric_value,2)}"
                    send_teams_message(message)
                    last_alert_timestamps[(system, metric_name)]\
= current_time
```

Figure 4.4: Code to Check for Anomalies

This implementation allows for the effective monitoring of metrics in real-time, ensuring that any significant deviations are addressed.

4.5 Conclusion

In summary, this chapter has delved into the intricate details of setting up a robust real-time monitoring system. We have explored the system's architecture, emphasizing its reliance on the Flask framework for metric collection, the use of in-memory data storage for efficiency, and the crucial role of the **APScheduler** for consistent monitoring. The process of detecting system outages and anomalies has been outlined, highlighting our methodological approach and the relevant code implementations. Through this comprehensive examination, the chapter has illuminated the intricacies and challenges of real-time monitoring, underscoring its significance in maintaining optimal system performance and minimizing downtime. This setup not only ensures that potential issues are swiftly identified and addressed but also sets the stage for the subsequent corrective actions, forming a cohesive and responsive monitoring ecosystem.

Chapter 5

Corrective actions

5.1 Introduction to Corrective Actions

Corrective actions are crucial interventions in scenarios where systems operate sub-optimally or go completely offline. These actions, aimed at restoring system functionality and optimal performance, are an essential component of our project's monitoring strategy. This section introduces the concept of corrective actions and their significance in maintaining system stability and efficiency.

5.2 Design Considerations for Corrective Actions

When designing corrective actions, we considered several automated solutions, such as terminating specific programs or services, removing obsolete files, and system restarts. However, these approaches posed potential disruptions to developers' work by not considering the context or circumstances of system usage. To address this, our chosen strategy involves notifying a system administrator responsible for the affected system. This respectful approach ensures that any necessary actions are taken with consideration for the developers intent and requirements. The implementation of this solution involves automated notifications being sent to a designated Microsoft Teams channel, enabling system administrators to promptly and effectively address issues.

5.3 Criteria for Sending Notifications

In this section we will go over all the factors taken into consideration before sending a message on MS teams.

The 2 scenarios where corrective actions need to be issued are when outages or anomalies are detected. However due to how the check for outages and anomalies is implemented sending a notification every time would just result in spamming of the MS teams channel. To avoid this a system where any information regarding any disruptive situation is communicated in an effective manner without cluttering the channel must be implemented. To meet these requirements a system of cool-downs was created. This system stores the time information of every notification sent to MS teams along with the type of information that was sent. So whenever the monitoring solution identifies a situation when a notification has to be sent it checks when a similar message has been sent last. If the time difference is less than the arbitrary amount of time chosen the message will not be sent. This amount of time that needs to elapse before sending a notification of the same type will be called the cool-down period. We can see how this is implemented in the code by taking a look at code

snippets: 4.1 , 4.4. For snippet: 4.1: Although the time data for when the message was sent regarding the outage last is stored, we don't use it as this situation is more urgent and needs to be resolved swiftly. Here the notification is only sent when the time interval for not receiving metrics is exceeded. This time interval is required to compensate for any possible lag in communications between the APIs and not just to avoid spamming the channels.

For snippet: 4.4: Whenever a notification is sent, the name of the system and the type of metric that crossed the threshold along with the time data is stored. This data is checked the next time the same anomaly is detected and only if the time difference between the current time and the time last sent exceeds the cool-down time will a notification be sent out.

5.4 Implementing Notifications in MS Teams

This section details the technical implementation of sending automated alerts and notifications through Microsoft Teams, as part of the corrective action process. To automate the process of sending notifications on MS teams we have to interact with the MS teams with our code. To do this an Incoming Webhook URL for our channel needs to be generated and then create a function that can use this URL to send messages on the channel. An Incoming Webhook is similar to the endpoint of an API in terms of function. However the difference is an API polls frequently to make sure the data it gets is real-time but an Incoming Webhook directly delivers the data to the app without the app needing to poll. The implementation is shown code snippets: A.1,A.2,A.3.

5.4.1 Making the Webhook

Before making the Incoming Webhook a new channel with all the relevant people is created. The Incoming Webhook can be created through the user interface. This process will be detailed in section : A

5.4.2 Using the Webhook

Once we have the Webhook URL we have to use it in our code to send notifications to the MS teams channel. This is done by simply using the requests library in python to send a post request with the payload and headers to the URL. The Payload consist of the message and the type of message. The header contains the content type.

```
def send_teams_message(message):
    headers = {"Content-Type": "application/json"}
    payload = {
        "text": message
    }

    response = requests.post(teams_webhook_url, data=json.dumps(payload),\
headers=headers)

    if response.status_code == 200:
        print("Message sent successfully.")
    else:
        print(f"Failed to send message. Status code: \
{response.status_code},Response: {response.text}")
```

Figure 5.1: Function to send a message on MS teams

5.5 Case Studies in Corrective Actions

Assessing all the scenarios when it comes to monitoring systems and sending notifications on MS teams. All systems should behave in the same way irrespective of operating system so in this case study the Linux system is chosen as our system being monitored. The metric CPU will be used to check for anomalies as the other metrics are also handled in the same way. All scenarios assume that both the APIs are online at all times and there is uninhibited communication between the APIs and the Linux system.

Cases checking whether there is an outage:

- Case 1: Linux system down, no previous metrics
Outcome: Notification is sent to the MS Teams channel.
- Case 2: Linux system down, previous metrics available, time difference exceeds threshold
Outcome: Notification is sent to the MS Teams channel.
- Case 3: Linux system down, previous metrics available, time difference acceptable
Outcome: No notification is sent to the MS Teams channel.
- Case 4: All metrics available and received regularly
Outcome: No notification is sent to the MS Teams channel.

Cases where all conditions of case 4 are true along with its outcome regarding the sending of the teams message about outages.

- Case 5: CPU metrics higher than threshold, no previous CPU alert for Linux system
Outcome: Notification sent to the Teams channel specifying CPU and Linux system.
- Case 6: CPU metrics higher than threshold, previous CPU alert exists, time difference exceeds Cooldown
Outcome: Notification sent to the Teams channel specifying CPU and Linux system.
- Case 7: CPU metrics higher than threshold, previous CPU alert exists, time difference within cooldown
Outcome: No notification sent to the Teams channel.
- Case 8: CPU metrics lower than threshold
Outcome: No notification sent to the Teams channel.

5.6 Conclusion

This chapter has comprehensively covered the framework and implementation of corrective actions within our monitoring system. We have explored the thoughtful design considerations that prioritize minimal disruption to developers while maintaining system integrity. The detailed criteria for triggering notifications and the methodical approach to employing Microsoft Teams for alerts underscore our commitment to effective and responsive system management. Through various case studies, we have demonstrated the practical application and adaptability of our methods in real-world scenarios. Overall, the corrective actions chapter underscores the importance of a balanced approach in automated system monitoring, ensuring prompt response to issues while respecting the operational context. This approach exemplifies our commitment to maintaining system stability and efficiency in a dynamic and demanding digital environment.

Chapter 6

Results

6.1 Results

This section of the thesis presents the results and effectiveness of the implemented automated monitoring solution. The evaluation focuses on assessing the system's performance, reliability, and effectiveness in real-time monitoring and corrective actions.

6.1.1 Performance Metrics

When it comes to measuring the impact and effectiveness of the solution we can use key metrics like system uptime, response time to anomalies and outages, and accuracy in anomaly detection.

6.1.2 Adaptability of the system

The solution is containerized due to which the solution is extremely salable. What this means is that when the number of systems to be monitored increase more containers running the same application can be spun up to accommodate the demand. All the python code is also very modular so in the future any changes that need to be made to system can be done easily, this adds to the adaptability of the system.

6.1.3 Effectiveness of Monitoring and Corrective Actions

The effectiveness of the monitoring solution was evaluated in real time based on its ability to detect system anomalies and the timeliness of corrective actions. Results, as shown in Figure 6.1, indicate a prompt response to identified issues, with the appropriate notification being sent to teams, this notifies the admins who can take appropriate steps to ensure system stability. Note in the figure 6.1 that for demonstrative purposes we have set the threshold to 50% so that we can actually see the relevant message in teams when checking for anomalies and this is reflected in the image. This threshold is set to 90% in the actual deployed solution.

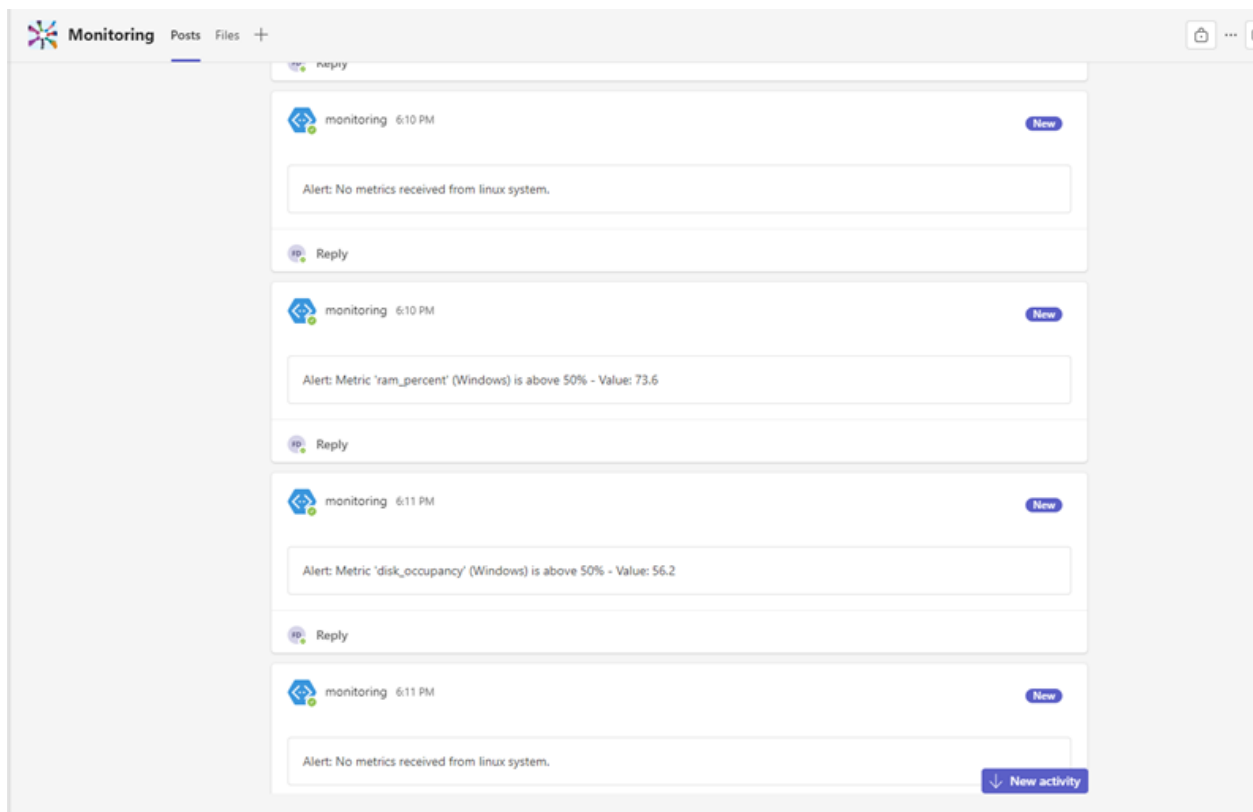


Figure 6.1: Enter Caption

With this system in place the admins that are responsible have a much faster response time in the case of any outages. Since the system also flags anomalies any potentially disruptive event, these events will also get flagged early and dealt with accordingly. This system has succeeded in reducing down time of the monitored system.

6.1.4 Conclusion

Overall, the implemented solution has proved effective in maintaining system health and performance. This section highlighted the solution's adaptability and reliability, proving that it is a viable option for real-time monitoring in heterogeneous system environments. Further enhancements and continuous monitoring are recommended to ensure sustained system efficiency.

Chapter 7

Conclusion

This thesis has successfully demonstrated the development process and all the steps taken to deploy a monitoring solution tailored for diverse and complex system environments.

The thesis began with a comprehensive exploration of the challenges in real-time monitoring and why it is essential in the modern IT infrastructure environment. The thesis then further explores what needs to be done when something is detected which leads to the implementation of effective corrective actions. Through the integration of technologies like Docker, Kubernetes, and Openshift, a scale-able, resilient and adaptable architecture was established. The implementation of automated monitoring, coupled with a thoughtful approach to corrective actions, ensures minimal system downtime and maximized operational efficiency.

The results chapter confirmed the solution's efficacy using key performance metrics like system uptime, and anomaly response times. The effectiveness of the monitoring strategy in a heterogeneous environment has showcased the system's ability to adapt and respond to various operational scenarios.

To conclude, the thesis has evidently achieved its objective of designing a reliable and efficient monitoring solution. The insights gained and the methodologies developed here will have significant implications for future advancements in system monitoring. Continuous improvements and adaptations to incorporate new and emerging technologies will further enhance the effectiveness of such solutions.

In closing, this thesis serves as a testament to the potential of integrating new technologies and cutting edge tools to create a sophisticated monitoring system. It lays a foundation for future research in the field of system monitoring and management, especially in the context of increasingly complex modern IT infrastructure.

Appendix A

Setting up Incoming Webhooks

A.0.1 Generating the Webhook

Navigate to the top right of the Channel and click on the options box and then select connectors. Once the page for connectors opens up search for Incoming Webhooks and click on add or configure if channel has existing Incoming Webhooks. After naming it clicking create will generate the URL. This Incoming Webhook URL is used in the code to send messages. The implementation can be seen in the code snippet: 5.1.

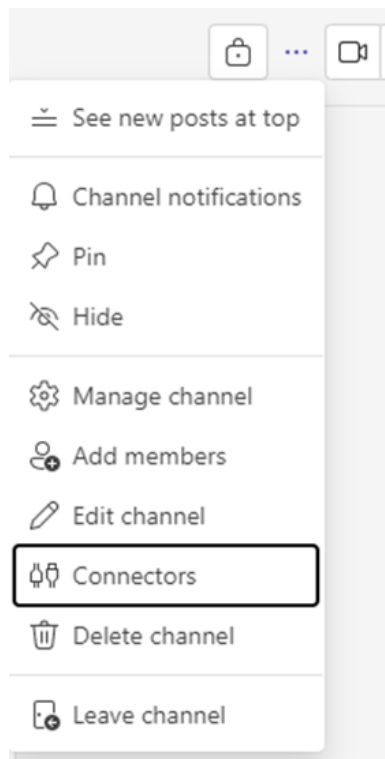


Figure A.1: Options in Channel

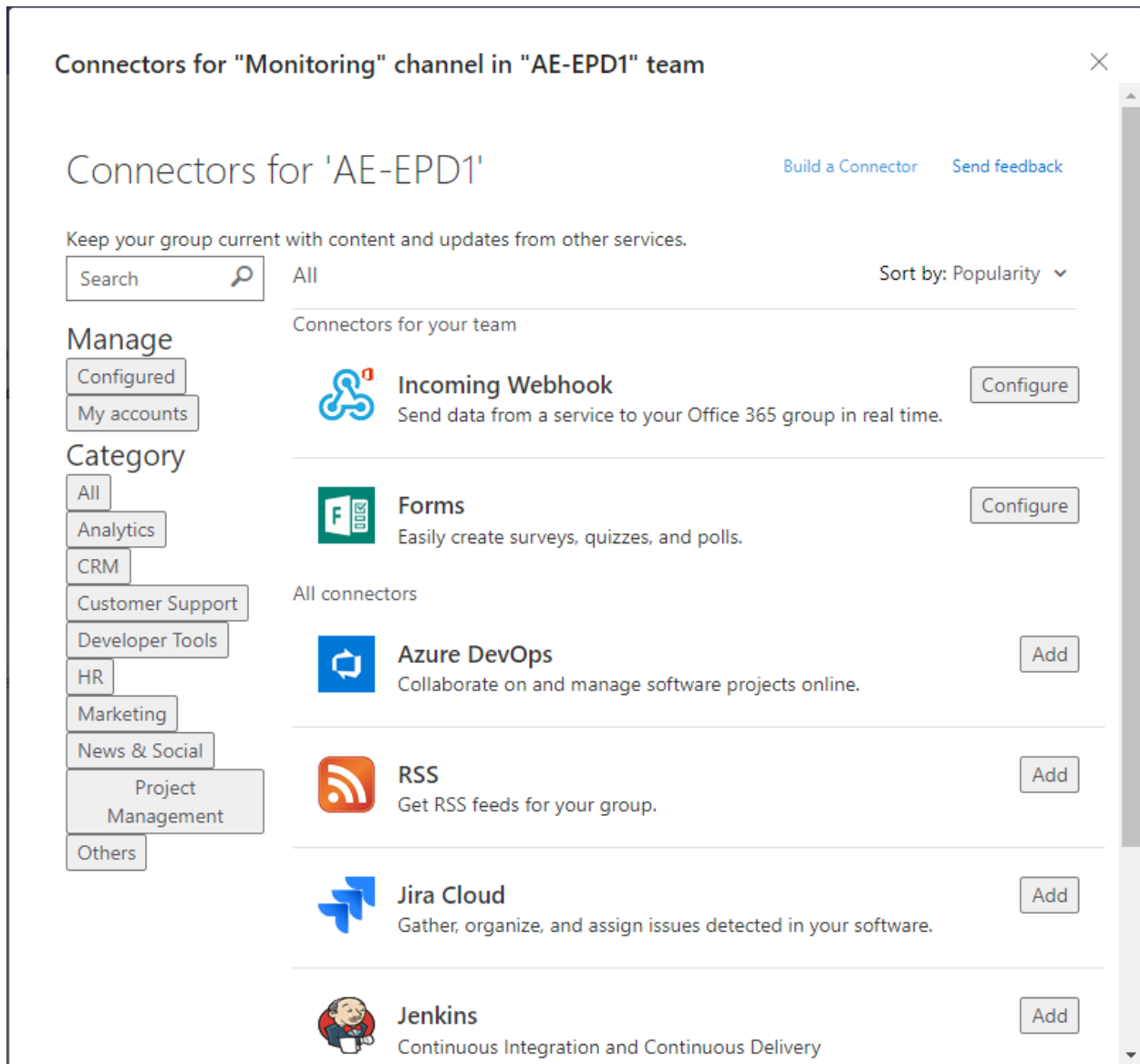



Figure A.2: Available connectors

Connectors for "Monitoring" channel in "AE-EPD1" team



Incoming Webhook

[Send feedback](#)


The Incoming Webhook connector enables external services to notify you about activities that you want to track. To use this connector, you'll need to create certain settings on the other service, which needs to support a webhook that's compatible with the [Office 365 connector format](#).

Fields marked with * are mandatory

To set up an Incoming Webhook, provide a name and select Create. *

Customize the image to associate with the data from this Incoming Webhook.

Upload Image


Default Image

Create

Cancel

Note: If you're a software developer and want to learn more about sending data to Office 365 using Incoming Webhook, see [Get started with Office 365 Connector Cards](#).

Figure A.3: Webhook creation

Bibliography

- [1] *API*. en. Page Version ID: 1193905608. Jan. 2024. URL: <https://en.wikipedia.org/w/index.php?title=API&oldid=1193905608> (visited on 01/08/2024).
- [2] *APScheduler: In-process task scheduler with Cron-like capabilities*. URL: <https://github.com/agronholm/apscheduler> (visited on 01/19/2024).
- [3] baeldung. *What Is an API Endpoint? — Baeldung on Computer Science*. en-US. Feb. 2023. URL: <https://www.baeldung.com/cs/api-endpoints> (visited on 01/08/2024).
- [4] *Docker overview*. en. 200. URL: <https://docs.docker.com/get-started/overview/> (visited on 01/08/2024).
- [5] *Layers*. en. 200. URL: <https://docs.docker.com/build/guide/layers/> (visited on 01/15/2024).
- [6] *Learn How to Build and Push a Docker Image to a Container Registry*. en. Mar. 2023. URL: <https://dev.to/pavanbelagatti/learn-how-to-build-and-push-a-docker-image-to-a-container-registry-282> (visited on 01/08/2024).
- [7] *Overview*. en. URL: <https://kubernetes.io/docs/concepts/overview/> (visited on 01/08/2024).
- [8] *psutil documentation — psutil 5.9.8 documentation*. URL: <https://psutil.readthedocs.io/en/latest/> (visited on 01/08/2024).
- [9] *Red Hat OpenShift enterprise Kubernetes container platform*. en. URL: <https://www.redhat.com/en/technologies/cloud-computing/openshift> (visited on 01/08/2024).
- [10] *Red Hat OpenShift vs. Kubernetes: What's the difference?* en. URL: <https://www.redhat.com/en/technologies/cloud-computing/openshift/red-hat-openshift-kubernetes> (visited on 01/19/2024).
- [11] *Sensu — How Kubernetes works*. en. URL: <https://sensu.io> (visited on 01/16/2024).
- [12] Gursimar Singh. *OpenShift Case Study*. en. May 2021. URL: <https://gursimarm.medium.com/openshift-case-study-7082865402d1> (visited on 01/08/2024).
- [13] *Welcome to Flask — Flask Documentation (3.0.x)*. URL: <https://flask.palletsprojects.com/en/3.0.x/> (visited on 01/08/2024).
- [14] *What is a Container? — Docker*. en-US. URL: <https://www.docker.com/resources/what-container/> (visited on 01/08/2024).
- [15] *What is an API Endpoint?* URL: <https://smartbear.com/learn/performance-monitoring/api-endpoints/> (visited on 01/08/2024).
- [16] *What is an API? A Beginner's Guide to APIs — Postman*. en. URL: <https://www.postman.com/what-is-an-api/> (visited on 01/08/2024).
- [17] *What is PaaS? Platform as a Service — Microsoft Azure*. en-US. URL: <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-paas> (visited on 01/08/2024).